A Course Note

for

Scala language: A functional approach to data science

at the School of Information Science, KMUTT

in 2018

Part I. Statistical Computing with Scala: A functional approach to data science by Darren J Wilkinson (2017) [1)]

Part II. Tour of Scala [2)]

November 23, 2018

Contents

I Statistical Computing with Scala: A functional approach to data science by Darren J Wilkinson (2017) [1)] 11

1 Introduction 13

1.1 Introduction to Scala and functional programming . . . 13
1.1.1 Statistical computing . . . 13
1.1.2 Functional Programming, concurrency, parallel programming and shared mutable state 14
1.1.3 What is scala? . . . 15
1.1.4 Scala tools and libraries . . . 17
1.1.5 Hello World . . . 21
1.2 Basic tools for Scala development . . . 21
1.2.1 Java and the JVM ecosystem . . . 21
1.2.2 Scala, scalac, and the Scala REPL . . . 21
1.2.3 Note: Scala script . . . 22
1.2.4 SBT . . . 22
1.2.5 The Scala IDE . . . 24
1.2.6 Emacs and Ensime . . . 24
1.2.7 Summary . . . 25

2 Scala and FP basics [1)] 27

2.1 Basic concepts and syntax . . . 27
2.1.1 val and var . . . 27
2.1.2 Immutable lists . . . 28
2.1.3 Immutable vectors . . . 29
2.1.4 Type inference . . . 30
2.1.5 Defining methods . . . 31
2.1.6 Complete programs . . . 34
2.2 Functional programming in Scala . . . 35
2.2.1 HOFs, closures, partial application and currying . . . 35
2.3 Note: Scala IO . . . 41

3 The Scala collections library **43**
3.1 Scala collections overview and basics . . . 43
3.1.1 Introduction . . . 43
3.1.2 The collections hierarchy . . . 44
3.1.3 Writing collection–generic code . . . 50
3.1.4 Writing type–generic code . . . 51
3.1.5 Writing (type– and) collection–generic code returning a collection . . . 51
3.1.6 Writing type-generic numerical code . . . 52
3.2 Parallel collections . . . 53
3.2.1 Introduction . . . 53
3.2.2 Writing parallelism-agnostic code . . . 55
3.3 Collections as monads . . . 56
3.3.1 First steps with monads in Scala . . . 56
3.3.2 Option monad . . . 58
3.3.3 The Future monad . . . 62

4 Scientific and statistical computing **63**
4.0.1 Scala math standard library: scala.math . . . 63
4.0.2 Note: Spire cfor macro . . . 64
4.1 Breeze: A quick introduction . . . 65
4.1.1 Introduction . . . 65
4.1.2 Non-uniform random number generation . . . 65
4.1.3 Vectors and matrices . . . 67
4.1.4 Reading and writing matrices from and to disk . . . 70
4.2 Breeze: Numerical linear algebra . . . 70
4.2.1 Symmetric eigen-decomposition . . . 70
4.2.2 Singular value decomposition (SVD) . . . 70
4.2.3 Principal components analysis (PCA) . . . 73
4.2.4 QR and Cholesky decompositions . . . 74
4.3 Breeze: Scientific computing . . . 76
4.3.1 Constants and special functions . . . 76
4.3.2 Integration and ODE-solving . . . 77
4.3.3 Optimisation . . . 77
4.4 Breeze–Viz . . . 78

5 Monte Carlo simulation **83**
5.1 Parallel Monte Carlo . . . 83
5.1.1 Monte Carlo integration . . . 83
5.1.2 Parallel implementation . . . 84

5.1.3 Timings, and varying the size of the parallel collection . . . 84
5.1.4 Rejection sampling . . . 86
5.2 Markov chain Monte Carlo . . . 86
5.2.1 Metropolis-Hastings algorithms . . . 87
5.2.2 Factoring out the updating logic . . . 89
5.2.3 A Gibbs sampler . . . 93

6 Statistical modelling **99**
6.1 Linear regression . . . 99
6.1.1 Introduction . . . 99
6.1.2 Linear regression in Scala . . . 100
6.1.3 Case study: linear regression for a real dataset . . . 106
6.2 Generalised linear models . . . 109
6.2.1 Introduction . . . 109
6.2.2 Logistic regression . . . 110
6.2.3 Poisson regression . . . 113
6.3 The scala-glm library . . . 115
6.4 Data frames and tables in Scala . . . 116
6.4.1 Spark DataFrames . . . 117
6.4.2 Summary . . . 118

7 Tools and libraries: Scala Tools **119**
7.1 SBT tips and tricks . . . 119
7.1.1 sbt new . . . 119
7.1.2 Increasing heap memory . . . 120
7.1.3 Building standalone applications (assembly JARs) . . . 120
7.2 Interfacing Scala with R . . . 121
7.2.1 Calling R from Scala . . . 121
7.2.2 Calling Scala from **R** . . . 123
7.3 CSV parsing libraries . . . 126
7.4 Testing . . . 127
7.4.1 Assert and require . . . 127
7.4.2 scalatest . . . 128
7.4.3 scalacheck . . . 129
7.4.4 Benchmarking and profiling . . . 129
7.4.5 Documentation using ScalaDoc . . . 129
7.5 Note: How to execute (exec) external system commands in Scala [3)] . . . 129
7.5.1 The methods, ! and !!, added to the String class by sys.process.stringToProcess . . . 129
7.5.2 Also, sys.process.stringSeqToProcess can be used. . . . 130

7.5.3 "|" and "cd" are a shell built-in command. . . . 130
7.5.4 How can a pipe shell-command be mimiced in Scala? . . . 130
7.5.5 Scala operators mimic normal shell operators . . . 131

8 Apache Spark **133**
8.1 Getting started with the Spark Shell . . . 133
8.1.1 Introduction . . . 133
8.1.2 Getting started – installing Spark . . . 133
8.1.3 First Spark shell commands . . . 135
8.1.4 Caching RDDs with persist . . . 137
8.2 Commonly used RDD methods . . . 137
8.2.1 Transformations of an RDD[T] . . . 137
8.2.2 Actions on an RDD[T] . . . 138
8.2.3 Transformations of a Pair RDD, RDD[(K,V)] . . . 138
8.2.4 Actions on a RDD[(K,V)] . . . 138
8.2.5 DataFrames . . . 138
8.3 Analysis of quantitative data . . . 139
8.3.1 Descriptive statistics . . . 139
8.3.2 Linear regression . . . 140
8.3.3 Logistic regression . . . 143
8.4 Tuning regularisation parameters using crossvalidation . . . 145
8.5 Compiling Spark jobs . . . 147
8.5.1 Further reading . . . 149

9 Advanced topics **151**
9.1 Typeclasses and implicits . . . 151
9.1.1 Using implicits to add a method to a pre-existing class . . . 151
9.1.2 Defining and using a simple typeclass . . . 153
9.1.3 A parametrised typeclass, using higher-kinded types . . . 154
9.1.4 Case study: A scalable particle filter in Scala . . . 156

10 Appendix **161**
10.1 Scala API for tensorflow . . . 161

II Tour of Scala [2)]

163

11 Scala Tour **165**

11.1 Introduction . . . 165
11.1.1 Welcome to the tour . . . 165
11.1.2 What is Scala? . . . 165
11.1.3 Scala is object-oriented . . . 165
11.1.4 Scala is functional . . . 165
11.1.5 Scala is statically typed . . . 166
11.1.6 Scala is extensible . . . 166
11.1.7 Scala interoperates . . . 166
11.2 Basic . . . 166
11.2.1 Trying Scala . . . 166
11.2.2 Expressions . . . 167
11.2.3 Values . . . 167
11.2.4 Variables . . . 167
11.2.5 Blocks . . . 167
11.2.6 Functions . . . 168
11.2.7 Methods . . . 168
11.2.8 Classes . . . 169
11.2.9 Case Classes . . . 169
11.2.10 Objects . . . 170
11.2.11 Traits . . . 170
11.2.12 Main Method . . . 171
11.3 Unified Types . . . 171
11.3.1 Scala Type Hierarchy . . . 171
11.3.2 Type Casting . . . 172
11.3.3 Nothing and Null . . . 173
11.4 Classes . . . 173
11.4.1 Defining a class . . . 173
11.4.2 Constructors . . . 174
11.4.3 Private Members and Getter/Setter Syntax . . . 174
11.5 Traits . . . 175
11.5.1 Defining a trait . . . 175
11.5.2 Using traits . . . 176
11.5.3 Subtyping . . . 176
11.6 Class Composition With Mixins . . . 177
11.7 Higher Order Functions . . . 178
11.7.1 Coercing methods into functions . . . 179
11.7.2 Functions that accept functions . . . 179
11.7.3 Functions that return functions . . . 179

11.8 Nested Methods . . . 180
11.9 Multiple Parameter Lists (Currying) . . . 180
11.9.1 Single functional parameter . . . 181
11.9.2 Implicit parameters . . . 181
11.10 Case Classes . . . 181
11.10.1 Defining a case class . . . 181
11.10.2 Comparison . . . 182
11.10.3 Copying . . . 182
11.11 Pattern Matching . . . 183
11.11.1 Syntax . . . 183
11.11.2 Matching on case classes . . . 183
11.11.3 Pattern guards . . . 184
11.11.4 Matching on type only . . . 185
11.11.5 Sealed classes . . . 185
11.11.6 Notes . . . 186
11.12 Singleton Objects . . . 186
11.12.1 Defining a singleton object . . . 186
11.12.2 Companion objects . . . 187
11.12.3 Notes for Java programmers . . . 188
11.13 Regular Expression Patterns . . . 188
11.14 Extractor Objects . . . 190
11.15 For Comprehensions . . . 191
11.16 Generic Classes . . . 192
11.16.1 Defining a generic class . . . 192
11.16.2 Usage . . . 192
11.17 Variences . . . 193
11.17.1 Covariance . . . 193
11.17.2 Contravariance . . . 194
11.17.3 Intravariance . . . 195
11.17.4 Other Examples . . . 196
11.17.5 Comparison With Other Languages . . . 196
11.18 Upper Type Bounds . . . 196
11.19 Lower Type Bounds . . . 197
11.20 Inner Classes . . . 199
11.21 Abstract Types . . . 201
11.22 Compound Types . . . 202
11.23 Self-type . . . 203
11.24 Implicit Parameters . . . 203

11.25 Implicit Conversions . . . 205
11.26 Polymorphic Methods . . . 206
11.27 Type Inference . . . 206
11.27.1 Omitting the type . . . 206
11.27.2 Parameters . . . 207
11.27.3 When not to rely on type inference . . . 207
11.28 Operators . . . 208
11.28.1 Defining and using operators . . . 208
11.28.2 Precedence . . . 208
11.29 By-name Parameters . . . 209
11.30 Annotations . . . 210
11.30.1 Annotations that ensure correctness of encodings . . . 210
11.30.2 Annotations affecting code generation . . . 211
11.30.3 Java Annotations . . . 211
11.31 Default Parameter Values . . . 212
11.32 Named Arguments . . . 213
11.33 Packages and Imports . . . 213
11.33.1 Creating a package . . . 213
11.33.2 Imports . . . 214

Part I

Statistical Computing with Scala: A functional approach to data science by Darren J Wilkinson (2017) [1)]

Chapter 1

Introduction

1.1 Introduction to Scala and functional programming

1.1.1 Statistical computing

- The current state of serious statistical computing is far from ideal ...
- R has become the de facto standard programming language for statistical computing – the S language was designed by statisticians for statisticians in the mid 1970's, and it shows!
  - Many dubious language design choices, meaning it will always be inelegant, slow, dangerous and inefficient (without many significant breaking changes to the language)
  - R's inherent inefficiencies mean that much of the R code-base isn't in R at all, but instead in other languages, such as FORTRAN, C and C++
  - Although faster and more efficient than R, these languages are actually all even worse languages for statistical computing than R!
- The fundamental problem is that all of the programming languages commonly used for scientific and statistical computing were designed 30-50 years ago, in the dawn of the computing age, and haven't significantly changed
  - Think how much computing hardware has changed in the last 40 years!
  - But the languages that most people are using were designed for that hardware using the knowledge of programming languages that existed at that time
- We have learned just as much about programming and programming languages in the last 40 years as we have about everything else (including statistical methodology)
- Our understanding has developed in parallel with developments in hardware
- People have been thinking a lot about how languages can and should exploit modern computing hardware such as multi-core processors and parallel computing clusters
- Modern functional programming languages are emerging as better suited to modern hardware

1.1.2 Functional Programming, concurrency, parallel programming and shared mutable state

- FP languages emphasise the use of *immutable data, pure, referentially transparent functions*, and *higher-order functions*
- Unlike commonly used *imperative* programming languages, they are closer to the Church end of the *Church-Turing* ththesis – eg. closer to *Lambda-calculus* than a *Turing-machine*
- The original Lambda-calculus was *untyped*, corresponding to a dynamically-typed programming language, such as Lisp
- *Statically-typed* FP languages (such as *Haskell*) are arguably more scalable, corresponding to the *simply-typed Lambda-calculus*, very closely related to *Cartesian closed categories...*
- In pure FP, all state is immutable – you can assign names to things, but you can't change what the name points to – no "variables" in the usual sense
- Functions are pure and referentially transparent – they can't have side-effects – they are just like functions in mathematics...
- Functions can be recursive, and recursion can be used to iterate over recursive data structures – useful since no conventional "for" or "while" loops in pure FP languages
- Functions are first class objects, and *higher-order functions* (HOFs) are used extensively – functions which return a function or accept a function as argument
- Modern computer architectures have processors with several cores, and possibly several processors – parallel programming is required to properly exploit this hardware
- The main difficulties with parallel and concurrent programming using imperative languages all relate to issues associated with *shared mutable state*
- In pure FP, state is not mutable, so there is no mutable state, and hence no shared mutable state – most of the difficulties associated with parallel and concurrent programming just don't exist in FP – this has been one of the main reasons for the recent resurgence of FP languages
- We should approach the problem of statistical modelling and efficient computation in a modular, composable, functional way
- To do this we need programming languages which are:
  - *Strongly statically typed* (but with type inference)
  - *Compiled* (but possibly to a VM)
  - *Functional* (with support for immutable values, immutable collections, ADTs and higher-order functions)
  - and have support for *type-classes* and *higher-kinded types*, allowing the adoption of design patterns from *category theory*

    Very few languages have proper support for higher kinded types, with Scala and Haskell (and various Haskell derivatives) being the best known examples

- For efficient statistical computing, it can be argued that evaluation should be *strict* rather than *lazy* by default

  > The relative merits of lazy and strict evaluation is controversial. Haskell is lazy by default. This enables some very elegant solutions to programming problems, but leads to some unpredictable behaviour and inefficiencies. Scala is strict by default, but as we will see, supports lazy evaluation when required.

- *Scala* is a popular language which meets the above constraints

1.1.3 What is scala?

Scala

- is a *general purpose* language with a sizeable user community and an array of general purpose libraries, including good GUI libraries, The name Scala derives from "Scalable networking and web frameworks
- is free, *open-source* and *platform independent*, *fast* and efficient with a strong type system, and is *statically typed* with good compile-time type checking and *type safety*
- has a good, well-designed *library for scientific computing*, including non-uniform random number generation and linear algebra
- has reasonable *type inference* and a *REPL* for interactive use
- has good *tool support* (including build tools, doc tools, testing tools, Scala is not a pure FP language. It is and intelligent IDEs)
- has excellent support for *functional programming*, including support for *immutability* and immutable data structures and monadic design exploiting *higher kinded types*
- allows imperative programming for those (rare) occasions where it makes sense
- is designed with *concurrency* and *parallelism* in mind, having excellent language and library support for building really *scalable* concurrent and parallel applications
- 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 byte-code to run on the JVM
- It was originally developed by Martin Odersky, one of the authors of the Java compiler, javac as well as the creator of Java generics, introduced in Java 5.
- Development driven by perceived shortcomings of the Java programming language
- 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, Coursera, 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 compiletime 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 language like

- 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 Optional 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

- It is 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 sbt is a very powerful build-tool for 1M Java file, sbt-launch.jar

Scala vs. Java

- Scala has a set of features that completely differ from Java. Some of these are
  - All types are objects.
  - Type inference
  - Nested Functions
  - Functions are objects
  - Domain specific language (DSL) support
  - Traits
  - Closures
  - Concurrency support inspired by Erlang

1.1.4 Scala tools and libraries

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 – there is no need to "install" anything, ever – this means that 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

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, but in reality this is a minor part of the reproducibility problem — you 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 in the meantime
- 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
- There is a standard code documentation format, Scaladoc, an improved Scala version of Javadoc, and standard testing frameworks such as ScalaTest and ScalaCheck.

Example sbt build file (build.sbt)

```
name := "app-template"

version := "0.1"

scalacOptions ++= Seq(
  "-unchecked", "-deprecation", "-feature"
)

libraryDependencies  ++= Seq(
  "org.scalacheck" %% "scalacheck" % "1.14.0" % "test",
  "org.scalatest" %% "scalatest" % "3.0.5" % "test",
  "org.typelevel" %% "spire" % "0.15.0",
  "org.scalanlp" %% "breeze" % "0.13.2",
  "org.scalanlp" %% "breeze" % "0.13.2",
  "org.scalanlp" %% "breeze-natives" % "0.13.2"
)

scalaVersion := "2.12.6"
```

The **%%** in the **libraryDependencies** tells sbt that it should append the current version of Scala being used to build the library to the dependency's name; so **%** must be used for java libraries. Note that scala binaries compiled with different major versions such as 2.11 and 2.12 are not compatible to each other. A nearly equivalent, manual alternative for a fixed version of Scala is:

```
libraryDependencies += "org.scalanlp" % "breeze-natives_2.12" % "0.13.2"
```

IDEs

- 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
- The "Scala IDE" (scala-ide.org) is based on Eclipse (a popular IDE for Java programmers) – this is a good "first Scala IDE" for many people
- IntelliJ is another Java IDE which some prefer to Eclipse. It has a Scala plugin which is good and is getting progressively better.
- For users already familiar with Emacs, the ENhanced Scala Interaction Mode for Emacs (ENSIME) arguably makes the best Scala IDE.
- If you use an IDE, your code will (almost) always compile first time.

Data structures and parallelism

- Scala has an extensive "Collections framework", providing a large range of data structures for almost any task (Array, List, Vector, Map, Range, Stream, Queue, Stack, ...)

- Most are 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"

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

When you start functional programming you start to see terms like "functor", "monad", "monoid" and "applicative" being used. This can be a bit intimidating for new users. These terms are associated with a branch of mathematics known as "category theory". Due to the close connection between category theory and the typed lambda calculus that strongly typed functional programming languages are based on, it is natural to borrow concepts from category theory, and to name them accordingly. In fact, these concepts are not difficult, as the brief guide below illustrates.
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" (or "zip") 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 parallelise 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, using tree-reduction)

We will unpack these ideas properly later.

Scala ecosystem

There are many useful libraries in the Scala ecosystem. Some commonly encountered libraries are given below:

- Breeze – library for scientific and statistical computing
- Spire – numeric type-classes

- Akka – actor-based concurrency framework (inspired by Erlang)
- Apache Spark – big data framework built on Akka
- Tensorflow_Scala – Scala API for https://www.tensorflow.org for neural networks
- cats – category theory types (functors, monads, etc.)
- Shapeless – type-safe dynamic type support
- Play – web framework
- Scala.js – use Scala client-side by compiling to JS

Breeze

- Breeze is a Scala library for scientific and numerical computing
- It 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 netlibjava, 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)

Apache Spark

- Spark is a scalable analytics library, including some ML (originally from Berkeley's AMP Lab)
- It is rapidly replacing MapReduce as the standard library for big data processing and analytics
- It is written in Scala, but also has APIs for Java and Python (and an experimental API for R)
- Only the Scala API leads to both concise elegant code and good performance
- Central to the approach is the notion of a *resilient distributed dataset* (RDD) – hooked up to Spark via a *connector* – using RDDs is very similar to using other Scala collections

Calling R from Scala and vice versa

- R CRAN package rscala allows bi-directional calling between R and Scala
- Useful for calling out to computationally intensive routine written in Scala from an R session
- Also useful for calling to R from Scala for a model-fitting procedure "missing" from the Scala libraries
- Can inline R code in Scala files and Scala code in R files

(Calling Python from Scala and vice versa)

1.1.5 Hello World

```
object HelloWorld {
   def main(args: Array[String]) = {
      println("Hello, world")
   }
}
```

1.2 Basic tools for Scala development

1.2.1 Java and the JVM ecosystem

Scala is a JVM language. This means that Scala source code is compiled to Java byte-code designed to be run on a Java Virtual Machine (JVM). Typically a single **.scala** source code file will be compiled to a .class byte-code file, and a collection of (linked) .class files are often packaged together in a **.jar** Java ARchive file. Once a Scala application is packaged up into a stand-alone (assembly) JAR file, it should be possible to run it on any JVM on any machine (even machines with different architectures running different OSs). This platform-independence of JVM languages is very convenient for many reasons. In particular, it is one of the reasons for the popularity of JVM languages in Cloud computing and Big Data scenarios – you can easily develop on a laptop and then deploy your pre-built application to a cluster in the Cloud.

In principle, only a basic Java Runtime Environment (JRE) is required for compiling and deploying Scala applications. However, recent versions of **sbt** require a full Java Development Kit (JDK), so now you need a JDK for sbt-based development (compiling and assembling), but still require only a basic JRE for deployment (running the application).

Before attempting to use Scala you should make sure that you have a Java 8 JDK installed. It doesn't matter whether this JDK is the OpenJDK or Oracle's. On Debian-based Linux systems, this can be done with

```
$ sudo apt-get install openjdk-8-jdk
```

Other Linux-based systems should be similar. For other OSs, try the Oracle Java JDK download page. To check whether you have Java installed correctly, type **java -version** into a terminal window. If you get a version number of the form 1.8.x you should be fine.

1.2.2 Scala, scalac, and the Scala REPL

It is possible to do a system-wide installation of **scala** the Scala compiler, **scalac**, though it is not necessary. On Debian-based Linux systems, this can be done with

```
$ sudo apt-get install scala
```

If you have Scala installed, you can check which version with

```
$ scala -version
```

For other OSs, just download and install from the Scala language homepage. With Scala installed, you can start a Scala REPL just by typing **scala** at your command prompt.

The disadvantage of this approach is that it inevitably makes this one particular version of Scala "special" on your system. It also doesn't make it easy to run code with dependencies on libraries that are not part of the

Scala Standard Library. Using **sbt** is usually preferable for everything other than small stand-alone scripts, so we won't be assuming a systemwide Scala installation for this course.

1.2.3 Note: Scala script

Scala scripts can be made by inserting a few lines at the top of a Scala program as follows. This method is suitable to and convenient for small stand-alone scripts that depend only on the Scala standard library, but is not preferable for Scala programs that depend on other libraries because a well-organized system-wide Scala installation is required together with Scala libraries.

```
#! /bin/sh
exec scala -savecompiled "$0" "$@"
!#
 /////// A Scala program starts on this line. ///////////////////
 /////// Wrapping with an object is not required in a script. //
 ////////////////////////////////////////////////////////////////
object scalaScript {

  def main(args: Array[String]) = {
    val fin = if (args.size > 0) {
                val filename = args(0)
                new java.io.FileInputStream(filename)
              } else {
                java.lang.System.in
              }
    val bufsource = scala.io.Source.fromInputStream(fin)

    println(bufsource.size)
    bufsource.close
  }

}
 /////// A Scala program ends on this line. ////////////////////
```

1.2.4 SBT

SBT is the Scala Build Tool – it is the most widely used build-tool for Scala. Other than a recent Java installation, SBT is all you need for building Scala projects. In order to install SBT, Refer to
"https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Windows.html"
or "https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Linux.html".
(Detailed instructions for installing SBT and testing that it works are provided in the exercises of this course.)
Once SBT is installed, it can be started by typing **sbt** at a command prompt. Note that SBT creates files and directories, so it should only be started from a directory containing a Scala project. On running SBT you will get an SBT command-prompt, **>**. SBT is designed to be used interactively. Type **exit** or ctrl-D to exit back to your OS command prompt or **help** to get an SBT command summary. Type **compile** to compile the project, and **run** to run the application. Note that the **run** task has a dependency on the **compile** task, and so it is safe to just type **run** and a compilation will first be triggered if required. The **test** task will run any tests

associated with the project. The **console** task will start up a Scala REPL. Type Scala expressions and then **:quit** to exit the REPL back to the SBT prompt. Typing **:help** at the Scala REPL prompt will give a REPL command summary. Note that the REPL provided by the **console** task will include all library dependencies specified in the SBT build file and also all of the functions, classes and objects defined in this particular Scala project. **clean** is another commonly used SBT task (for deleting compiled artifacts), as is reload (after editing build.sbt).

```
# Note that SBT creates files and directories, so it should only be
started from a directory containing a Scala project.
$ pwd
directory-of-scala-project
$ sbt                # To start sbt.
>                    // ">" is sbt command-prompt.
> help               // for help message
> tasks              // to list rasks; tasks [-v|-vv|...] for details
> compile            // to compile
> run                // if required, compile and then run the project
> test               // to run any tests
> package            // Produces the main artifact, such as a binary jar.
> assembly           // Builds a deployable fat jar.
> clean              // for deleting compiled artifacts
> console            // to enter into Scala REPL
scala> :help         // for the Scala REPL comand summary
scala> :quit         // to exit the REPL back to the SBT
> exit               // to exit back to your OS command prompt
$                    #
```

Note that prefixing a task with a tilde ~ causes it to wait after completion and watch for changes to source code files and automatically re-run whenever any source code file changes. For example, running ~test will cause all tests to re-run whenever any project source code file is saved. Similarly for ~compile and ~run. SBT-based Scala projects usually assume a standard directory layout along the lines of

```
.
  build.sbt
  project
    build.properties
  src
    main
      scala
        scala-files.scala
        files
          scala-files.scala
    test
      scala
        scala-test-files.scala
        files
          scala-test-files.scala
```

Note that the version of SBT to be used is specified in the file **project/build.properties** with a line like **sbt.version=0.13.8**. All other library and version dependencies are specified in the top-level SBT build file **build.sbt**. (Note that The SBT version can't be specified in build.sbt, as this file is interpreted by the required version of SBT.) See pages 18 and 213 for a simple example of the file, build.sbt; also some simple examples of build files are included in the exercises.

Note that although SBT is mainly intended to be used interactively, it is also possible to use directly from an OS command prompt, so that typing **sbt run** at your OS command prompt is like running the run task from the SBT prompt. Read the official SBT Getting Started Guide [4)] for more information.

1.2.5 The Scala IDE

The Scala IDE, based on Eclipse, is the IDE most commonly used by programmers new to Scala. Note that to use the Scala IDE with SBT projects, you must also install the **sbteclipse** SBT plugin which provides a new SBT task, **eclipse**.

The main thing to understand is that the ScalaIDE needs to know about the structure of your sbt project. This information is encoded in Eclipse project files in the top-level directory of your sbt project (where the file **build.sbt** will often be present). An initial set of project files for an sbt project can be generated using the **eclipse** sbt task provided by the **sbteclipse** plugin.

So, before using the ScalaIDE with a particular SBT project for the first time, first run

```
$ sbt eclipse
```

to analyse the project and create eclipse project files for it. Then start the ScalaIDE. If it asks about a workspace, make sure you select something different to the SBT project directory. Then import the project using the *Import Wizard* (under the File menu) to import *Existing Projects into Workspace*. You may need to repeat this process if you make significant changes to the **build.sbt** file.

Once you are up-and-running, Eclipse provides fairly sophisticated IDE functionality. Some commonly used commands include:

- Shift-Ctrl-F – Reformat source file
- Shift-Ctrl-P – Go to matching bracket
- Ctrl-Space – Content assist
- Shift-Ctrl-W – Close all windows (from package explorer)

Scala worksheet

- Shift-Ctrl-B – Re-run all code

See the ScalaIDE Documentation for further information.

1.2.6 Emacs and Ensime

(omitted)

1.2.7 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.)

Chapter 2

Scala and FP basics [1)]

2.1 Basic concepts and syntax

2.1.1 val and var

We'll start with a few simple Scala concepts in order to introduce some syntax. There's a lot of introductory material on Scala on-line, and we will examine some of this during the practical sessions. Here we will just look at some of the most relevant concepts for this course.

```
bash$ scala
Welcome to Scala 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_151).
Type in expressions for evaluation. Or try :help.

scala> val a = 5
a: Int = 5

scala> a
res0: Int = 5
```

So far, so good. Using the Scala REPL is much like using the Python or R command line, so will be very familiar to anyone used to these or similar languages. The first thing to note is that labels need to be declared on first use. We have declared **a** to be a **val**. These are *immutable* values, which can not be just re-assigned, as the following code illustrates.

```
scala> a = 6
<console>:12: error: reassignment to val
       a = 6
         ^

scala> a
res1: Int = 5
```

The type for an object in the val and var statements can be explicitly specified.

```
val a: Double = 1
```

```
//a: Double = 1.0
val a: Int = 1.0
//<console>:11: error: type mismatch;
// found   : Double(1.0)
// required: Int
/       val a:Int = 1.0
//                  ^
```

Immutability can be a difficult concept for people unfamiliar with functional programming. But fear not, as Scala allows declaration of *mutable* variables as well:

```
scala> a
res1: Int = 5

scala> var b = 7
b: Int = 7

scala> b
res2: Int = 7

scala> b = 8
b: Int = 8

scala> b
res3: Int = 8
```

2.1.2 Immutable lists

The Zen of functional programming is to realise that immutability is generally a good thing, and we shall explore this more during the course. Scala has excellent support for both mutable and immutable collections as part of the standard library, and we shall examine these further in the next Chapter. For example, it has immutable lists.

```
val c = List(3,4,5,6)
//c: List[Int] = List(3, 4, 5, 6)
c(0)
//res4: Int = 3
c.sum
//res5: Int = 18
c.length
//res6: Int = 4
c.product
//res7: Int = 360
```

Again, this should be pretty familiar stuff for anyone familiar with dynamic languages such as R or Python. Note that the **sum** and **product** methods are special cases of *reduce* operations, which are well supported in Scala. For example, we could compute the sum reduction using

```
c.foldLeft(0)((x,y) => x+y)
//res8: Int = 18
```

or the slightly more condensed form given below, and similarly for the product reduction.

```
c.foldLeft(0)(_+_)
//res9: Int = 18
c.foldLeft(1)(_*_)
//res10: Int = 360
```

In fact, if the reduction is initialised with an *identity* of the operation (so that the values and operator together form a *monoid*), then we can use the even simpler **reduce** method.

```
c.reduceLeft(_*_)
//res11: Int = 360
```

Note: if the reduction function being applied is associative, then **foldLeft** and **reduceLeft** can be replaced by **fold** and **reduce**, respectively. See API and Visual Scala Reference for details.

2.1.3 Immutable vectors

Scala also has a nice immutable **Vector** class, which offers a range of (essentially) constant time operations (but note that this has nothing to do with the mutable **Vector** class that is part of the Breeze library)

```
val d = Vector(2,3,4,5,6,7,8,9)
//d: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4, 5, 6, 7, 8, 9)
d
//res12: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4, 5, 6, 7, 8, 9)
d.slice(3,6)
//res13: scala.collection.immutable.Vector[Int] = Vector(5, 6, 7)
val e = d.updated(3,0)
//e: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4, 0, 6, 7, 8, 9)
d
//res14: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4, 5, 6, 7, 8, 9)
e
//res15: scala.collection.immutable.Vector[Int] = Vector(2, 3, 4, 0, 6, 7, 8, 9)
```

Note that when **e** is created as an updated version of **d** the whole of **d** is not copied – only the parts that have been updated. And we don't have to worry that aspects of **d** and **e** point to the same information in memory, as they are both immutable. As should be clear by now, Scala has excellent support for functional programming techniques. In addition to the reduce operations mentioned already, maps and filters are also well covered.

```
val f=(1 to 10).toList
//f: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
f
//res16: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
f.map(x => x*x)
//res17: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

```
f map {x => x*x}
//res18: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
f filter {_ > 4}
//res19: List[Int] = List(5, 6, 7, 8, 9, 10)
```

Note how Scala allows methods with a single argument to be written as an infix operator, making for more readable code.

For most single expressions, it doesn't matter whether they are enclosed in regular parentheses or curly braces. However, braces allow the use of multiple expressions on new lines without semi-colons, so tend to be used more often by default

```
println({
  val x = 1
  x + 1
})
// 3
```

You can combine expressions by surrounding them with . We call this a block. The result of the last expression in the block is the result of the overall block, too.

2.1.4 Type inference

Given that Scala is a strongly statically typed language, it is possibly surprising that none of the code in the previous sessions has involved the manual writing of any kind of type annotation. As previously mentioned, Scala has type *inference* which means that type annotations are optional when the compiler can figure out what the types must be. In all of the previous examples, it is possible for the compiler to figure out the type of the result of each expression. But it is fine to include type annotations if desired

```
val a1 = 1
//a1: Int = 1
val a2: Int = 1
//a2: Int = 1
val l1 = List(1,2,3)
//l1: List[Int] = List(1, 2, 3)
val l2 : List[Int] = List(2,3,4)
//l2: List[Int] = List(2, 3, 4)
```

Sometimes it can be useful to include type annotations on numeric types, for example to ensure that a numeric constant is interpreted as a **Double**.

```
val a3 : Double = 1
//a3: Double = 1.0
val a4: Int = 1.0
//<console>:11: error: type mismatch;
// found   : Double(1.0)
// required: Int
//       val a4: Int = 1.0
//                     ^
```

But the second example shows that care must be taken with numeric types. There are no automatic *down-castings* or *type conversions*. Generally speaking, it is rare to see type annotations inside the body of methods or functions. The main place that type annotations are required are in the argument list of functions and methods. For type safety, it is usually considered good practice to also annotate the return type of functions and methods, though this is often not required.

Note: See 172 for type casting.

2.1.5 Defining methods

Let us now think about defining our own factorial function

$$n! = \prod_{i=1}^{n} i$$

There is a subtle distinction between methods and functions in Scala that we will return to shortly. For now it is fine to think of methods as functions.

The canonical functional definition is just

```
def fact1(n: Int): Int = (1 to n).product
//fact1: (n: Int)Int
fact1(5)
//res0: Int = 120
```

Note that we explicitly annotate the method argument **n** to be an Int. This annotation is required. We also annotate the return type of the method to be **Int**. This is not required, but it is good practice to do so. There is nothing really wrong with this definition, but for very large values of the argument **n**, this would involve creating a large collection in heap memory to then reduce using the **product** method. This isn't really a problem here, as the factorial function will overflow **Int** long before heap space is a problem. But conceptually we shouldn't need to consume significant heap space in order to evaluate this function. We could implement this function in an imperative style using mutable variables as follows.

```
def fact2(n: Int): Int = {
  var acc = 1
  var i = 2
  while (i <= n) {
    acc *= i
    i += 1
  }
  acc
}
//fact2: (n: Int)Int
fact2(5)
//res1: Int = 120
```

This works fine, but isn't very functional. The standard functional approach to this problem is to exploit recursion, as follows.

```
def fact3(n: Int): Int = {
  if (n == 1)
    1
  else
    n * fact3(n-1)
}
//fact3: (n: Int)Int
fact3(5)
//res3: Int = 120
```

There are numerous important things to notice here. The first is that in Scala **if** is an *expression* and not a *statement* – it simply returns a value depending on the outcome of evaluating the predicate. The next thing to note is that although **n** is an immutable value, the value it takes depends on the value it is passed when it is called, and this can be different at different times. Further, there is no problem with the function calling itself recursively, although annotating the function with its return type is required in this case.

Although the above function is *recursive*, it is not *tail-recursive*, in the sense that the value of the recursive value has to be modified (by multiplying by n) in order to obtain the return value of the function. This means that each recursive call consumes a stack frame, and so very deep recursions of this kind can cause a *stack overflow*. Contrast this with the following.

```
@annotation.tailrec
def fact4(n: Int, acc: Int = 1): Int = {
  if (n == 1)
    acc
  else
    fact4(n-1, acc*n)
}
//fact4: (n: Int, acc: Int)Int
fact4(5)
//res4: Int = 120
```

Here we avoid manipulating the return of the recursive call by passing through an *accumulator* which can be returned directly on termination. This makes the function *tail recursive*. Knowing that a function is tail recursive is important in Scala, as the Scala compiler can (usually) perform *tail call elimination*. That is, the compiler will re-write the tail recursive function in a form not dissimilar to the imperative version, **fact2**. This does not consume stack frames, and hence will not blow the stack. The annotation, @annotation.tailrec is not required in order for tail call elimination to be performed. Its purpose is simply to cause the compiler to report an error if tail call elimination can not be performed. If you have a function that you think is tail recursive you usually want to know if the compiler can't perform tail call elimination, and so it is a good idea to always use the annotation on any method or function that you consider should be tail recursive. Also note that this function introduces *default arguments* – if this function is called without a second argument, **acc** defaults to a value of 1.

The factorial function quickly overflows **Int** for large values of the argument **n**. It is therefore simpler to illustrate the stack problem using the log-factorial function, $\log(n!)$. We can compute this without first computing **n!** using the fact that

$$\log(n!) = \sum_{i=1}^{n} \log(i)$$

We can write a simple function to recursively compute this as follows.

```
import math.log
math.log(fact4(5))
//res2: Double = 4.787491742782046
def lfact(n: Int): Double = {
  if (n == 1)
    0.0
  else
    math.log(n) + lfact(n-1)
}
//lfact: (n: Int)Double
lfact(5)
//res3: Double = 4.787491742782046
lfact(10000)
//res4: Double = 82108.92783681415
//lfact(10000) // will cause stack overflow
```

Evaluating this function for an argument of 10,000 should certainly not overflow **Double**, but does blow the stack – try it! Contrast this with the tail recursive version:

```
@annotation.tailrec
def lfacttr(n: Int, acc: Double = 0.0): Double = {
  if (n == 1)
    acc
  else
    lfacttr(n-1, acc + math.log(n))
}
//lfacttr: (n: Int, acc: Double)Double
lfacttr(5)
//res5: Double = 4.787491742782046
lfacttr(10000)
//res6: Double = 82108.92783681446
```

This one works fine for even large arguments since it is not gobbling up stack frames.

> This technique of making a non-tail- recursive function tail-recursive by introducing and explicitly passing a continuation for the computation is a very simple example of a powerful functional programming technique known as *continuation passing* style.

If you do want to work with large factorials directly, this is also possible: just switch from **Int** to **BigInt**, as follows.

```
@annotation.tailrec
def factbi(n: BigInt, acc: BigInt = 1): BigInt = {
  if (n == 1)
    acc
  else
    factbi(n-1, acc*n)
```

```
}
//factbi: (n: scala.math.BigInt, acc: scala.math.BigInt)scala.math.BigInt
factbi(5)
//res15: scala.math.BigInt = 120
factbi(10000)
//res16: scala.math.BigInt = 284625968091705451890641321211986889014805l4...
```

2.1.6 Complete programs

So far we have been using the Scala REPL interactively. But we typically want to build complete programs that can be compiled and then run (usually using SBT). The Scala compiler expects that all methods will be contained in either **Classes** or **Objects**, and that the entry point to the program will be a method called **main** with signature **main(args: Array[String]): Unit**, where **args** will contain any command–line arguments for the program. **Unit** is the Scala type containing a single value, **()**. Here it is used to signal that the **main** method does not return a useful value. A simple example program for computing a log-factorial and printing the result to screen is given below.

```
/*
log-fact.scala
Program to compute the log-factorial function
*/
object LogFact {

  import annotation.tailrec
  import math.log

  @tailrec
  def logfact(n: Int, acc: Double = 0.0): Double =
    if (n == 1)
      acc
    else
      logfact(n-1, acc + log(n))

  def main(args: Array[String]): Unit = {
    val n = if (args.length == 1) args(0).toInt else 5
    val lfn = logfact(n)
    println(s"logfact($n) = $lfn")
  }

}
//eof
```

If you have a system-wide scala installation, this program could be run from the OS command line, if nessary, with particular arguments, as follows.

```
bash$ scala log-fact.scala 100 1000
```

We can also use SBT to build and run Scala applications. If the file **log-fact.scala** is placed into an empty directory, SBT can be started from this directory (no build file is necessary), and then the code can be

compiled and run with the **run** task. It can be run with a particular argument by typing, say, **run 100** at the SBT prompt. If you want to run the program directly from an OS command prompt, you can do so with

```
sbt run
```

If you want to use a non-default argument, you must quote the argument together with the **run** task as

```
sbt "run 100 1000"
```

2.2 Functional programming in Scala

We have already started to think about how recursion can be used in place of iteration in order to avoid mutating state. It's now time to look at some other techniques that are widely used in functional programming.

2.2.1 HOFs, closures, partial application and currying

Introduction

Functional programming (FP) emphasises the use of referentially transparent pure functions and immutable data structures. Higher order functions (HOFs) tend to be used extensively to enable a clean functional programming style. A HOF is just a function that either takes a function as an argument or returns a function. For example, the default **List** collection in Scala is immutable. So, if one defines a list via

```
val l1 = List(1,2,3)
//l1: List[Int] = List(1, 2, 3)
```

we add a new value to the front of the list by creating a new list from the old list and leaving the old list unchanged:

```
val l2 = 4 :: l1
//l2: List[Int] = List(4, 1, 2, 3)
```

We can create a new list the same length as an existing list by applying the same function to each element of the list in turn using **map**:

```
val l3 = l2 map { x => x*x }
//l3: List[Int] = List(16, 1, 4, 9)
```

We could write this slightly differently as

```
val l4 = l2.map(x => x*x)
//l4: List[Int] = List(16, 1, 4, 9)
```

which makes it clearer that **map** is a higher order function on lists. HOFs are ubiquitous in FP, and very powerful. But there are a few techniques for working with functions in Scala (and other FP languages) which make creating and using HOFs more convenient.

> **Note:** x => $x*x$, where the type inference is used to get x : *Double* => $x*x$, is called a function literal. See 168 for function literals.

Plotting a function of one scalar variable

There are many, many reasons for using functions and HOFs in scientific and statistical computing (optimising, integrating, differentiating, or sampling, to name just a few). But the basic idea can be illustrated simply by considering the problem of plotting a function of one scalar variable. For this we will use the plotting library breeze-viz which is not part of the Scala standard library, so the following code should be run from an SBT console from a project directory containing a build file with a dependence on **breeze-viz**. We will start by defining a function, **plotFun**, to plot on the screen the graph of a real valued function of a single real parameter. We will look at **breeze** and **breeze-viz** in more detail tomorrow so it doesn't matter if you don't understand exactly how this function works.

```
import breeze.plot._
def plotFun(fun: Double => Double, xmin: Double = -3.0, xmax: Double = 3.0):
  Figure = {
  val f = Figure()
  val p = f.subplot(0)
  import breeze.linalg._
  val x = linspace(xmin, xmax)
  p += plot(x, x map fun)
  p.xlabel = ""
  p.ylabel = "f(x)"
  f
}
//plotFun: (fun: Double => Double, xmin: Double, xmax: Double)breeze.plot.Figure
```

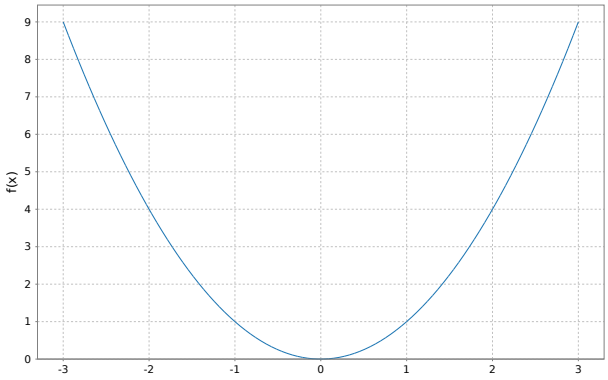


Figure 2.1. Plots of a quadratic function

Once we have this defined, we can use it to start plotting functions. For example,

```
plotFun(x => x*x)
//res0: breeze.plot.Figure = breeze.plot.Figure@6f1a16fe
```

produces the plot shown in Fig. 2.1.

We can use this method to plot other functions, for example:

```
def myQuad1(x: Double): Double = x*x - 2*x + 1
//myQuad1: (x: Double)Double
```

```
plotFun(myQuad1)
//res4: breeze.plot.Figure = breeze.plot.Figure@611640f0
def myQuad2(x: Double): Double = x*x - 3*x - 1
//myQuad2: (x: Double)Double
plotFun(myQuad2)
//res5: breeze.plot.Figure = breeze.plot.Figure@6215366a
```

Now technically, **myQuad1** and **myQuad2** are methods rather than functions. The distinction is a bit subtle, and they can often be used interchangeably, but the difference does sometimes matter, so it is good to understand it. To actually define a function and plot it, we could instead use code like:

```
val myQuad3: (Double => Double) = x => -x*x + 2
//myQuad3: Double => Double = <function1>
plotFun(myQuad3)
//res6: breeze.plot.Figure = breeze.plot.Figure@8bd076a
```

Note that here myQuad3 is a *value* whose type is a *function* mapping a **Double** to a **Double**. This is a proper function. This style of function declaration should make sense to people coming from other functional languages such as Haskell, but is potentially confusing to those coming from O-O languages such as Java. Note that it is easy to convert a method to a function using an underscore, so that, for example, **myQuad2 _** will give the function corresponding to **myQuad2**. Note that there is no corresponding way to get a method from a function, so that is one reason for preferring method declaration syntax (and there are others, such as the ability to parametrise method declarations with generic types). Now, rather than defining lots of specific instances of quadratic functions from scratch, it would make more sense to define a generic quadratic function and then just plot particular instances of this generic quadratic. It is simple enough to define a generic quadratic with:

```
def quadratic(a: Double, b: Double, c: Double, x: Double):
Double = a*x*x + b*x + c
//quadratic: (a: Double, b: Double, c: Double, x: Double)Double
```

But we clearly can't pass that in to the plotting function directly, as it has the wrong type signature (not Double =>Double), and the specific values of a, b and c need to be given. This issue crops up a lot in scientific and statistical computing – there is a function which has some additional parameters which need to be fixed before the function can actually be used. This is sometimes referred to as the "function environment problem". Fortunately, in functional languages it's easy enough to use this function to create a new "partially specified" or "partially applied" function and pass that in. For example, we could just do

```
plotFun(x => quadratic(3,2,1,x))
//res7: breeze.plot.Figure = breeze.plot.Figure@3c232051
```

We can define another function, **quadFun**, which allows us to construct these partially applied function closures, and use it as follows:

```
def quadFun(a: Double, b: Double, c: Double):
  Double => Double = x => quadratic(a,b,c,x)
//quadFun: (a: Double, b: Double, c: Double)Double => Double
val myQuad4 = quadFun(2,1,3)
//myQuad4: Double => Double = <function1>
plotFun(myQuad4)
```

```
//res8: breeze.plot.Figure = breeze.plot.Figure@5e9bbd9d
plotFun(quadFun(1,2,3))
//res9: breeze.plot.Figure = breeze.plot.Figure@23e3f5cd
```

Here, **quadFun** is a HOF in the sense that it returns a function closure corresponding to the partially applied **quadratic** function. The returned function has the type **Double =>Double**, so we can use it wherever a function with this signature is expected. Note that the function carries around with it its lexical "environment", specifically, the values of **a**, **b** and **c** specified at the time **quadFun** was called. This style of constructing closures works in most lexically scoped languages which have functions as first class objects. Again, the intention is perhaps slightly more explicit if we re-write the above using function syntax as:

```
val quadFunF: (Double,Double,Double) => (Double =>
      Double) = (a,b,c) => (x => quadratic(a,b,c,x))
//quadFunF: (Double, Double, Double) => Double => Double = <function3>
val myQuad5 = quadFunF(-1,1,2)
//myQuad5: Double => Double = <function1>
plotFun(myQuad5)
//res10: breeze.plot.Figure = breeze.plot.Figure@1f129467
plotFun(quadFunF(1,-2,3))
//res11: breeze.plot.Figure = breeze.plot.Figure@353e6389
```

Now, this concept of partial application is so prevalent in FP that some languages have special syntactic support for it. In Scala, we can partially apply using an underscore to represent unapplied parameters as:

```
val myQuad6 = quadratic(1,2,3,_: Double)
//myQuad6: Double => Double = <function1>
plotFun(myQuad6)
//res12: breeze.plot.Figure = breeze.plot.Figure@238291d4
```

In Scala we can also directly write our functions in *curried* form, with parameters (or parameter lists) ordered as they are to be applied. So, for this example, we could define (partially) curried **quad** and use it with:

```
def quad(a: Double, b: Double, c: Double)(x: Double):
      Double = a*x*x + b*x + c
//quad: (a: Double, b: Double, c: Double)(x: Double)Double
plotFun(quad(1,2,-3))
//res13: breeze.plot.Figure = breeze.plot.Figure@38811103
val myQuad7 = quad(1,0,1) _
//myQuad7: Double => Double = <function1>
val myQuad7 = quad(1,0,1)(_)
//myQuad7: Double => Double = <function1>
val myQuad7: Double => Double = quad(1,0,1)
//myQuad7: Double => Double = <function1>
plotFun(myQuad7)
//res14: breeze.plot.Figure = breeze.plot.Figure@6537ac
```

Note the use of an underscore to convert a partially applied method to a function. Also note that every function has a method **curried** which turns an uncurried function into a (fully) curried function. So in the case of our quadratic function, the fully curried version will be a chain of four functions.

```
def quadCurried = (quadratic _).curried
//quadCurried: Double => (Double => (Double => (Double => Double)))
def quadCurried = (quadratic(_,_,_,_)).curried
//quadCurried: Double => (Double => (Double => (Double => Double)))
plotFun(quadCurried(1)(2)(3))
//res15: breeze.plot.Figure = breeze.plot.Figure@7c4d1c7b
```

Again, note the strategic use of an underscore. The underscore isn't necessary if we have a true function to start with, as the following illustrates:

```
val quadraticF: (Double,Double,Double,Double) => Double =
      (a,b,c,x) => a*x*x + b*x + c
def quadCurried2 = quadraticF.curried
plotFun(quadCurried2(-1)(2)(3))
```

Working with functions, closures, HOFs and partial application is fundamental to effective functional programming style. Currying functions is one approach to handling the function environment problem, and is the standard approach in languages such as Haskell. However, in Scala there are other possible approaches, such as using underscores for partial application. The preferred approach will depend on the context. Currying is often used for HOFs accepting a function as argument (as it can help with type inference), and also in conjunction with *implicits* (which we will consider briefly in the final chapter). In other contexts partial application using underscores seems to be more commonly used.

Function composition

Many programs fundamentally consist of the application of a function to input data to get output data. In other words, the program itself can be considered to be a function. However, this function may be very complicated, and when writing software we tend to break down big complex functions into smaller, simpler functions, and repeat, until we are dealing with small functions of manageable size and complexity. But then we compose together the smaller pieces to get larger programs. It is worth considering carefully the act of building a function from smaller components. Here we consider a trivial example, but the same principles apply for functions of arbitrary size and complexity.

We start out thinking about a function to measure the length of a string.

```
val aLongString = (1 to 10000).map(_.toString).
     reduce(_+_)
//aLongString: String = 12345678910111213141516171819 20...

val stringLength: String => Int = s => s.length
//stringLength: String => Int = <function1>

stringLength(aLongString)
//res0: Int = 38894
```

Clearly this is trivial, using the method **length** on **String**. Now suppose that for large strings, we would rather measure the length in kilo- characters, and that we have a function to convert an **Int** to a **Double** (in K). A very imperative way of composing these functions would be the following.

```
def convertToK: Int => Double = i => i.toDouble/1024
//convertToK: Int => Double

scala>
def stringLengthInK1(s: String): Double = {
  val l = stringLength(s)
  val lk = convertToK(l)
  lk
}
//stringLengthInK1: (s: String)Double

stringLengthInK1(aLongString)
//res1: Double = 37.982421875
```

There isn't anything fundamentally wrong with this, but it's useful to consider more functional approaches. A first attempt at improving this might be the following.

```
val stringLengthInK2: String => Double =
      s => convertToK(stringLength(s))
//stringLengthInK2: String => Double = <function1>

stringLengthInK2(aLongString)
//res3: Double = 37.982421875
```

This captures the fundamental notion of function compostition. But since it is such a fundamental concept, it is helpful to abstract the notion as a HOF, which in Scala is called **compose**. We can modify this slightly using **compose** as follows.

```
val stringLengthInK3: String => Double =
      s => (convertToK compose stringLength)(s)
//stringLengthInK3: String => Double = <function1>

stringLengthInK3(aLongString)
//res4: Double = 37.982421875
```

But this is missing the point of introducing the abstraction. When we look at this carefully, it becomes clear that we don't need the string value s at all, and we could just directly define and use the composition.

```
val stringLengthInK4: String => Double =
      convertToK compose stringLength
//stringLengthInK4: String => Double = <function1>

stringLengthInK4(aLongString)
//res5: Double = 37.982421875
```

This style of programming without values is known as point–free style, and is used extensively in functional programming.

2.3 Note: Scala IO

Java IO (java.io) is utilized for IO in Scala.

```
/* Classes for IO
import java.lang.System.in              // java.io.InputStream
import java.lang.System.{out, err}      // java.io.PrintStream
import java.lang.System.out.print
import java.lang.System.err.println

import java.io.File

import java.io.InputStream
import java.io.InputStream.{close, read}

import java.io.OutputStream
import java.io.OutputStream.{write, flush, close}
import java.io.PrintStream
import java.io.PrintStream.{print, println, flush, close}

import java.io.FileInputStream
import java.io.FileInputStream.{close, read}
import java.io.FileOutputStream
import java.io.FileOutputStream.{close, flush, write}

import scala.sys.process.stdin // == java.lang.System.in
import scala.sys.process.stdout // == java.lang.System.out
import scala.sys.process.stderr // == java.lang.System.err
import scala.io.StdIn         //Interactive IO on console
import scala.io.Source        //An iterable representation of source data
*/

val filename = "/tmp/output.txt"

val outfile = new java.io.File(filename)
//outfile: java.io.File = /tmp/output.txt
val fout = new java.io.FileOutputStream(outfile)
//fout: java.io.FileOutputStream = java.io.FileOutputStream@7582ff54
val pout = new java.io.PrintStream(fout)
//pout: java.io.PrintStream = java.io.PrintStream@5bf0fe62
pout.print("%d Output to PrintStream by the print method\n".format(1))
//
val line = 2
//line: Int = 2
pout.print(s"${line}nd line\n")
//
pout.close
```

```
//
val sysout = scala.sys.process.stdout // val sysout = java.lang.System.out
//sysout: java.io.PrintStream = java.io.PrintStream@224aed64
val fin = new java.io.FileInputStream(filename)
//fin: java.io.FileInputStream = java.io.FileInputStream@63a5e46c
val bufsource = scala.io.Source.fromInputStream(fin)
//bufsource: scala.io.BufferedSource = empty iterator
val (lines, linesDup) = bufsource.getLines.duplicate
//lines: Iterator[String] = empty iterator
lines.foreach{ sysout.println(_) } // not including NL.
//1 Output to PrintStream by the print method
//2nd line
lines.foreach{ sysout.println(_) }      // Iterator ends.
//
val textLines = linesDup.mkString("\n")   // Iterator works.

val bufsource2 = scala.io.Source.fromInputStream(fin)
//bufsource: scala.io.BufferedSource = empty iterator
val lines2 = bufsource2.getLines
//lines: Iterator[String] = empty iterator
lines2.foreach{ sysout.println(_) } // There is no data in InputStream.
//
bufsource.close

// Convert text lines to inputStream
val stream: java.io.InputStream = new java.io.ByteArrayInputStream(
textLines.getBytes(java.nio.charset.StandardCharsets.UTF_8.name))
```

Chapter 3

The Scala collections library

3.1 Scala collections overview and basics

3.1.1 Introduction

The Scala Standard Library provides a rich set of data structures for dealing with "collections" of objects. We have already encountered one of these: the **List** collection. Note that **List** itself is not a concrete type. It is a *parametrised* type (called a *generic* type in Java), and more commonly referred to as a *type constructor* in Scala. It is a type constructor in the sense that if you provide a type parameter it will return a concrete type. So, for example, **List** and **List[_]** are type constructors, but **List[Int]** and **List[Double]** are concrete types. The List collection that we have previously encountered is *immutable*. That means that it is not possible to change any of the elements in a **List**. But the Scala collections library also contains a *mutable* list, **MutableList**. Let's see how this works.

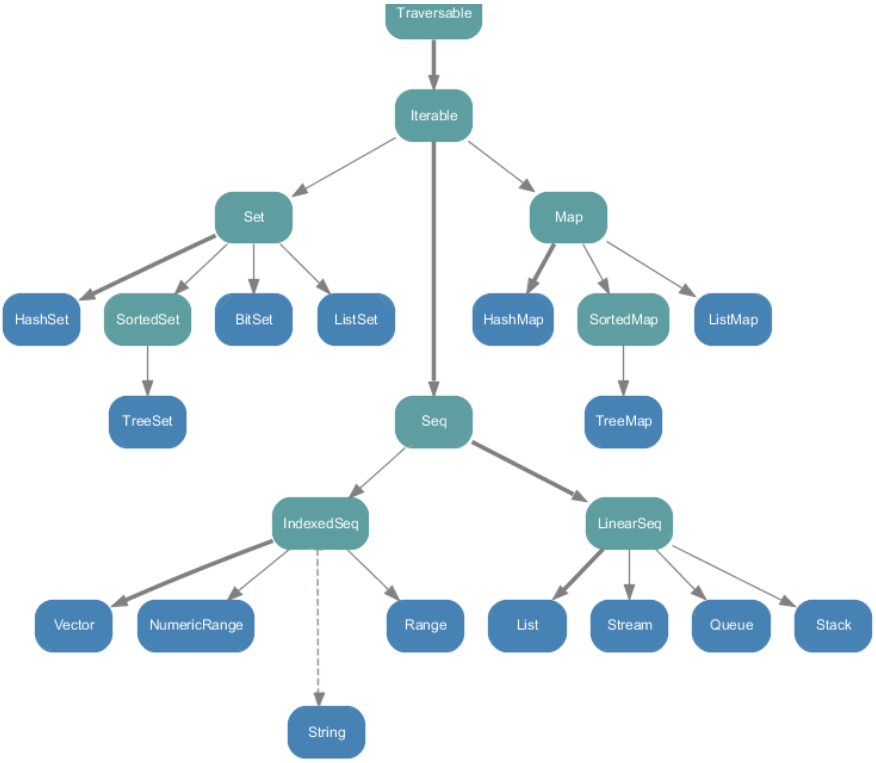
```
val l1 = List(6,7,8,9,10)
//l1: List[Int] = List(6, 7, 8, 9, 10)
l1.head
//res6: Int = 6
l1.tail
//res7: List[Int] = List(7, 8, 9, 10)
l1(0)
//res8: Int = 6
l1(2)
//res9: Int = 8
l1(2) = 22
//<console>:13: error: value update is not a member of List[Int]
//       l1(2) = 22
//       ^
l1
//res10: List[Int] = List(6, 7, 8, 9, 10)
val l2: List[Double] = List(1,2,3)
//l2: List[Double] = List(1.0, 2.0, 3.0)

import scala.collection.mutable
//import scala.collection.mutable
```

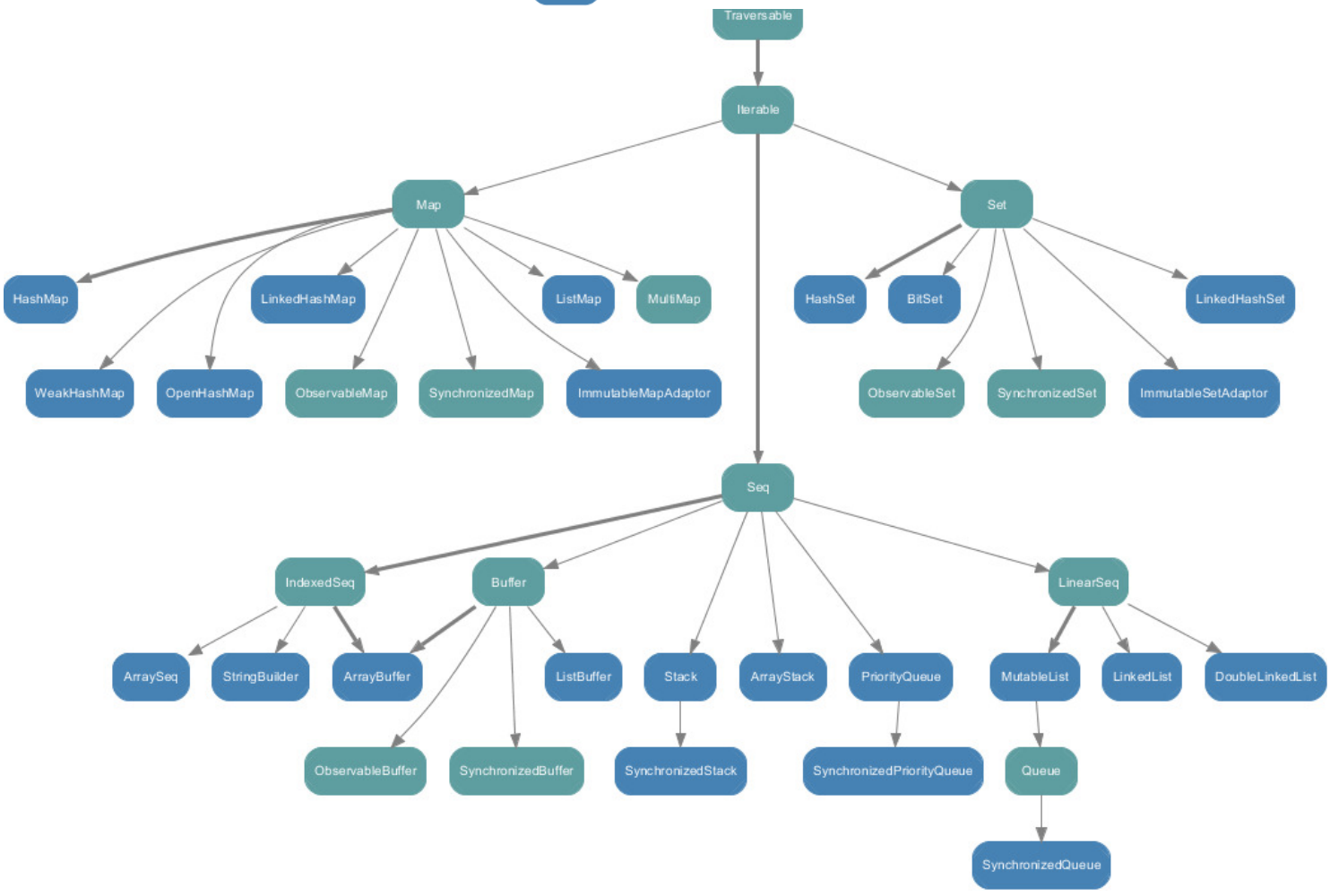
```
val l3 = mutable.MutableList(5,6,7,8,9)
//l3: scala.collection.mutable.MutableList[Int] = MutableList(5, 6, 7, 8, 9)
l3.head
//res12: Int = 5
l3.tail
//res13: scala.collection.mutable.MutableList[Int] = MutableList(6, 7, 8, 9)
l3(2)
//res14: Int = 7
l3(2) = 22
//
l3
//res16: scala.collection.mutable.MutableList[Int] = MutableList(5, 6, 22, 8, 9)
val l4 = l3
//l4: scala.collection.mutable.MutableList[Int] = MutableList(5, 6, 22, 8, 9)
val l5 = l3 ++ l4
//l5: scala.collection.mutable.MutableList[Int] = MutableList(5, 6, 22, 8, 9, 5, 6,
l4(0) = 0
//
l3(3) = 1
//
l3
//res17: scala.collection.mutable.MutableList[Int] = MutableList(0, 6, 22, 1, 9)
l4
//res18: scala.collection.mutable.MutableList[Int] = MutableList(0, 6, 22, 1, 9)
l5
//res19: scala.collection.mutable.MutableList[Int] = MutableList(5, 6, 22, 8, 9, 5,
```

Note that l3 is an immutable val even though it holds a reference to a **MutableList** containing elements which can be changed. FP advocates the use of immutable references to immutable collections, but when mutability is required/desired, you will usually want *either* an immutable reference to a mutable collection (as above), *or* a mutable (**var**) reference to an immutable collection (so that the collection itself is immutable, but the collection referenced by the variable can be changed). Which of these two possibilities is most desirable will be problem-dependent. However, it is very rare to really require a mutable reference to a mutable collection – this is usually indicative of confusion.

3.1.2 The collections hierarchy

List is by no means the only collection provided by the Scala Standard Library. There are many collections, abstract and concrete, arranged in a hierarchy in an Object-Oriented style. So, if we start with concrete collections like **List** and **Stream**, these are both a **LinearSeq**, and a **LinearSeq** is a **Seq**, and a **Seq** is an **Iterable** and an **Iterable** is a **Traversable**. On the other hand, **Vector** and **Range** are both examples of **IndexedSeq**, which is itself an example of **Seq**. So the common superclass of **Vector** and **List** is Seq. Unfortunately we don't have time to go through the full collections hierarchy here in detail, but the official documentation available on-line is good. See the Collections Overview in the Scala Documentation for further details; see `https://docs.scala-lang.org/overviews/collections/overview.html`.


Traversable
Iterable
Set
Map
HashSet
SortedSet
BitSet
ListSet
HashMap
SortedMap
ListMap
TreeSet
Seq
TreeMap
IndexedSeq
LinearSeq
Vector
NumericRange
Range
List
Stream
Queue
Stack
String


Traversable
Iterable
Map
Set
HashMap
LinkedHashMap
ListMap
MultiMap
HashSet
BitSet
LinkedHashSet
WeakHashMap
OpenHashMap
ObservableMap
SynchronizedMap
ImmutableMapAdaptor
ObservableSet
SynchronizedSet
ImmutableSetAdaptor
Seq
IndexedSeq
Buffer
LinearSeq
ArraySeq
StringBuilder
ArrayBuffer
ListBuffer
Stack
ArrayStack
PriorityQueue
MutableList
LinkedList
DoubleLinkedList
ObservableBuffer
SynchronizedBuffer
SynchronizedStack
SynchronizedPriorityQueue
Queue
SynchronizedQueue

Here we will just focus on a few of the most useful Scala collections from the perspective of statistical computing: **List**, **Array**, **Vector**, **Map**, **Stream** and **Range**. As we will see, most functionality is common to all Scala collections.

List

We have already seen **List** – it is an immutable linked list. It has very fast prepend (::) and **head** and **tail** operations. But it has relatively slow indexed access and append operations, so it should be avoided if these are the main focus – **Array** or **Vector** will be better in this case. Here are some more examples of using **List**.

```
val list1 = List(3,4,5,6)
//list1: List[Int] = List(3, 4, 5, 6)
val list2 = 2 :: list1
//list2: List[Int] = List(2, 3, 4, 5, 6)
val list3 = list1 ++ list2
//list3: List[Int] = List(3, 4, 5, 6, 2, 3, 4, 5, 6)
list3.take(3)
//res17: List[Int] = List(3, 4, 5)
list3.drop(2)
//res18: List[Int] = List(5, 6, 2, 3, 4, 5, 6)
list3.filter(_<5)
//res19: List[Int] = List(3, 4, 2, 3, 4)
val list4 = list1 map (_*2)
//list4: List[Int] = List(6, 8, 10, 12)
list4.length
//res20: Int = 4
list4.sum
//res21: Int = 36
list4.reverse
//res22: List[Int] = List(12, 10, 8, 6)
list1.foldLeft(0)(_+_)
//res23: Int = 18
list1.scanLeft(0)(_+_)
//res24: List[Int] = List(0, 3, 7, 12, 18)
list1.reduce(_+_)
//res25: Int = 18
list1.sortWith(_>_)
//res26: List[Int] = List(6, 5, 4, 3)
list1.foreach{e => println(e)}
//3
//4
//5
//6
```

Array

Array is a mutable data structure providing a thin layer over Java arrays. Consequently, **Array** has very fast indexed access and in-place update. When used with primitive numeric types such as **Int** and **Double**, they should be more-or-less as fast as numeric arrays in natively compiled C code. **Array**s tend to be used whenever fast in-place mutation is required, or inter-op with Java arrays.

```
val arr1 = Array(2,3,4,5)
//arr1: Array[Int] = Array(2, 3, 4, 5)
arr1(1)=6
//
arr1
//res1: Array[Int] = Array(2, 6, 4, 5)
val arr2 = arr1 map(_+1)
//arr2: Array[Int] = Array(3, 7, 5, 6)
arr1
//res2: Array[Int] = Array(2, 6, 4, 5)
arr1.reduce(_+_)
//res3: Int = 17
arr1 ++ arr2
//res4: Array[Int] = Array(2, 6, 4, 5, 3, 7, 5, 6)
val arr3 = arr2
//arr3: Array[Int] = Array(3, 7, 5, 6)
arr2(0)= 1
//
arr3(1) = 0
//
arr2
//res5: Array[Int] = Array(1, 0, 5, 6)
arr3
//res6: Array[Int] = Array(1, 0, 5, 6)
```

Vector

Vector is an immutable alternative to **Array** which tends to be used a lot in Scala code. It has fast indexed access, prepend and append, and a fast immutable update, which returns a new **Vector**, leaving the old **Vector** unchanged, but without copying (most of) the original **Vector**. **Vector** has many applications in statistical computing, and as we will see shortly, also has a parallel implementation.

```
val vec1 = Vector(6,5,4,3)
//vec1: scala.collection.immutable.Vector[Int] = Vector(6, 5, 4, 3)
vec1(2)
//res0: Int = 4
vec1.toList
//res1: List[Int] = List(6, 5, 4, 3)
vec1 map (_*0.5)
//res2: scala.collection.immutable.Vector[Double] = Vector(3.0, 2.5, 2.0, 1.5)
val vec2 = vec1.updated(2,10)
```

```
//vec2: scala.collection.immutable.Vector[Int] = Vector(6, 5, 10, 3)
vec2.length
//res3: Int = 4
vec1
//res4: scala.collection.immutable.Vector[Int] = Vector(6, 5, 4, 3)
vec2
//res5: scala.collection.immutable.Vector[Int] = Vector(6, 5, 10, 3)
vec1 ++ vec2
//res6: scala.collection.immutable.Vector[Int] = Vector(6, 5, 4, 3, 6, 5, 10, 3)
vec1 :+ 22
//res7: scala.collection.immutable.Vector[Int] = Vector(6, 5, 4, 3, 22)
33 +: vec1
//res8: scala.collection.immutable.Vector[Int] = Vector(33, 6, 5, 4, 3)
vec1.reduce(_+_)
//res9: Int = 18
Vector.fill(5)(0)
//res10: scala.collection.immutable.Vector[Int] = Vector(0, 0, 0, 0, 0)
Vector.fill(5)(math.random)
//res11: scala.collection.immutable.Vector[Double] = Vector(0.9780496017500556,
//         0.5184170214103406, 0.8486192282618858, 0.17440966048867612, 0.9053754317
```

Note that the final example relies on the fact that the second argument to .fill is lazily evaluated.

Map

Map is an immutable hash-map implementation, used for storing "keys" and "values". These have many potential applications requiring fast lookup operations.

```
val map1 = Map("a" -> 10, "b" -> 20, "c" -> 30)
//map1: scala.collection.immutable.Map[String,Int] = Map(a -> 10, b -> 20, c -> 30)
map1("b")
//res0: Int = 20
val map2 = map1.updated("d",40)
//map2: scala.collection.immutable.Map[String,Int] = Map(a -> 10, b -> 20, c -> 30,
map1
//res1: scala.collection.immutable.Map[String,Int] = Map(a -> 10, b -> 20, c -> 30)
map2
//res2: scala.collection.immutable.Map[String,Int] = Map(a -> 10, b -> 20, c -> 30,
map2.updated("d", 50)
//res3: scala.collection.immutable.Map[String,Int] = Map(a -> 10, b -> 20, c -> 30,
map2.keys
//res4: Iterable[String] = Set(a, b, c, d)
map2.values
//res5: Iterable[Int] = MapLike(10, 20, 30, 40)
map2 mapValues (_*2)
//res6: scala.collection.immutable.Map[String,Int] = Map(a -> 20, b -> 40, c -> 60,
val mapK = List(3,4,5,6)
```

```
//mapK: List[Int] = List(3, 4, 5, 6)
val mapV = List(0.3,0.4,0.5,0.6)
//mapV: List[Double] = List(0.3, 0.4, 0.5, 0.6)
val pairs = mapK zip mapV
//pairs: List[(Int, Double)] = List((3,0.3), (4,0.4), (5,0.5), (6,0.6))
val map3 = pairs.toMap
//map3: scala.collection.immutable.Map[Int,Double] = Map(3 -> 0.3, 4 -> 0.4, 5 -> 0.
```

Stream

A **Stream** is a *lazy* immutable **List**. It is lazy in the sense that just like a regular list, it has a **head** and a **tail**, but for a **Stream** only the **head** is eagerly evaluated. Evaluation of the tail is evaluated only when required. **Stream**s are therefore useful for dealing with data of undetermined or infinite size. In particular, **Stream**s can be used to define and operate on **List**s of infinite length, as the examples below demonstrate. This is a very powerful concept, and there are many potential applications of **Stream**s in statistical computing — they provide an elegant framework for any kind of iterative algorithm. However, great care must be exercised when working with infinite **Stream**s, since it is very easy to apply an operation requiring an infinite amount of computation.

```
val stream1 = Stream(1,2,3)
//stream1: scala.collection.immutable.Stream[Int] = Stream(1, ?)
val stream2 = 0 #:: stream1
//stream2: scala.collection.immutable.Stream[Int] = Stream(0, ?)
stream2.toList
//res0: List[Int] = List(0, 1, 2, 3)
stream2.foldLeft(0)(_+_)
//res1: Int = 6
stream2
//res2: scala.collection.immutable.Stream[Int] = Stream(0, 1, 2, 3)
def fibFrom(a: Int,b: Int): Stream[Int] = a #:: fibFrom(b,a+b)
//fibFrom: (a: Int, b: Int)Stream[Int]
val fib = fibFrom(1,1)
//fib: Stream[Int] = Stream(1, ?)
fib.take(8).toList
//res3: List[Int] = List(1, 1, 2, 3, 5, 8, 13, 21)
val naturals = Stream.iterate(1)(_+1)
//naturals: scala.collection.immutable.Stream[Int] = Stream(1, ?)
naturals.take(5).toList
//res4: List[Int] = List(1, 2, 3, 4, 5)
val evens = naturals map (_*2)
//evens: scala.collection.immutable.Stream[Int] = Stream(2, ?)
evens.take(6).toList
//res5: List[Int] = List(2, 4, 6, 8, 10, 12)
val triangular = naturals.scanLeft(0)(_+_).drop(1)
//triangular: scala.collection.immutable.Stream[Int] = Stream(1, ?)
triangular.take(8).toList
//res6: List[Int] = List(1, 3, 6, 10, 15, 21, 28, 36)
```

Range

Range is a very simple collection used to represent a simple linear numerical sequence. It has constant size since it can be represented by three numbers (start, end and step size). It is intended that a **Range** is constructed using one of the convenience methods **to** or **until** (**until** excludes the final value).

```
1 to 5
//res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
0 to 5
//res1: scala.collection.immutable.Range.Inclusive = Range(0, 1, 2, 3, 4, 5)
0 until 5
//res2: scala.collection.immutable.Range = Range(0, 1, 2, 3, 4)
0 to 10 by 2
//res3: scala.collection.immutable.Range = Range(0, 2, 4, 6, 8, 10)
val range = 0 to 10 by 2
//range: scala.collection.immutable.Range = Range(0, 2, 4, 6, 8, 10)
range.toList
//res4: List[Int] = List(0, 2, 4, 6, 8, 10)
(0 to 5) map (_*2)
//res5: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 2, 4, 6, 8, 10)
(0 to 5).sum
//res6: Int = 15
```

3.1.3 Writing collection–generic code

Suppose we want to write our own function to compute the mean of a collection of **Double**s. We could write a function for **List[Double]** as follows.

```
def meanList(ld: List[Double]):Double=ld.sum/ld.length
//meanList: (ld: List[Double])Double
meanList(List(1,2,3,4))
//res0: Double = 2.5
```

This works fine for a **List**, but essentially the same code could be used for a **Vector**, or an **Array**. Rather than writing a different version of the function for every collection type we are interested in, we can instead write it once for the common supertype.

```
def mean(sq: Seq[Double]): Double = sq.sum/sq.length
//mean: (sq: Seq[Double])Double
mean(List(2,3,4))
//res1: Double = 3.0
mean(Vector(1,2,3))
//res2: Double = 2.0
mean(Array(1,2,3))
//res3: Double = 2.0
```

This will work for any collection derived from **Seq** in the collections hierarchy.

3.1.4 Writing type–generic code

A similar issue arises when we want to write code that abstracts over the type parametrising the collection. Suppose we want to write a function which replicates a **List**. We can write such a function for **List[Double]** as follows.

```
def repD(ld: List[Double], n: Int): List[Double] =
  if (n <= 1) ld else ld ++ repD(ld,n-1)
//repD: (ld: List[Double], n: Int)List[Double]
repD(List(1,2,3),3)
//res0: List[Double] = List(1.0, 2.0, 3.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0)
```

This works fine, but clearly we did not rely on the fact that the type parameter was **Double**. The same code should work for **List[Int]** or even **List[String]**. We can write the function for a **List[T]** where **T** is any type as follows.

```
def repl[T](l: List[T], n: Int): List[T] =
  if (n <=1) l else l ++ repl(l,n-1)
//repl: [T](l: List[T], n: Int)List[T]
repl(List(1,2,3),3)
//res1: List[Int] = List(1, 2, 3, 1, 2, 3, 1, 2, 3)
 repl(List(1.0,2.0),3)
//res2: List[Double] = List(1.0, 2.0, 1.0, 2.0, 1.0, 2.0)
repl(List("a", "b", "c"), 2)
//res3: List[String] = List(a, b, c, a, b, c)
repl[Double](List(1,2,3),3)
//res4: List[Double] = List(1.0, 2.0, 3.0, 1.0, 2.0, 3.0, 1.0, 2.0, 3.0)
```

3.1.5 Writing (type– and) collection–generic code returning a collection

The method above replicates a **List**, but we haven't used anything **List**–specific, so it's natural to want to generalise this code to work for any collection. We kind of know how to do this already, by replacing **List** with a desired super–type, say **Seq** (or **Traversable**, or **Iterable**, ...).

```
def reps[T](l: Seq[T], n: Int): Seq[T] =
  if (n <=1) l else l ++ reps(l,n-1)
//reps: [T](l: Seq[T], n: Int)Seq[T]
reps(List(1,2,3),3)
//res0: Seq[Int] = List(1, 2, 3, 1, 2, 3, 1, 2, 3)
reps(Vector(1.0,2.0),3)
//res1: Seq[Double] = Vector(1.0, 2.0, 1.0, 2.0, 1.0, 2.0)
reps(Array("a","b","c"),2)
//res2: Seq[String] = ArrayBuffer(a, b, c, a, b, c)
```

This works fine, but may not be exactly what you want. Look carefully at the return types. When you replicate a (say) **List[Int]**, you do actually get back a **List[Int]**, but its type is (obviously) **Seq[Int]**. This might be OK, but this means that at compile time all you can assume about the return collection is that it is a **Seq**, which would mean that you couldn't call any **List**–specific methods on the result. This is likely to be problematic in many situations. What is required is a function that works for any **Seq**, but returns a

collection of the same type as was passed in. It is perfectly possible to do this, but it requires a bit of "type magic" that is beyond the scope of this course to explain properly.

```
import scala.collection.SeqLike
//import scala.collection.SeqLike
def rep[T,C<:Seq[T]](l:C with SeqLike[T,C],n:Int):C=
  if (n <=1) l else (l ++ rep(l,n-1)).asInstanceOf[C]
//rep: [T, C <: Seq[T]](l: C with scala.collection.SeqLike[T,C], n: Int)C
rep(List(1,2,3),3)
//res0: List[Int] = List(1, 2, 3, 1, 2, 3, 1, 2, 3)
rep(Vector(1.0,2.0),3)
//res1: scala.collection.immutable.Vector[Double] = Vector(1.0, 2.0, 1.0, 2.0, 1.0,
rep(Array("a","b","c").seq,2)
//res2: scala.collection.mutable.IndexedSeq[String] = ArrayBuffer(a, b, c, a, b, c)
```

This works, but is a bit ugly. Many think that the ugliness of this kind of code is a symptom of the fundamental inadequacy of traditional inheritance– based object–oriented programming paradigms for providing polymorphism. It is possible to develop a more elegant approach to polymorphism and generic code using *type classes*, which we will consider briefly in the final Chapter.

Note now that if you pass in a **List[Int]** the return type is also **List[Int]**, and so you could still use **List**–specific methods on the return value.

3.1.6 Writing type-generic numerical code

The above approach to abstracting over the type parameter works fine so long as you don't need to know anything about the abstract type. But in numerical code, you often want to use numerical operations within generic functions. For example, suppose we want a function to compute the sum-of-squares of a collection of numerical values. We can write such a function for **Seq[Double]** easily.

```
def ssd(sq: Seq[Double]): Double = (sq map (x => x*x)).sum
//ssd: (sq: Seq[Double])Double
sd(List(2,3,4))
//res0: Double = 29.0
```

Now this will work for other numeric types, but will always return a **Double**. This isn't necessarily what we want. For example, if we pass in a list of **Integer**s, we might want to get the sum–of–squares back as an **Integer**. But if we naively abstract over the numeric type we have a problem.

```
def ssg[T](sq: Seq[T]): T = (sq map (x => x*x)).sum
//<console>:11: error: value * is not a member of type parameter T
//       def ssg[T](sq: Seq[T]): T = (sq map (x => x*x)).sum
//                                                   ^
```

The error message helpfully explains that since we know nothing about T, there is no reason to suppose that it will have a multiplication operation defined. We need to tell the compiler that we expect the type **T** to be a numeric type, for which basic numerical operations are defined. Unfortunately in Scala this is more difficult than it should be, due to the need to keep compatibility with the underlying Java numeric types (for performance reasons). We can do it, but using some advanced features of the language that we are not yet ready to discuss in detail.

```
def ss[T](sq: Seq[T])(implicit num: Numeric[T]): T = {
  import num._
  (sq map (x => x*x)).sum
}
//ss: [T](sq: Seq[T])(implicit num: Numeric[T])T
ss(List(2,3,4))
//res1: Int = 29
ss(Vector(1.0,2.0,3.0))
//res2: Double = 14.0
```

This works, but is somewhat mysterious and a bit ugly. We can write essentially the same code slightly differently as follows.

```
def ssg[T: Numeric](sq: Seq[T]): T = {
     val num = implicitly[Numeric[T]]
  import num._
  (sq map (x => x*x)).sum
}
//ssg: [T](sq: Seq[T])(implicit evidence$1: Numeric[T])T
sg(List(2,3,4))
//res3: Int = 29
```

It is still a little ugly. **Spire** [5)] is a nice Scala library for numerical program- https://github.com/non/spire ming in Scala. We don't have time to explore it in detail in this course, but using it, we can solve this problem in a slightly neater way.

```
import spire.math._
import spire.implicits._
def ss[T: Numeric](sq: Seq[T]): T =
  (sq map (x=>x*x)).reduce(_+_)
//
ss(List(1,2,3))
/res4: Int = 14
ss(Vector(2.0,3.0,4.0))
//res5: Double = 29.0
```

By bringing in Spire implicits associated with the **Numeric** typeclass and declaring that **T** belongs to the numeric typeclass, we ensure that the multiplication operation ***** is available for objects of type T.

> Note that *Breeze* (which we will be discussing more later) has a dependency on Spire, so if you have a REPL with a dependency on Breeze, you do not need to add an additional dependency on Spire in order to use it.

3.2 Parallel collections

3.2.1 Introduction

In addition to the regular (serial) collections built in to the Scala standard library, there are also parallel versions of many of the collection classes. For example, the parallel version of **Vector** is **ParVector** and the

parallel version of **Array** is **ParArray**. A serial collection can be converted to a parallel version by calling the method **.par**.

```
(1 to 12).par
//res0: scala.collection.parallel.immutable.ParRange = ParRange(1, 2, 3, 4, 5, 6, 7,
Vector(2,4,6,8).par
//res1: scala.collection.parallel.immutable.ParVector[Int] = ParVector(2, 4, 6, 8)
Array("a", "b", "c").par
//res2: scala.collection.parallel.mutable.ParArray[String] = ParArray(a, b, c)
List(1,2,3,4).par
//res3: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(1, 2, 3, 4)
```

Note in the final example above, calling **.par** on a **List** returns a **ParVector**. There is no **ParList**, since a linked list is a fundamentally serial data structure. It should be clear that mapping a pure function over a collection can be parallelised easily, since the mapping of each element can happen independently of the mapping of any other element. Thus, a mapping operation can be split among available threads or processors to give a speed-up. The precise speed-up obtained in practice will depend on many factors, such as the size of the collection and the computational complexity of the function to be mapped. But in the ideal case, it should be possible to get close to an N times speed-up by using (N) cores or processors, and in the theoretical limit of infinite parallel hardware, a **map** on a collection of size n should have its time-complexity reduced from $O(n)$ to $O(1)$. **filter** operations parallelise similarly.

Reduction operations can also be parallelised, provided that the reduction function is **associative**, using a technique called **tree reduction**. The details are not important here, but in the theoretical best case limit on infinite parallel hardware it should reduce the time-complexity of a fold or reduce operation from $O(n)$ to $O(logn)$.

```
def isPrime(n: Int): Boolean = (2 until n) forall (n % _ != 0)
//isPrime: (n: Int)Boolean
(2 to 20) filter isPrime
//res0: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 3, 5, 7, 11, 13, 17,
((2 to 100000) filter isPrime).length
//res1: Int = 9592
((2 to 100000).par filter isPrime).length
//res2: Int = 9592
(100000000 to 100000100) filter isPrime
//res4: scala.collection.immutable.IndexedSeq[Int] = Vector(100000007, 100000037, 10
(100000000 to 100000100).par filter isPrime
//res5: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(100000007, 10000
```

By entering these commands into a Scala REPL, it should be clear that filtering a large collection executes significantly faster when using a parallel collection. The final example should also illustrate that the collection doesn't have to be large to benefit from parallelisation, so long as the function used to map or filter is expensive. Again, if you want to work with numbers bigger than **Int.MaxValue = 2,147,483,647** (roughly 2 billion) you can switch to **BigInt**s.

Scala auto-detects the number of cores available on a system and sets the level of parallelism used accordingly. You can easily find out how many threads it will use.

```
List().par.tasksupport.parallelismLevel
//res6: Int = 4
```

It is possible to change the level of parallelism used on a par-collection basis (see the on-line docs for details).

3.2.2 Writing parallelism-agnostic code

We previously considered the problem of writing collection-generic code. That is, code which will work when passed in any sequential collection, irrespective of whether it is an **Array**, **List**, **Vector**, etc. The trick was to parametrise the method signature using the **Seq** collection supertype. But this doesn't work for parallel collections as the following generic sum–of–squares method illustrates.

```
def sss(sq: Seq[Int]): Int = (sq map (x => x*x)).reduce(_+_)
//sss: (sq: Seq[Int])Int
sss(List(2,3,4))
//res0: Int = 29
sss(Vector(2,3,4))
//res1: Int = 29
sss(List(2,3,4).par)
//<console>:13: error: type mismatch;
// found   : scala.collection.parallel.immutable.ParSeq[Int]
// required: Seq[Int]
//       sss(List(2,3,4).par)
//                       ^
```

The problem is that **Seq** is not a supertype of the parallel collections. **Seq** is a supertype of the *serial* sequential collections. **ParSeq** is the supertype of the *parallel* sequential collections. So if we want to write code that is agnostic to the kind of parallel collection used, we can used **ParSeq**. However, for code that is agnostic to whether a serial or parallel collection is used, we need to know that there is a class **GenSeq** that is a parent of
TEXTBFSeq and **ParSeq**. So if we want to write code that is parallelism-agnostic, **GenSeq** is often what we want. This requires an import.

```
import scala.collection.GenSeq
//import scala.collection.GenSeq
def ssp(sq: GenSeq[Int]): Int = (sq map (x => x*x)).reduce(_+_)
//ssp: (sq: scala.collection.GenSeq[Int])Int
ssp(List(2,3,4))
//(2,3,4))
ssp(List(2,3,4).par)
//res4: Int = 29
ssp(Vector(2,3,4).par)
//res5: Int = 29
ssp(Array(2,3,4).par)
//res6: Int = 29
```

This approach works fine so long as we don't need to return a collection. If we want to return a collection of the same type as passed in, we need to use the same sub-typing tricks that we used with **Seq** earlier, or typeclasses, which we will consider in the final chapter.

3.3 Collections as monads

3.3.1 First steps with monads in Scala

Before leaving the topic of collections, it is worth pointing out that Scala collections turn out be a special case of the more general functional programming notion of a *monad*. This is why Scala collections are often referred to as *monadic* collections, and why collection classes are often referred to as monads, such as the textitList monad. Consequently, Scala collections provide quite an intuitive introduction to the very powerful notion of a monad. Monads are one of those concepts that turns out to be very simple and intuitive once you "get it", but completely impenetrable until you do! The term "monad" is borrowed from that of the corresponding concept in category theory. The connection between functional programming and category theory is strong and deep. We will explore this a little more later, but for now the connection is not important and no knowledge of category theory is assumed.

> Visual Scala reference [6)] may be helpful to understand the operations of various methods for Scala collections.

Functors and Monads

Maps and Functors

The **map** method is one of the first concepts one meets when beginning functional programming. It is a higher order method on many (immutable) *collection* and other *container* types. Let's start by reminding ourselves of how map operates on Lists.

```
val x = (0 to 4).toList
//x: List[Int] = List(0, 1, 2, 3, 4)
val x2 = x map { x => x * 3 }
//x2: List[Int] = List(0, 3, 6, 9, 12)
val x3 = x map { _ * 3 }
//x3: List[Int] = List(0, 3, 6, 9, 12)
val x4 = x map { _ * 0.1 }
//x4: List[Double] = List(0.0, 0.1, 0.2, 0.30000000000000004, 0.4)
```

The last example shows that a **List[T]** can be converted to a **List[S]** if map is passed a function of type **T =>S**. Of course there's nothing particularly special about **List** here. It works with other collection types in the same way, as the following example with (immutable) **Vector** illustrates:

```
val xv = x.toVector
//xv: Vector[Int] = Vector(0, 1, 2, 3, 4)
val xv2 = xv map { _ * 0.2 }
//xv2: scala.collection.immutable.Vector[Double] = Vector(0.0, 0.2, 0.4, 0.600000000
val xv3 = for (xi <- xv) yield (xi * 0.2)
//xv3: scala.collection.immutable.Vector[Double] = Vector(0.0, 0.2, 0.4, 0.600000000
```

Note here that the for comprehension generating **xv3** is exactly equivalent to the **map** call generating **xv2** – the for–comprehension is just *syntactic sugar* for the **map** call. The benefit of this syntax will become apparent in the more complex examples we consider later. Many collection and other container types have a **map** method that behaves this way. Any parametrised type that does have a **map** method like this is known

as a Functor. Again, the name is due to category theory. From a Scala-programmer perspective, a functor can be thought of as a trait, in pseudo-code as

```
trait F[T] {
  def map(f: T => S): F[S]
}
```

with **F** representing the functor. In fact it turns out to be better to think of a functor as a type class, but as previously mentioned, that is a concept we will return to later. Also note that to be a functor in the strict sense (from a category theory perspective), the map method must behave sensibly – that is, it must satisfy the functor laws. But again, I'm keeping things informal and intuitive for now.

FlatMaps and Monads

Once we can map functions over elements of containers, we soon start mapping functions which themselves return values of the container type. eg. we can map a function returning a **List** over the elements of a **List**, as illustrated below.

```
val x5 = x map { x => List(x - 0.1, x + 0.1) }
//x5: List[List[Double]] = List(List(-0.1, 0.1), List(0.9, 1.1), List(1.9, 2.1), Lis
```

Clearly this returns a list-of-lists. Sometimes this is what we want, but very often we actually want to *flatten* down to a single list so that, for example, we can subsequently map over all of the elements of the base type with a single map. We could take the list-of-lists and then flatten it, but this pattern is so common that the act of mapping and then flattening is often considered to be a basic operation, usually known in Scala as **flatMap**. So for our toy example, we could carry out the **flatMap** as follows.

```
val x6 = x flatMap { x => List(x - 0.1, x + 0.1) }
//x6: List[Double] = List(-0.1, 0.1, 0.9, 1.1, 1.9, 2.1, 2.9, 3.1, 3.9, 4.1)
```

The ubiquity of this pattern becomes more apparent when we start thinking about iterating over multiple collections. For example, suppose now that we have two lists, **x** and **y**, and that we want to iterate over all pairs of elements consisting of one element from each list.

```
val y = (0 to 12 by 2).toList
//y: List[Int] = List(0, 2, 4, 6, 8, 10, 12)
val xy = x flatMap { xi => y map { yi => xi * yi } }
//xy: List[Int] = List(0, 0, 0, 0, 0, 0, 0, 0, 2, 4, 6, 8, 10, 12, 0, 4, 8, 12, 16,
```

This pattern of having one or more nested **flatMap**s followed by a final **map** in order to iterate over multiple collections is very common. It is exactly this pattern that the for–comprehension is syntactic sugar for. So we can re–write the above using a for–comprehension

```
val xy2 = for {
  xi <- x
  yi <- y
} yield (xi * yi)
//xy2: List[Int] = List(0, 0, 0, 0, 0, 0, 0, 0, 2, 4, 6, 8, 10, 12, 0, 4, 8, 12, 16,
val xy3 = for ( xi <- x; yi <- y) yield (xi * yi)
//xy3: List[Int] = List(0, 0, 0, 0, 0, 0, 0, 0, 2, 4, 6, 8, 10, 12, 0, 4, 8, 12, 16,
```

This for–comprehension (usually called a for–expression in Scala) has an intuitive syntax reminiscent of the kind of thing one might write in an imperative language. But it is important to remember that <- is not actually an imperative assignment. The for–comprehension really does expand to the pure-functional nested **flatMap** and TEXTBFmap call given above. Recalling that a functor is a parametrised type with a **map** method, we can now say that a *monad* is just a functor which also has a **flatMap** method. We can write this in pseudo-code as

```
trait M[T] {
  def map(f: T => S): M[S]
  def flatMap(f: T => M[S]): M[S]
}
```

Not all functors can have a flattening operation, so not all functors are monads, but all monads are functors. Monads are therefore more powerful than functors. Of course, more power is not always good. The *principle of least power* is one of the main principles of functional programming, but monads are useful for sequencing dependent computations, as illustrated by for–comprehensions. In fact, since for–comprehensions de–sugar to calls to **map** and **flatMap**, monads are precisely what are required in order to be usable in for–comprehensions. Collections supporting **map** and **flatMap** are referred to as *monadic*. Most Scala collections are monadic, and operating on them using **map** and **flatMap** operations, or using for–comprehensions is referred to as *monadic–style*. People will often refer to the monadic nature of a collection (or other container) using the word monad, eg. the "List monad". So far the functors and monads we have been working with have been collections, but not all monads are collections, and in fact collections are in some ways atypical examples of monads. Many monads are *containers* or *wrappers*, so it will be useful to see examples of monads which are not collections.

3.3.2 Option monad

One of the first monads that many people encounter is the **Option** monad (referred to as the Maybe monad in Haskell, and Optional in Java 8). You can think of it as being a strange kind of "collection" that can contain at most one element. So it will either contain an element or it won't, and so can be used to wrap the result of a computation which might fail. If the computation succeeds, the value computed can be wrapped in the Option (using the type **Some**), and if it fails, it will not contain a value of the required type, but simply be the value **None**. It provides a referentially transparent and type-safe alternative to raising exceptions or returning **NULL** references. We can transform Options using **map**.

```
val three = Option(3)
//three: Option[Int] = Some(3)
val twelve = three map (_ * 4)
//twelve: Option[Int] = Some(12)
```

encounter is the **Option** monad (referred to as the Maybe monad in Haskel

```
val four = Option(4)
//four: Option[Int] = Some(4)
val twelveB = three map (i => four map (i * _))
//twelveB: Option[Option[Int]] = Some(Some(12))
```

Here we have ended up with an Option wrapped in another Option, which is not what we want. But we now know the solution, which is to replace the first **map** with **flatMap**, or better still, use a for–comprehension.

```
val twelveC = three flatMap (i => four map (i * _))
//twelveC: Option[Int] = Some(12)
val twelveD = for {
  i <- three
  j <- four
} yield ( i * j )
//twelveD: Option[Int] = Some(12)
```

Again, the for-comprehension is a little bit easier to understand than the chaining of calls to **flatMap** and **map**. Note that in the for–comprehension we don't worry about whether or not the Options actually contain values – we just concentrate on the "happy path", where they both do, safe in the knowledge that the **Option** monad will take care of the failure cases for us. Two of the possible failure cases are illustrated below.

```
val oops: Option[Int] = None
//oops: Option[Int] = None
val oopsB = for {
  i<- three
  j<- oops
} yield (i * j)
//oopsB: Option[Int] = None
val oopsC = for {
  i <- oops
  j <- four
} yield (i * j)
//oopsC: Option[Int] = None
```

This is a typical benefit of code written in a monadic style. We chain together multiple computations thinking only about the canonical case and trusting the monad to take care of any additional computational context for us.

IEEE floating point and NaN

Those with a background in scientific computing are probably already familiar with the **NaN** value in IEEE floating point. In many regards, this value and the rules around its behaviour mean that **Float** and **Double** types in IEEE compliant languages behave as an Option monad with **NaN** as the **None** value. This is simply illustrated below.

```
val nan = Double.NaN
//nan: Double = NaN
3.0 * 4.0
//res0: Double = 12.0
3.0 * nan
//res1: Double = NaN
nan * 4.0
//res2: Double = NaN
```

The NaN value arises naturally when computations fail. eg

```
val nanB = 0.0 / 0.0
//nanB: Double = NaN
```

This Option-like behaviour of **Float** and **Double** means that it is quite rare to see examples of **Option[Float]** or **Option[Double]** in Scala code. But note that this only works for **Float**s and **Double**s, and not for other types, including, say, **Int**.

```
val nanC=0/0
//java.lang.ArithmeticException: / by zero
//  ... 32 elided
```

But there are some disadvantages of the IEEE approach, as discussed elsewhere: https://blogs.janestreet.com/making-something-out-of-nothing-or-why-none-is-better-than-nan-and-null/

Option for matrix computations

Good practical examples of scientific computations which can fail crop up frequently in numerical linear algebra, so it's useful to see how Option can simplify code in that context. Note that the code in this section requires the Breeze library, which we will look at in more detail in the next Chapter, so should be run from an **sbt console** using an **sbt** build file including a Breeze dependency. In statistical applications, one often needs to compute the Cholesky factorisation of a square symmetric matrix. This operation is built into Breeze as the function **cholesky**. However the factorisation will fail if the matrix provided is not positive semi-definite, and in this case the cholesky function will throw a runtime exception. We will use the built in **cholesky** function to create our own function, **safeChol** (using a monad called **Try** which is closely related to the **Option** monad) returning an Option of a **matrix** rather than a matrix. This function will not throw an exception, but instead return **None** in the case of failure, as illustrated below.

```
import breeze.linalg._
//import breeze.linalg._
def safeChol(m: DenseMatrix[Double]):
  Option[DenseMatrix[Double]] = scala.util.Try(cholesky(m)).toOption
//safeChol: (m: breeze.linalg.DenseMatrix[Double])Option[breeze.linalg.DenseMatrix[D
val m = DenseMatrix((2.0, 1.0), (1.0, 3.0))
//m: breeze.linalg.DenseMatrix[Double] =
//2.0  1.0
//1.0  3.0
// column major ordering like Matlab, and 0-based indexing line Numphy
m.toArray
//res0: Array[Double] = Array(2.0, 1.0, 1.0, 3.0)
val c = safeChol(m)
//c: Option[breeze.linalg.DenseMatrix[Double]] =
//Some(1.4142135623730951  0.0
//0.7071067811865475  1.5811388300841898  )
val m2 = DenseMatrix((1.0, 2.0), (2.0, 3.0))
//m2: breeze.linalg.DenseMatrix[Double] =
//1.0  2.0
//2.0  3.0
val c1 = = cholesky(m2)
//breeze.linalg.NotConvergedException:
//  at breeze.linalg.cholesky$ImplCholesky_DM$.apply(cholesky.scala:43)
//  at breeze.linalg.cholesky$ImplCholesky_DM$.apply(cholesky.scala:16)
```

```
//  at breeze.generic.UFunc$class.apply(UFunc.scala:48)
//  at breeze.linalg.cholesky$.apply(cholesky.scala:15)
//  ... 32 elided
val c2 = safeChol(m2)
//c2: Option[breeze.linalg.DenseMatrix[Double]] = None
```

A Cholesky factorisation is often followed by a forward or backward solve. This operation may also fail, independently of whether the Cholesky factorisation fails. There doesn't seem to be a forward solve function directly exposed in the Breeze API, but we can easily define one, which I call **dangerousForwardSolve**, as it will throw an exception if it fails, just like the **cholesky** function. But just as for the **cholesky** function, we can wrap up the dangerous function into a safe one that returns an Option.

```
import com.github.fommil.netlib.BLAS.{getInstance => blas}
//import com.github.fommil.netlib.BLAS.{getInstance=>blas}
def dangerousForwardSolve(A: DenseMatrix[Double],
      y: DenseVector[Double]): DenseVector[Double] = {
  val yc = y.copy
  blas.dtrsv("L", "N", "N", A.cols, A.toArray, A.rows, yc.data, 1)
  yc
}
//dangerousForwardSolve: (A: breeze.linalg.DenseMatrix[Double], y: breeze.linalg.Den
def safeForwardSolve(A: DenseMatrix[Double],
      y:DenseVector[Double]): Option[DenseVector[Double]]=
      scala.util.Try(dangerousForwardSolve(A, y)).toOption
```

Now we can write a very simple function which chains these two operations together, as follows.

```
def safeStd(A: DenseMatrix[Double],
      y: DenseVector[Double]):
      Option[DenseVector[Double]] =
  for {
      L <- safeChol(A)
      z <- safeForwardSolve(L, y)
  } yield z
//safeStd: (A: breeze.linalg.DenseMatrix[Double], y: breeze.linalg.DenseVector[Doubl
safeStd(m,DenseVector(1.0,2.0))
//res6: Option[breeze.linalg.DenseVector[Double]] = Some(DenseVector(0.7071067811865
```

Note how clean and simple this function is, concentrating purely on the "happy path" where both operations succeed and letting the Option monad worry about the three different cases where at least one of the operations fails.

```
def getT[T](opt: Option[T], val_if_none: T): T =
  opt match {
    case Some(x) => opt.get
    case None    => val_if_none
  }
}
```

3.3.3 The Future monad

Let's finish this Chapter with a monad for parallel and asynchronous computation, the **Future** monad. The Future monad is used for wrapping up slow computations and dispatching them to another thread for completion. The call to **Future** returns immediately, allowing the calling thread to continue while the additional thread processes the slow work. So at that stage, the **Future** will not have completed, and will not contain a value, but at some (unpredictable) time in the future it (hopefully) will (hence the name). In the following code snippet I construct two **Future**s that will each take at least 10 seconds to complete. On the main thread I then use a for–comprehension to chain the two computations together. Again, this will return immediately returning another **Future** that at some point in the future will contain the result of the derived computation. Then, purely for illustration, I force the main thread to stop and wait for that third future (**f3**) to complete, printing the result to the console.

```
import scala.concurrent.duration._
//import scala.concurrent.duration._
import scala.concurrent.{Future, ExecutionContext,Await}
//import scala.concurrent.{Future, ExecutionContext, Await}
import ExecutionContext.Implicits.global
val f1=Future{
  Thread.sleep(10000)
  1 }
//f1: scala.concurrent.Future[Int] = List()
val f2=Future{
  Thread.sleep(10000)
  2 }
//f2: scala.concurrent.Future[Int] = List()
val f3=for {
   v1 <- f1
   v2 <- f2
   } yield (v1+v2)
//f3: scala.concurrent.Future[Int] = List()
println(Await.result(f3,30.second))
//3
```

When you paste this into your console you should observe that you get the result in 10 seconds, as **f1** and **f2** execute in parallel on separate threads. So the **Future** monad is an alternative to parallel collections to get started with parallel and async programming in Scala.

Here I've tried to give a quick informal introduction to the monad concept, and tried to use examples that will make sense to those interested in scientific and statistical computing. There's loads more to say about monads, and there are many more commonly encountered useful monads that haven't been covered here. I've skipped over lots of details, especially those relating to the formal definitions of functors and monads, including the laws that **map** and **flatMap** must satisfy and why. We'll return to these issues later.

Chapter 4

Scientific and statistical computing

4.0.1 Scala math standard library: scala.math

Many of the most often used mathematical functions can be found in the package **scala.math**.

```
math.log(2.0)
//res0: Double = 0.6931471805599453
math.sin(1.0)
//res1: Double = 0.8414709848078965
log(2.0)
//<console>:12: error: not found: value log
//       log(2.0)
//       ^
import math.log
//import math.log
log(2.0)
//res3: Double = 0.6931471805599453
```

The REPL has TAB– completion, so typing math. and then hitting the TAB key will show a list of items in the math package.

```
math.<TAB>
//BigDecimal                    ScalaNumericConversions   max
//BigInt                        abs                       min
//E                             acos                      package
//Equiv                         asin                      pow
//Fractional                    atan                      random
//IEEEremainder                 atan2                     rint
//Integral                      cbrt                      round
//LowPriorityEquiv              ceil                      signum
//LowPriorityOrderingImplicits  cos                       sin
//Numeric                       cosh                      sinh
//Ordered                       exp                       sqrt
//Ordering                      expm1                     tan
//PartialOrdering               floor                     tanh
//PartiallyOrdered              hypot                     toDegrees
```

```
//Pi                              log                     toRadians
//ScalaNumber                     log10                   ulp
//ScalaNumericAnyConversions      log1p

import math._
//import math._
sin(Pi/2)
//res0: Double = 1.0
exp(log(sin(Pi/2)))
//res1: Double = 1.0
sin(asin(0.1))
//res2: Double = 0.1
atan(1)*4
//res3: Double = 3.141592653589793
log(sqrt(exp(1)))
//res4: Double = 0.5
abs(min(-1,2))
//res5: Int = 1
pow(2,8)
//res6: Double = 256.0
random
//res7: Double = 0.16990524978066768
random
//res8: Double = 0.9127627491418674
random
//res9: Double = 0.1441973077460399
floor(random*3)
//res10: Double = 1.0
floor(random*3)
//res11: Double = 1.0
floor(random*3)
//res12: Double = 2.0
floor(random*3)
//res13: Double = 2.0
```

Doing wild-card imports like import **math._** is often frowned upon in production code, as it clutters the name-space, but it can be especially useful for interactive experiments in the REPL.

4.0.2 Note: Spire cfor macro

Spire provides a loop macro called cfor whose syntax bears a slight resemblance to a traditional for-loop from C or Java. This macro expands to a tail-recursive function, which will inline literal function arguments. The macro can be nested in itself and compares favorably with other looping constructs in Scala such as for and while:

The purpose of cfor is not to do anything you can't do with ordinary Scala iterator primitives. The sole purpose of cfor is performance, since unfortunately ordinary scala iteration performance is pretty terrible.

```
import spire.syntax.cfor.{cfor, cforRange, cforRange2}

cfor(0)( _ < 10, _ + 1) ( i => {
  print(i)
} )
//0123456789
cfor(0)( i => i < 10, i => i + 1) ( i => {
  print(i)
} )
//0123456789

cforRange( 0 until 10 by 2 ) ( i => { print(i) } )
//02468
cforRange2( 0 to 2, 10 until 12 ) ( (i, j) => { print(i, j) } )
//(0,10)(0,11)(1,10)(1,11)(2,10)(2,11)
```

4.1 Breeze: A quick introduction

4.1.1 Introduction

In this Chapter I want to give a quick taste of the Breeze numerical library to indicate its capabilities. To follow along you should run a **console** from an **sbt** session with a Breeze dependency, or you could use a Scala Worksheet from inside a ScalaIDE project with a Breeze dependency.

4.1.2 Non-uniform random number generation

We begin by taking a quick look at everyone's favourite topic of nonuniform random number generation. Let'start by generating a couple of draws from a Poisson distribution with mean 3.

```
import breeze.stats.distributions._
//import breeze.stats.distributions._
val poi = Poisson(3.0)
//poi: breeze.stats.distributions.Poisson = Poisson(3.0)
poi.draw
//res0: Int = 6
poi.draw
//res1: Int = 4
```

If more than a single draw is required, an iid (independent and identically distributed) **sample** can be obtained

```
val x = poi.sample(10)
//x: IndexedSeq[Int] = Vector(5, 3, 1, 2, 4, 5, 2, 1, 4, 2)
x
//res2: IndexedSeq[Int] = Vector(5, 3, 1, 2, 4, 5, 2, 1, 4, 2)
x.sum
//res3: Int = 29
x.length
```

```
//res4: Int = 10
x.sum.toDouble/x.length
//res5: Double = 2.9
```

The probability mass function (PMF = $\lambda^k e^{-\lambda}/k!$) of the Poisson distribution is also available.

```
poi.probabilityOf(2)
//res6: Double = 0.22404180765538775
x map {x => poi.probabilityOf(x)}
//res9: IndexedSeq[Double] = Vector(0.10081881344492458, ...)
x map {poi.probabilityOf(_)}
//res10: IndexedSeq[Double] = Vector(0.10081881344492458, ...)
```

Obviously, Gaussian variables (and Gamma, and several others) are supported in a similar way.

```
val gau=Gaussian(0.0,1.0)
//gau: breeze.stats.distributions.Gaussian = Gaussian(0.0, 1.0)
gau.draw
//res11: Double = 0.5358890494992097
gau.draw
//res12: Double = -1.0710648279538835
val y=gau.sample(20)
//y: IndexedSeq[Double] = Vector(3.2360057581913373,...)
y
//res13: IndexedSeq[Double] = Vector(3.2360057581913373, ...)
y.sum/y.length
//res14: Double = 0.1820848884511536
y map {gau.logPdf(_)}
//res15: IndexedSeq[Double] = Vector(-6.154805166728419, ...)
Gamma(2.0,3.0).sample(5)
//res16: IndexedSeq[Double] = Vector(1.0580607210792405, ...)
```

This is all good stuff for those of us who like to do Markov chain Monte Carlo. There are not masses of statistical data analysis routines built into Breeze, but a few basic tools are provided, including some basic summary statistics.

```
import breeze.stats._
mean(y)
//res17: Double = -0.05138816029069082
variance(y)
//res18: Double = 0.4647546420020672
val mV = meanAndVariance(y)
//mV: breeze.stats.MeanAndVariance = MeanAndVariance(-0.05138816029069082,0.46475464
mV.mean
//res18: Double = -0.05138816029069082
mV.variance
//res19: Double = 0.4647546420020672
```

Note

Please note that if the inverse function of a cumulative distribution function (CDF) can be analytically or numerically calculated, random numbers obeying the CDF can be generated from the uniform random numbers; $X = F_X^{-1}(U)$, where F_X^{-1} is the inverse function of the CDF (F_X) of random variable X, and $U(0, 1)$ is the uniform random variable on the interval $(0, 1)$.

4.1.3 Vectors and matrices

Support for linear algebra is an important part of any scientific library. Here the Breeze developers have made the wise decision to provide a nice Scala interface to netlib-java. This in turn calls out to any native optimised BLAS or LAPACK libraries installed on the system, but will fall back to Java code if no optimised libraries are available. This means that linear algebra code using Scala and Breeze should run as fast as code written in any other language, including C, C++ and Fortran, provided that optimised libraries are installed on the system. For further details see the Breeze linear algebra cheat sheet [7)]. Let's start by creating and messing with a dense vector.

```
import breeze.linalg._
//import breeze.linalg._
val v=DenseVector(y.toArray)
//v: breeze.linalg.DenseVector[Double] = DenseVector(-1.248668434699002, 0.640927962
v(1) = 0
//
v
//res1: breeze.linalg.DenseVector[Double] = DenseVector(-1.248668434699002, 0.0, ...
v(1 to 3) := 1.0
//res2: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 1.0, 1.0)
v
//res3: breeze.linalg.DenseVector[Double] = DenseVector(-1.248668434699002, 1.0, 1.0
v >:> 0.0
//res4: breeze.linalg.BitVector = BitVector(1, 2, 3, 7, 8, 9, 15)
(v >:> 0.0).toArray
//res5: Array[Boolean] = Array(false, true, ...
```

Next let's create and mess around with some dense matrices.

```
val m = new DenseMatrix(5,4,linspace(1.0,20.0,20).toArray)
//1.0  6.0   11.0  16.0
//2.0  7.0   12.0  17.0
//3.0  8.0   13.0  18.0
//4.0  9.0   14.0  19.0
//5.0  10.0  15.0  20.
m
//res21: breeze.linalg.DenseMatrix[Double] =
//1.0  6.0   11.0  16.0
//2.0  7.0   12.0  17.0
//3.0  8.0   13.0  18.0
//4.0  9.0   14.0  19.0
```

```
//5.0  10.0  15.0  20.0
m.rows
//res23: Int = 5
m.cols
//res24: Int = 4
m(::,1)
//res25: breeze.linalg.DenseVector[Double] = DenseVector(6.0, 7.0, 8.0, 9.0, 10.0)
m(1,::)
//res26: breeze.linalg.Transpose[breeze.linalg.DenseVector[Double]] = Transpose(Dens
m(1,::) := linspace(1.0,2.0,4).t
//res28: breeze.linalg.Transpose[breeze.linalg.DenseVector[Double]] = Transpose(Dens
m
//1.0  6.0                 11.0                16.0
//1.0  1.3333333333333333  1.6666666666666665  2.0
//3.0  8.0                 13.0                18.0
//4.0  9.0                 14.0                19.0
//5.0  10.0                15.0                20.0
```

Note in the above how the **::** notation is used to mean *all*, so that **m(1,::)** means all entries for row 1 (which is the second row) and **m(::,1)** means all entries for column 1.

Users of R will be familiar with the **apply** command for applying functions to rows and columns of matrices. This is referred to as *broadcasting* in Breeze. In Breeze the **::** notation is accompanied by the * notation, where * denotes "foreach" — the index to be looped over and kept. So **sum(m(::,*))** means "foreach **k** ranging over the columns of **m**, compute **sum(m(::,k))** and keep those column sums in a row vector.

```
// broadcasting (like "apply" in R)
sum(m(::,*))            // column sum
//res32: breeze.linalg.Transpose[breeze.linalg.DenseVector[Double]] = Transpose(Dens
sum(m(*,::))            // row sum
//res33: breeze.linalg.DenseVector[Double] = DenseVector(34.0, 6.0, 42.0, 46.0, 50.0
val n = m.t
//n: breeze.linalg.DenseMatrix[Double] =
//1.0   1.0                 3.0   4.0   5.0
//6.0   1.3333333333333333  8.0   9.0   10.0
//11.0  1.6666666666666665  13.0  14.0  15.0
//16.0  2.0                 18.0  19.0  20.0
val o = m*n
//o: breeze.linalg.DenseMatrix[Double] =
//414.0              59.33333333333333  482.0              516.0              550.0
//59.33333333333333  9.555555555555555  71.33333333333333  77.33333333333333  83.333
//482.0              71.33333333333333  566.0              608.0              650.0
//516.0              77.33333333333333  608.0              654.0              700.0
//550.0              83.33333333333333  650.0              700.0              750.0
val p = n*m
//p: breeze.linalg.DenseMatrix[Double] =
//52.0               117.33333333333333  182.66666666666666  248.0
```

```
//117.33333333333333  282.77777777777777  448.22222222222223  613.6666666666667
//182.66666666666666  448.22222222222223  713.7777777777778   979.3333333333334
//248.0               613.6666666666667   979.3333333333334   1345.0
DenseMatrix.eye[Double](3)
//res2: breeze.linalg.DenseMatrix[Double] =
//1.0  0.0  0.0
//0.0  1.0  0.0
//0.0  0.0  1.0
```

So, messing around with vectors and matrices is more-or-less as convenient as in well-known dynamic and math languages. To conclude this section, let us see how to simulate some data from a regression model and then solve the least squares problem to obtain the estimated regression coefficients. We will simulate 1,000 observations from a model with 2 covariates (plus an intercept).

```
val N = 1000
//N: Int = 1000
val P = 2
//P: Int = 2
val XX = new DenseMatrix(N,P,gau.sample(P*N).toArray)
//XX: breeze.linalg.DenseMatrix[Double] =
//1.2916324982216227     0.9382783929386121
//0.1428544587374136     0.7655910910660422
//2.5416689479775796     -0.3467029782936591
//0.914023866200091      3.163236902933159
//0.40802673780281645    1.371030672077766
// ...
val X = DenseMatrix.horzcat(
          DenseVector.ones[Double](N).toDenseMatrix.t, XX)
//X: breeze.linalg.DenseMatrix[Double] =
//1.0  1.2916324982216227     0.9382783929386121
//1.0  0.1428544587374136     0.7655910910660422
// ...
val b0 = linspace(1.0,2.0,P+1)
//b0: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 1.5, 2.0)
val y0 = X * b0
//y0: breeze.linalg.DenseVector[Double] = DenseVector(4.8140055332096585, ...
val y = y0 + DenseVector(gau.sample(1000).toArray)
//y: breeze.linalg.DenseVector[Double] = DenseVector(5.4735469394154475, 2.263343274
val b = X \ y              // linearsolve
//b: breeze.linalg.DenseVector[Double] = DenseVector(1.058096900659516, 1.5158803821
```

Note the use of the notation **X** **y** for the solution of the linear equation $Xb = y$ (for b). In the case of a (square) invertible X, this will give $X^{-1}y$, but for an over-determined system (like this) it will give the b which minimises $\|Xb - y\|^2$ – the *least squares solution*, and this is computed using a QR-decomposition of X [8)]; see page 99.

So all of the most important building blocks for statistical computing are included in the Breeze library.

4.1.4 Reading and writing matrices from and to disk

The easiest way to get data in and out of matrices is via Breeze's simple CSV reader and writer, as illustrated below.

```
import java.io.File
//
csvwrite(new File("X-matrix.csv"),X)
//
val X3 = csvread(new File(X-matrix.csv"))
X3: breeze.linalg.DenseMatrix[Double] =
//1.0  1.2916324982216227     0.9382783929386121
//1.0  0.1428544587374136     0.7655910910660422
//...
```

4.2 Breeze: Numerical linear algebra

The discussion of the
operator near the end of the previous section hints at some of the numerical linear algebra routines that are included in Breeze. We will now look at these in a bit more detail.

4.2.1 Symmetric eigen-decomposition

```
p
//res28: breeze.linalg.DenseMatrix[Double] =
//52.0                117.33333333333333  182.66666666666666  248.0
//117.33333333333333  282.77777777777777  448.22222222222223  613.6666666666667
//182.66666666666666  448.22222222222223  713.7777777777778   979.3333333333334
//248.0               613.6666666666667   979.3333333333334   1345.0
val e=eigSym(p)
//e: breeze.linalg.eigSym.DenseEigSym =
//EigSym(DenseVector(-5.5047236546692537E-14, 1.0833794172869972E-13, 7.857939135975
e.eigenvalues
//res29: breeze.linalg.DenseVector[Double] = DenseVector(-5.5047236546692537E-14, 1.
e.eigenvectors
//res30: breeze.linalg.DenseMatrix[Double] =
//-0.498663456654444   0.22657174801251923   -0.8248907010788654    -0.1398403778370
//0.49600799876027224  -0.6737774596747976   -0.42669574478180183   -0.3434104561382
//0.5039743724427688   0.6678396753120598    -0.028500788484718136  -0.5469805344395
//-0.5013189145486021  -0.22063396364977528  0.36969416781235653    -0.7505506127407
```

4.2.2 Singular value decomposition (SVD)

Note for SVD:

The singular value decomposition is to decompose a complex matrix M into the product of a unitary matrix U ($U^* = U^{-1}$), a complex rectangular diagonal matrix, and a unitary matrix V as

follows.

$$M_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^*_{n\times n} \tag{4.1}$$

where

$$\Sigma_{ij} = 0 \text{ for } i \neq j\,,\, UU^* = U^*U = I\,,\, VV^* = V^*V = I\,, \tag{4.2}$$

with Kronecker's δ_{ij}, and U^* is the complex-conjugate and transpose of U. Then, the following equations are derived.

$$M^*M = (U\Sigma V^*)^*(U\Sigma V^*) \tag{4.3}$$
$$M^*M = V\overline{\Sigma}U^*U\Sigma V^* \tag{4.4}$$
$$V^*(M^*M)V = \overline{\Sigma}\Sigma \tag{4.5}$$
$$(MV)^*(MV) = \overline{\Sigma}\Sigma \tag{4.6}$$

Also,

$$MM^* = (U\Sigma V^*)(U\Sigma V^*)^* \tag{4.7}$$
$$U^*(MM^*)U = \Sigma\overline{\Sigma} \tag{4.8}$$

Therefore,

- U, the left-singular vectors of M, is a set of orthonormal eigenvectors of MM^*.
- V, the right-singular vectors of M, is a set of orthonormal eigenvectors of M^*M.
- Σ_{ii}, singlular values, are the square roots of the eigen values of both MM^* and M^*M.

Applications that employ the SVD include computing the pseudoinverse, least squares fitting of data, multivariable control, matrix approximation, and determining the rank, range and null space of a matrix.

We start with the full SVD:

```
p
//res40: breeze.linalg.DenseMatrix[Double] =
//52.0                117.33333333333333  182.66666666666666  248.0
//117.33333333333333  282.77777777777777  448.22222222222223  613.6666666666667
//182.66666666666666  448.22222222222223  713.7777777777778   979.3333333333334
//248.0               613.6666666666667   979.3333333333334   1345.0
val s=svd(p)                 // full SVD
//s: breeze.linalg.svd.DenseSVD =
//SVD(-0.13984037783705727  -0.8248907010788751, ...
s.U
//res31: breeze.linalg.DenseMatrix[Double] =
//-0.13984037783705727  -0.8248907010788751    -0.007271090327652322  -0.54767429302
//-0.34341045613829163  -0.42669574478178857   0.41790710335128994    0.724812839958
//-0.5469805344395258   -0.028500788484715378  -0.8140009357196326    0.193397199171
//-0.7505506127407597   0.3696941678123502     0.40336492269599244    -0.37053574610
```

```
s.S
//res32: breeze.linalg.DenseVector[Double] = DenseVector(2385.69761641958, 7.8579391
s.Vt
//res33: breeze.linalg.DenseMatrix[Double] =
//-0.13984037783705763  -0.3434104561382918   -0.5469805344395259    -0.750550612740
//-0.8248907010788822   -0.42669574478178196  -0.028500788484708495  0.3696941678123
//-0.3564872784300705   0.7852597534511133    -0.5010576716119877    0.0722851965909
//0.41583268307998733   -0.28873364821216885  -0.6700307528157005    0.5429317179478
s.U * diag(s.S) * s.Vt
//res39: breeze.linalg.DenseMatrix[Double] =
//52.000000000000014  117.33333333333306  182.66666666666623  247.9999999999996
//117.33333333333334  282.7777777777777   448.22222222222206  613.6666666666667
//182.66666666666669  448.2222222222223   713.7777777777776   979.3333333333335
//248.00000000000003  613.6666666666666   979.3333333333331   1344.9999999999998
```

In statistical applications we rarely want the full SVD, as we typically deal with tall, thin $N \times p$ matrices with $N >> p$. In this case we can get the "thin" SVD, called **reduced** in Breeze.

```
m
//res35: breeze.linalg.DenseMatrix[Double] =
//1.0  6.0                 11.0                16.0
//1.0  1.3333333333333333  1.6666666666666665  2.0
//3.0  8.0                 13.0                18.0
//4.0  9.0                 14.0                19.0
//5.0  10.0                15.0                20.0
val ts=svd.reduced(m)
//ts: breeze.linalg.svd.DenseSVD =
SVD(-0.41409513816680077   0.7907147500096334    0.38814262434379926    -0.211382487
//...
ts.U
//res34: breeze.linalg.DenseMatrix[Double] =
//-0.41409513816680077   0.7907147500096334    0.38814262434379926    -0.21138248773
//-0.061634609473825513  -0.2504034467665379   -0.030170726645010366  -0.53089952999
//-0.4870128530154549    0.14117668550853302   -0.5963009520201685    0.526951395267
//-0.5234717104397821    -0.18359234674201724  -0.38008241109820085   -0.56230585957
//-0.5599305678641093    -0.5083613789925672   0.5902521205509041     0.282130254044
ts.S
//res36: breeze.linalg.DenseVector[Double] = DenseVector(48.843603638752754, 2.80320

//res36: breeze.linalg.DenseVector[Double] = DenseVector(48.843603638752754, 2.80320

//res36: breeze.linalg.DenseVector[Double] = DenseVector(48.843603638752754, 2.80320
ts.Vt
//res37: breeze.linalg.DenseMatrix[Double] =
//-0.13984037783705755  -0.3434104561382917   -0.5469805344395258   -0.7505506127407
/-0.8248907010788696   -0.42669574478179495  -0.02850078848471957  0.369694167812355
//0.5078471767142547    -0.5242112081926765   -0.47511911375741117  0.49148314523583
```

```
//0.20516150979986608   -0.6520756161712953   0.6886667029429929    -0.2417525965715
ts.U * diag(ts.S) * ts.Vt
//res38: breeze.linalg.DenseMatrix[Double] =
//1.0000000000000029  6.000000000000008   11.000000000000012  16.00000000000002
//1.0000000000000009  1.333333333333336   1.6666666666666698  2.0000000000000053
//3.000000000000001   8.000000000000004   13.000000000000007  18.00000000000001
//4.000000000000002   9.000000000000007   14.000000000000009  19.000000000000014
//5.000000000000002   10.000000000000005  15.000000000000007  20.000000000000014
```

Crucially, unlike the full SVD, the reduced SVD does not involve any $N \times N$ matrices.

4.2.3 Principal components analysis (PCA)

Note: Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components.

Suppose that M in Eq. 4.1 is a data matrix ($m > n$) such that the M^*M corresponds to its covariance matrix. Then, MV is the representation of data in the new coordinate system, in which there is no correlation between coordinates. Coordinates corresponding to the larger eigen values are called principal components.

Breeze has a function for PCA built-in: **breeze.linalg.PCA**. However, this works via a spectral decomposition of a covariance matrix (analogous to the R function **princomp**). Generally it is better to construct principal components directly from the SVD of the centred data matrix (analogous to the R function **prcomp**). We can write some simple code to do this as follows:

```
import breeze.linalg._
import breeze.stats._
case class Pca(mat: DenseMatrix[Double]) {
  val xBar = mean(mat(::,*)).t
  val x = mat(*,::) - xBar
  val SVD = svd.reduced(x)
  val loadings = SVD.Vt.t
  val sdev = SVD.S / math.sqrt(x.rows - 1)
  lazy val scores = x * loadings
}
```

Calling this with **Pca(myMatrix)** will return an object of type **Pca**. See how the Breeze broadcast notation is used first to compute the column sums of a matrix, and then to *sweep out* the column sums from the matrix. Then the usual PCA attributes can be constructed simply from the SVD. Note how **scores** has been declared a **lazy val**, to be computed only if required. We can use this as follows.

```
val X = DenseMatrix((1.0,1.5),(1.5,2.0),(2.0,1.5))
//X: breeze.linalg.DenseMatrix[Double] =
//1.0  1.5
// ...
val pca = Pca(X)
//pca: Pca =
```

```
//Pca(1.0  1.5
//1.5  2.0
//2.0  1.5  )
pca.sdev
//res0: breeze.linalg.DenseVector[Double] = DenseVector(0.5, 0.28867513459481287)
pca.loadings
//res1: breeze.linalg.DenseMatrix[Double] =
//1.0  -0.0
//0.0  -1.0
pca.scores
//res2: breeze.linalg.DenseMatrix[Double] =
//-0.5  0.16666666666666674
//0.0   -0.33333333333333326
//0.5   0.16666666666666674
pca.scores
```

Note that the loadings defined here are consistent with how they are usually defined for PCA in languages such as R, but they are the transpose of the loadings returned by the Breeze PCA class.

4.2.4 QR and Cholesky decompositions

Note: A QR decomposition (also called a QR factorization) of a matrix is a decomposition of a matrix **A** into a product **A = QR** of an orthogonal/unitary matrix **Q** and an upper triangular matrix **R**. QR decomposition is often used to solve the linear least squares problem, and is the basis for a particular eigenvalue algorithm, the QR algorithm [8)].

For $n \geq p$,

$$A_{n\times p} = Q_{n\times n}R_{n\times p} \text{ where } QQ^* = Q^*Q = I \text{ and } R = \text{upper triangular matrix} \tag{4.9}$$

$$A = QR = Q\begin{bmatrix} R_1 \\ 0 \end{bmatrix} = [Q_1 Q_2]\begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1 \tag{4.10}$$

where R_1 is an $p \times p$ upper triangular matrix, and Q_1 and Q_2 are $n \times p$ and $n \times (n - p)$ matrices with orthogonal columns. $A = Q_1R_1$ is called the thin/reduced QR decomposition.

If A is of full rank p and we require that the diagonal elements of R_1 are positive, then R_1 and Q_1 are unique, but in general Q_2 is not. R_1 is then equal to the upper triangular factor of the Cholesky decomposition of A^*A.

As above, we will typically want the reduced QR decomposition in statistical applications.

```
m
//res5: breeze.linalg.DenseMatrix[Double] =
//1.0  6.0                 11.0                 16.0
//1.0  1.3333333333333333  1.6666666666666665  2.0
//3.0  8.0                 13.0                 18.0
```

```
//4.0  9.0                14.0                19.0
//5.0  10.0               15.0                20.0
val q=qr.reduced(m)          // thin QR
//q: breeze.linalg.qr.DenseQR =
//QR(-0.13867504905630734  0.881744764716757      0.1402689190825892    0.427930872
//-0.1386750490563073   -0.21741651732741957   0.9622045838542994    0.0874126143424
// ...
q.q
//res6: breeze.linalg.DenseMatrix[Double] =
//-0.13867504905630734  0.881744764716757      0.1402689190825892    0.427930872164
//-0.1386750490563073   -0.21741651732741957   0.9622045838542994    0.0874126143424
//-0.41602514716892186  0.2898886897698929     0.07426633834284485   -0.733230232444
//-0.5547001962252291   -0.006039347703539422  -0.06813863044651419  -0.186987947538
//-0.6933752452815364   -0.30196738517697175   -0.2105435992358733   0.4864598001963
q.r
//res7: breeze.linalg.DenseMatrix[Double] =
//-7.211102550927978  -16.271205755940056  -25.33130896095213      -34.3914121659642
//0.0                 4.245661435588219    8.49132287117644        12.73698430676466
//0.0                 0.0                  -9.930136612989092E-16  -3.57484918067607
//0.0                 0.0                  0.0                     1.745558720414801
cholesky(DenseMatrix((3.0,1.0),(1.0,2.0)))
//res8: breeze.linalg.DenseMatrix[Double] =
//1.7320508075688772  0.0
//0.5773502691896258  1.2909944487358056
```

Note: The Cholesky decomposition or factorization is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, which is useful for efficient numerical solutions, e.g. Monte Carlo simulations. It was discovered by André-Louis Cholesky for real matrices. When it is applicable, the Cholesky decomposition is roughly twice as efficient as the LU decomposition for solving systems of linear equations.

$$A = LL^* \tag{4.11}$$

(4.12)

where

$$A = A^* \quad \text{and Hermitian positive-definite} \tag{4.13}$$

$$L = \begin{bmatrix} L_{11} & 0 & 0 \\ & \dots & \\ L_{ij} & \dots L_{ii} & 0 \\ & \dots & \\ L_{ij} & \dots & L_{nn} \end{bmatrix} \qquad L_{ii} > 0 \tag{4.14}$$

If A is positive definite, the Cholesky decomposition is unique. The converse holds trivially: if A can be written as LL^* for some invertible L, lower triangular or otherwise, then A is Hermitian and positive definite.

In the case that A is positive semi-definite, $A = LL^*$ is possible if L_{ii} are allowed to be zero, but not unique.

```
cholesky(DenseMatrix((3.0,1.0),(1.0,2.0)))
//res8: breeze.linalg.DenseMatrix[Double] =
//1.7320508075688772  0.0
//0.5773502691896258  1.2909944487358056
```

4.3 Breeze: Scientific computing

4.3.1 Constants and special functions

There are some built-in constants.

```
math.Pi
//res0: Double = 3.141592653589793
import breeze.numerics.constants._
Pi
//res1: Double = 3.141592653589793
E
//res2: Double = 2.718281828459045
eulerMascheroni
//res3: Double = 0.5772156649015329
```

There are also various physics constants, but these typically aren't required in statistical applications. There are also a number of special functions.

```
import breeze.numerics._
erf(2.0)                // error function
//res5: Double = 0.9953222650189527
erfinv(erf(2.0))
//res6: Double = 1.999999999999999
sigmoid(1.0)            // expit/logistic
//res7: Double = 0.7310585786300049
lgamma(4.0)             // log gamma function
//res8: Double = 1.791759469228055
exp(lgamma(4.0))
//res9: Double = 6.0
gammp(2.0,1.0)          // incomplete gamma
//res10: Double = 0.2642411176571152
digamma(2.0)
//res11: Double = 0.4227843322079321
lbeta(1.0,2.0)          // log beta function
//res12: Double = -0.6931471805599453
Bessel.i0(1.0)          // Bessel functions
//res13: Double = 1.2660658777520086
Bessel.i1(1.0)
```

```
//res14: Double = 0.5651591039924851
erf(DenseVector(1.0,2.0,3.0))
//res16: breeze.linalg.DenseVector[Double] = DenseVector(0.8427007929497151, 0.99532
```

The last example shows that special functions can typically be used as Breeze **UFuncs** in vectorised operations.

4.3.2 Integration and ODE-solving

There a couple of simple functions for 1-D integration.

```
// https://github.com/scalanlp/breeze/wiki/Integration
import breeze.integrate._
import math._
//
trapezoid(x => sin(x), 0, Pi, 10)
//res1: Double = 1.9796508112164832
simpson(x => sin(x), 0, Pi, 100)
//res2: Double = 2.0000000007042207
```

There are also functions for numerically integrating (forward-solving) ODE models.

```
import breeze.integrate._
// min and max time steps
val ode=new HighamHall54Integrator(0.001, 0.1)
//ode: breeze.integrate.HighamHall54Integrator = breeze.integrate.HighamHall54Integr

ode.integrate((x,t) => x, DenseVector(1.0),
              linspace(0,1,5).toArray)
//res2: Array[breeze.linalg.DenseVector[Double]] = Array(
//  DenseVector(1.0),
//  DenseVector(1.2840254150249555),
//  DenseVector(1.6487212664298634),
//  DenseVector(2.1170000083877234),
//  DenseVector(2.718281814377245))
```

4.3.3 Optimisation

There are also various routines for solving optimisation problems, with and without gradient information. **LBFGS** is a good default. It is a quasi-Newton method, but if gradients are not available a numerical gradient calculation can be used.

```
import breeze.linalg._
import breeze.optimize._
//import breeze.optimize._
def f(x: DenseVector[Double]) = (x(0)-5.0)*(x(0)-5.0)
//f: (x: breeze.linalg.DenseVector[Double])Double
val ag = new ApproximateGradientFunction(f)
```

```
//ag: breeze.optimize.ApproximateGradientFunction[Int,breeze.linalg.DenseVector[Doub
val opt = new LBFGS[DenseVector[Double]]()
//opt: breeze.optimize.LBFGS[breeze.linalg.DenseVector[Double]] = breeze.optimize.LB
opt.minimize(ag,DenseVector(0.0))
//res2: breeze.linalg.DenseVector[Double] = DenseVector(4.999999570300508)
```

But if gradients are available it's better to use them.

```
import breeze.linalg._
import breeze.optimize._
//import breeze.optimize._
def f(x: DenseVector[Double]) = (x(0)-5.0)*(x(0)-5.0)
//f: (x: breeze.linalg.DenseVector[Double])Double
def eg(x: DenseVector[Double]) = DenseVector(2.0*x(0) - 10.0)
//eg: (x: breeze.linalg.DenseVector[Double])breeze.linalg.DenseVector[Double]
val df = new DiffFunction[DenseVector[Double]] {
  def calculate(x: DenseVector[Double]) = (f(x),eg(x))
}
//df: breeze.optimize.DiffFunction[breeze.linalg.DenseVector[Double]] = <function1>
val opt = new LBFGS[DenseVector[Double]]()
//opt: breeze.optimize.LBFGS[breeze.linalg.DenseVector[Double]] = breeze.optimize.LB
opt.minimize(df,DenseVector(0.0))
//res4: breeze.linalg.DenseVector[Double] = DenseVector(5.0)
```

4.4 Breeze–Viz

Breeze–viz is the plotting library for Breeze. Note that this is separate from the main Breeze library, and therefore requires an additional dependency in your SBT build file:

```
  "org.scalanlp" %% "breeze-viz" % "0.13"
```

Then from a REPL:

```
// from the subsection, "Non-uniform random number generation"
import breeze.linalg._
import breeze.stats.distributions._
val gau=Gaussian(0.0,1.0)
val N = 1000
val P = 2
val XX = new DenseMatrix(N,P,gau.sample(P*N).toArray)
val X = DenseMatrix.horzcat(
          DenseVector.ones[Double](N).toDenseMatrix.t, XX)
val b0 = linspace(1.0,2.0,P+1)
val y0 = X * b0
val y = y0 + DenseVector(gau.sample(1000).toArray)
//
import breeze.plot._
```

```
//import breeze.plot._
val fig = Figure("My Figure")
//fig: breeze.plot.Figure = breeze.plot.Figure@7e772d11
fig.width=1000
//fig.width: Int = 1000
fig.height=800
//fig.height: Int = 800
val p1 = fig.subplot(0)
//p1: breeze.plot.Plot = breeze.plot.Plot@3fc261b7
p1 += hist(y,50)
//res0: breeze.plot.Plot = breeze.plot.Plot@3fc261b7
p1.xlim = (-10,15)
//p1.xlim: (Double, Double) = (-10.0,15.0)
p1.xlabel = "y"
//p1.xlabel: String = y
p1.title = "Distribution of observed response"
//p1.title: String = Distribution of observed response
fig.refresh
//
fig.saveas("hist.pdf")          // or "eps", or "png" for bitmap
```

So **Figure** corresponds to a window on your graphics device, and within a window you can have multiple sub-plots, but above we just have one, so it has index 0. You can add stuff to a **subplot** with the **+=** operator. Here we just add a histogram to an empty subplot. Note that there are lots of options which can be tweaked. It can sometimes be useful to call **refresh** to get the plot to update. Plots can be saved in vector format as PDF or EPS, or in bitmap format as PNG. Also note that plots are interactive, so right-clicking on a subplot will give a

```
val p2=fig.subplot(1,2,1)     // rows, cols, subplot index
//p2: breeze.plot.Plot = breeze.plot.Plot@639d8311
```

Here the arguments to **subplot** are the number of rows and columns in the array of subplots to be displayed in the figure, and the final number is the index of the subplot in question – here **1**, the second subplot in the array (row-major ordering). We can then add stuff to the subplot as before. Here we will first add some points and then a line.

```
p2 += plot(y0,y,'+',colorcode="red")
//res2: breeze.plot.Plot = breeze.plot.Plot@639d8311
val xvals=linspace(min(y0),max(y0),2)
//xvals: breeze.linalg.DenseVector[Double] = DenseVector(-6.606887188023363, 8.57432
p2 += plot(xvals,xvals,colorcode="[40, 40, 200]")
//res3: breeze.plot.Plot = breeze.plot.Plot@639d8311
p2.xlabel = "y0"
//p2.xlabel: String = y0
p2.ylabel = "y"
//p2.ylabel: String = y
p2.title = "Observed against true response"
//p2.title: String = Observed against true response
```

The final result of the above commands is shown in Fig. fig: 4.1. Some more plotting features are illustrated below: producing an image of a matrix and plotting multiple functions with a legend. The result of this is shown in Fig. fig: 4.2.

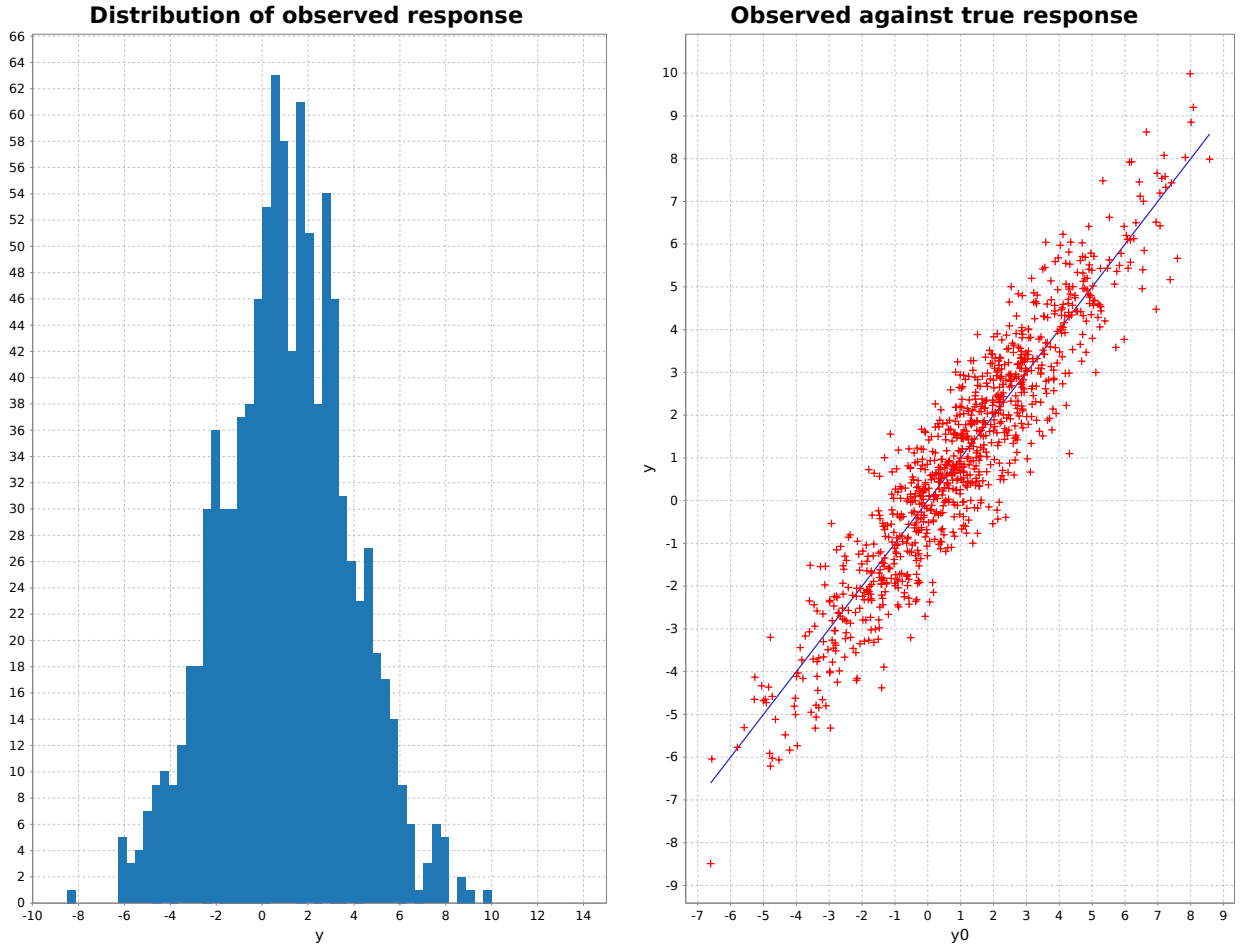


Figure 4.1. A Breeze–viz Figure with two subplots: a histogram and a scatter–plot

```
import breeze.numerics._
//import breeze.numerics._
val fig2 = Figure("More plots")
//fig2: breeze.plot.Figure = breeze.plot.Figure@3eabb64
fig2.width=800
//fig2.width: Int = 800
fig2.height=600
//fig2.height: Int = 600
val p3 = fig2.subplot(1,2,0)
//p3: breeze.plot.Plot = breeze.plot.Plot@32fd55a6
p3 += image(X)
//res12: breeze.plot.Plot = breeze.plot.Plot@32fd55a6
p3.xlabel = "p"
//p3.xlabel: String = p
```

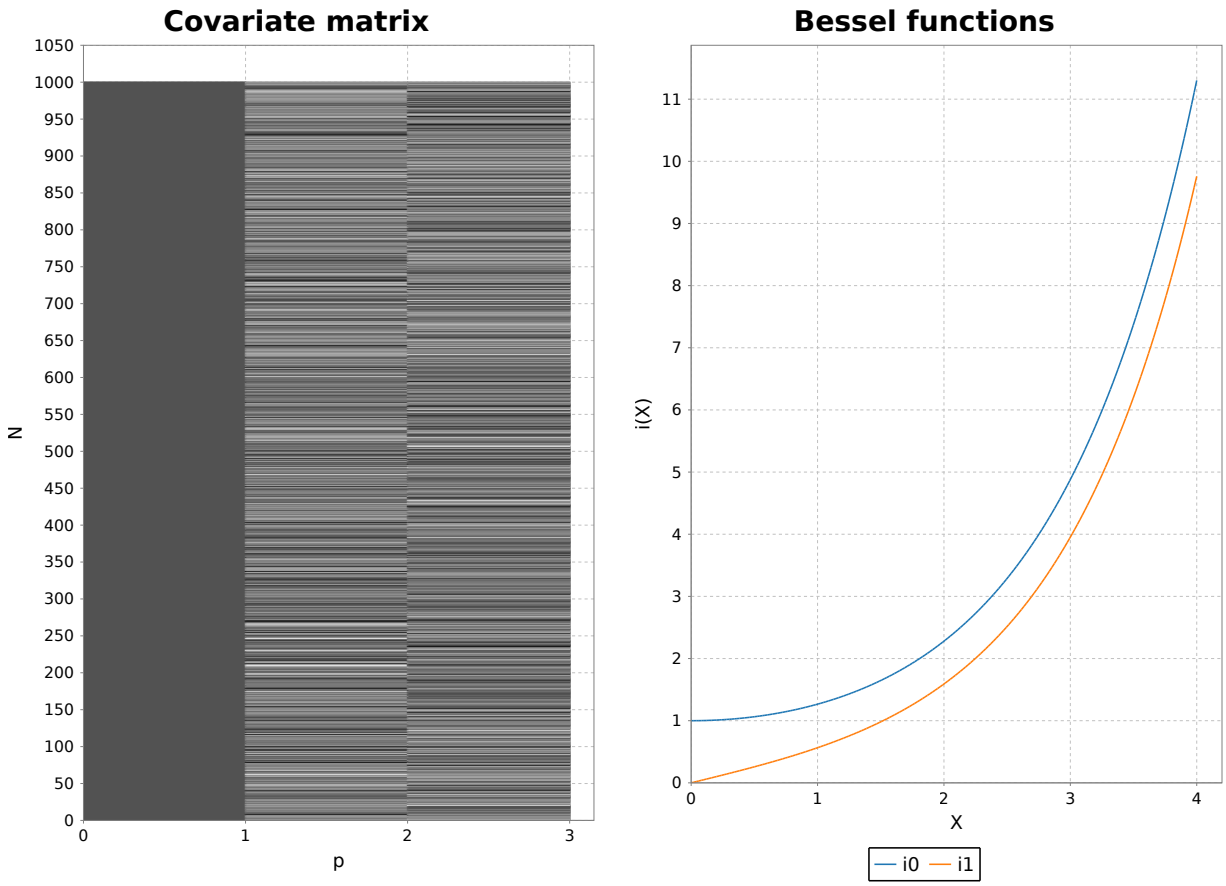


Figure 4.2. A Breeze–viz Figure with two subplots: a matrix image and a line graph

```
p3.ylabel = "N"
//p3.ylabel: String = N
p3.title = "Covariate matrix"
//p3.title: String = Covariate matrix
val p4 = fig2.subplot(1,2,1)
//p4: breeze.plot.Plot = breeze.plot.Plot@4ad7985d
val xs = linspace(0,4,200)
//xs: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 0.020100502512562814, ...
p4 += plot(xs,xs map (Bessel.i0(_:Double)),name=i0")
//res13: breeze.plot.Plot = breeze.plot.Plot@4ad7985d
p4 += plot(xs,xs map (Bessel.i1(_:Double)),name="i1")
//res14: breeze.plot.Plot = breeze.plot.Plot@4ad7985d
p4.legend = true
//p4.legend: Boolean = true
p4.xlabel = "X"
//p4.xlabel: String = X
p4.ylabel = "i(X)"
//p4.ylabel: String = i(X)
p4.title = "Bessel functions"
//p4.title: String = Bessel functions
```

Chapter 5

Monte Carlo simulation

Monte Carlo is a standard technique in statistical computing for avoiding integration problems associated with marginalisation or expectation calculations. Some (pure) Monte Carlo algorithms parallelise well, and others, such as Markov chain Monte Carlo (MCMC), don't. Amenability to parallelisation is closely related to amenability to vectorisation, so MCMC algorithms don't vectorise well, either. This means that MCMC algorithms tend to run slowly in dynamic and interpreted languages. The desire to want to write fast efficient Monte Carlo algorithms is one reason why statisticians might want to consider using a fast, efficient, compiled language like Scala.

5.1 Parallel Monte Carlo

5.1.1 Monte Carlo integration

We begin with a very simple Monte Carlo algorithm. Suppose that we want to construct a Monte Carlo estimate of the integral

$$I = \int_0^1 \exp\{-u^2\}du \tag{5.1}$$

We first note that we can consider this integral as an expectation taken wrt to a $U(0,1)$ random variable, U, on the interval $(0,1)$.

$$I = E(\exp\{-U^2\}) = \int_0^1 \exp\{-u^2\}du \tag{5.2}$$

Then given iid realisations $u_1, u_2, \ldots, u_n$ of U, we can form the Monte Carlo estimate,

$$\hat{I}_n = \frac{1}{n}\sum_{i=1}^{n} \exp\{-u_i^2\} \tag{5.3}$$

Some very simple serial Scala code to implement this algorithm follows.

```
import java.util.concurrent.ThreadLocalRandom
import scala.math.exp
import scala.annotation.tailrec
val N = 1000000L
```

```
def rng = ThreadLocalRandom.current()

def mc(its: Long): Double = {
  @tailrec def sum(its: Long, acc: Double): Double = {
    if (its == 0) acc
    else {
      val u = rng.nextDouble()
      sum(its-1, acc + exp(-u*u))
    }
  }
  sum(its,0.0)/its
}

mc(N)
//res0: Double = 0.7465635439396162
```

ThreadLocalRandom is a parallel random number generator introduced into recent versions of the Java programming language, which can be easily utilised from Scala code. It is the default generator used by Breeze.

Note that this code has no dependency on Breeze. The above code is serial, but fast – comparable with serial code written in C/C++. Now that we have a good working implementation we should think how to parallelise it.

Exercise

5.1.2 Parallel implementation

As we have already seen, the simplest way to introduce parallelisation into Scala code is to parallelise a **map** over a collection. We therefore need a collection and a **map** to apply to it. Here we will just divide our **N** iterations into (say) 4 separate computations, and use a **map** to compute the required Monte Carlo sums.

```
def mcp(its: Long,np: Int = 4): Double =
  (1 to np).par.map(i => mc(its/np)).sum/np
mcp(N)
//res1: Double = 0.7469400102523494
```

Running this new code confirms that it works and gives similar estimates for the Monte Carlo integral as the previous version. It's also faster, though we will look at timings shortly. The function **par** converts the collection (here a **Range**) to a parallel collection, and then subsequent **map**s, **filter**s, etc., can be computed in parallel on appropriate multicore architectures.

5.1.3 Timings, and varying the size of the parallel collection

We can define a function to do a simple timing with

```
def time[A](f: => A) = {
  val s = System.nanoTime
  val ret = f
  println("time: "+(System.nanoTime-s)/1e6+"ms")
```

```
  ret
}
//time: [A](f: => A)A
```

See page 209 for **f: => A**.

which we can test as follows.

```
val bigN = 100000000L
//bigN: Long = 100000000
time(mc(bigN))
//time: 5640.346873ms
//res3: Double = 0.7468043237063586
time(mcp(bigN))
//time: 2281.011243ms
//res4: Double = 0.7468051965668517
```

So this works, and confirms that the parallel version appears to be around 3 times faster than the serial version on my laptop. Now benchmarking code on the JVM is non-trivial, for various reasons. We will look at this topic later if time permits. For now, we can now run this code with varying sizes of **np**, and do each test 3 times, in order to see how the runtime of the code changes as the size

```
(1 to 12).foreach{i =>
  println("np = "+i)
  (1 to 3).foreach(j => time(mcp(bigN,i)))
}
//np = 1
//time: 5578.791748ms
//time: 5526.368988ms
//time: 5517.75116ms
//np = 2
/time: 2910.025414ms
//time: 2923.361272ms
//time: 2910.761801ms
//np = 3
//time: 2093.758369ms
//time: 2115.365902ms
//time: 2054.741932ms
//np = 4
//time: 2236.159519ms
//time: 2254.091659ms
//time: 2217.866655ms
//np = 5
//time: 2720.698771ms
//time: 2405.750295ms
//time: 2376.458361ms
//np = 6
//time: 2280.923904ms
```

```
//time: 2362.903402ms
//time: 2985.779844ms
//np = 7
//time: 2266.469841ms
//time: 2390.492396ms
//time: 2807.634278ms
//np = 8
//time: 2270.302995ms
//time: 2141.775647ms
//time: 2304.89651ms
//np = 9
//time: 3212.22879ms
//time: 2499.237914ms
//time: 2233.026895ms
//np = 10
//time: 2241.239172ms
//time: 2372.663931ms
//time: 2402.156959ms
//np = 11
//time: 2907.986425ms
//time: 2260.827075ms
//time: 2254.387181ms
//np = 12
//time: 3101.684684ms
//time: 2968.021836ms
//time: 2554.808189ms
```

So we see that the timings decrease steadily until the size of the parallel collection hits 4 (my laptop is a Quad-core), and then increases very slightly, but not much as the size of the collection increases, with another sweet-spot at 8 (the number of processors my hyper-threaded Quadcore presents to Linux). The Scala compiler and JVM runtime manage an appropriate number of threads for the collection irrespective of the actual size of the collection.

5.1.4 Rejection sampling

Rejection sampling is another technique that can be used for Monte Carlo integration. The implementation issues are very similar to the Monte Carlo integration algorithms we have just considered. There is a complete example of a rejection sampling algorithm in the course repo examples directory. This example uses Breeze, for illustrative purposes.

5.2 Markov chain Monte Carlo

Pure Monte Carlo is concerned with generating iid samples from a distribution of interest. Often, and especially for high dimensional distributions, this can be difficult. It turns out that it is often easier to generate a Markov chain with an equilibrium distribution corresponding to a distribution of interest. In applications of Bayesian statistics, this distribution is the posterior distribution, but MCMC has other applications, including statistical physics, from where the techniques originate.

```
import scala.annotation.tailrec
import breeze.linalg._
import breeze.stats.distributions._
```

5.2.1 Metropolis-Hastings algorithms

We will begin our investigation of MCMC methods by considering a very simple one-dimensional problem. This will allow us to focus on the structure of the problem and the code, and not get lost in the technical details of building complex samplers.

Note for Metropolis-Hastings algorithms:

- A basic assumption is that in principle any state can be visited in the following algorithm in principle; that is, the system is assumed to be ergodic.
- Generate a next state of x' from x with the generation probability (a proposal distribution), $p^0(x \to x')$.
- The new state is accepted, with the probability of acceptance, which is taken to be $\min\{1, p(x')p^0(x' \to x)/(p(x)p^0(x \to x'))\}$. where $p(x)(\propto \exp\{-E(x)\})$ denotes the probability or probability density of the states x.
- The transition probability from the state x to x' in this Markov process is

$$p(x \to x') = p^0(x \to x')) \min\{1, p(x')p^0(x' \to x)/(p(x)p^0(x \to x'))\} \tag{5.4}$$

  Then, the ratio of the transition probabilities between the forward and backward becomes

$$p(x \to x')/p(x' \to x) = p^0(x \to x')/p^0(x' \to x)[p(x')p^0(x' \to x)/(p(x)p^0(x \to x')) \tag{5.5}$$

$$= p(x')/(p(x) \tag{5.6}$$

- Thus, the following detailed balance equation is satisfied.

$$p(x)p(x \to x') = p(x')p(x' \to x) \tag{5.7}$$

  Therefore, the equilibrium distribution is equal to $p(x)$. If $p^0(x \to x') =$ constant, this algorithm is called as the Metropolis method.

A simple MH sampler

I will just consider a trivial toy Metropolis algorithm using a Uniform random walk proposal to target a standard normal distribution. So, if the current state of a Markov chain is x, a new candidate $x' = x + u$ is formed, where $u \sim U(-\epsilon, \epsilon)$, for some fixed tuning parameter ϵ. This candidate should be accepted as the new state of the chain with probability $\min\{1, A\}$, where $A = \phi(x')/\phi(x)$, and

A fairly direct translation of this algorithm into Scala (using Breeze) is:

```
def metrop1(n: Int = 1000, eps: Double = 0.5):
  DenseVector[Double] = {
    val vec = DenseVector.fill(n)(0.0)
    var x = 0.0
    var oldll = Gaussian(0.0, 1.0).logPdf(x)
```

```
    vec(0) = x
    (1 until n).foreach { i =>
      val can = x + Uniform(-eps, eps).draw
      val loglik = Gaussian(0.0, 1.0).logPdf(can)
      val loga = loglik - oldll
      if (math.log(Uniform(0.0, 1.0).draw) < loga) {
        x = can
        oldll = loglik
      }
      vec(i) = x
    }
    vec
  }
//metrop1: (n: Int, eps: Double)breeze.linalg.DenseVector[Double]
```

This code works, and is reasonably fast and efficient, but there are several issues with it from a functional programmers perspective. One issue is that we have committed to storing all MCMC output in RAM in a **DenseVector**. This probably isn't an issue here, but for some big problems we might prefer to not store the full set of states, but to just print the states to (say) the console, for possible re-direction to a file. It is easy enough to modify the code to do this:

```
def metrop2(n: Int = 1000, eps: Double = 0.5): Unit =
{
  var x = 0.0
  var oldll = Gaussian(0.0, 1.0).logPdf(x)
  (1 to n).foreach { i =>
    val can = x + Uniform(-eps, eps).draw
    val loglik = Gaussian(0.0, 1.0).logPdf(can)
    val loga = loglik - oldll
    if (math.log(Uniform(0.0, 1.0).draw) < loga) {
      x = can
      oldll = loglik
    }
    println(x)
 }
}
//metrop2: (n: Int, eps: Double)Unit
```

But now we have two versions of the algorithm. One for storing results locally, and one for streaming results to the console. This is clearly unsatisfactory, but we shall return to this issue shortly. Another issue that will jump out at functional programmers is the reliance on mutable variables for storing the state and old likelihood. Let's fix that now by re-writing the algorithm as a tail-recursion.

```
import scala.annotation.tailrec
@annotation.tailrec
def metrop3(n: Int = 1000, eps: Double = 0.5,
  x: Double = 0.0, oldll: Double = Double.MinValue):
  Unit = {
```

```
    if (n > 0) {
      println(x)
      val can = x + Uniform(-eps, eps).draw
      val loglik = Gaussian(0.0, 1.0).logPdf(can)
      val loga = loglik - oldll
      if (math.log(Uniform(0.0, 1.0).draw) < loga)
        metrop3(n - 1, eps, can, loglik)
      else
        metrop3(n - 1, eps, x, oldll)
    }
  }
//metrop3: (n: Int, eps: Double, x: Double, oldll: Double)Unit
```

This has eliminated the vars, and is just as fast and efficient as the previous version of the code. However, this is for the print-to-console version of the code. What if we actually want to keep the iterations in RAM for subsequent analysis? We can keep the values in an accumulator, as follows.

```
@tailrec
def metrop4(n: Int = 1000, eps: Double = 0.5,
  x: Double = 0.0, oldll: Double = Double.MinValue,
  acc: List[Double] = Nil): DenseVector[Double] = {
    if (n == 0)
      DenseVector(acc.reverse.toArray)
    else {
      val can = x + Uniform(-eps, eps).draw
      val loglik = Gaussian(0.0, 1.0).logPdf(can)
      val loga = loglik - oldll
      if (math.log(Uniform(0.0, 1.0).draw) < loga)
        metrop4(n - 1, eps, can, loglik, can :: acc)
      else
        metrop4(n - 1, eps, x, oldll, x :: acc)
    }
  }
//metrop4: (n: Int, eps: Double, x: Double, oldll: Double, acc: List[Double])breeze.
```

5.2.2 Factoring out the updating logic

This is all fine, but we haven't yet addressed the issue of having different versions of the code depending on what we want to do with the output. The problem is that we have tied up the logic of advancing the Markov chain with what to do with the output. What we need to do is separate out the code for advancing the state. We can do this by defining a new function.

```
def newState(x: Double, oldll: Double, eps: Double):
  (Double, Double) = {
    val can = x + Uniform(-eps, eps).draw
    val loglik = Gaussian(0.0, 1.0).logPdf(can)
    val loga = loglik - oldll
```

```
    if (math.log(Uniform(0.0, 1.0).draw) < loga)
      (can, loglik)
    else
      (x, oldll)
}
//newState: (x: Double, oldll: Double, eps: Double)(Double, Double)
```

This function takes as input a current state and associated log likelihood and returns a new state and log likelihood following the execution of one step of a MH algorithm. This separates the concern of state updating from the rest of the code. So now if we want to write code that prints the state, we can write it as

```
@tailrec
def metrop5(n: Int = 1000, eps: Double = 0.5,
  x: Double = 0.0,
  oldll: Double = Double.MinValue): Unit = {
    if (n > 0) {
      println(x)
      val ns = newState(x, oldll, eps)
      metrop5(n - 1, eps, ns._1, ns._2)
    }
  }
//metrop5: (n: Int, eps: Double, x: Double, oldll: Double)Unit
```

Note the use of **._1** and **._2** to access the first and second elements of a tuple. Referring to elements of a tuple by position can sometimes lead to code that is difficult to read. Often it is better to unpack the tuple, as follows.

```
@tailrec
def metrop5b(n: Int = 1000, eps: Double = 0.5,
  x: Double = 0.0,
  oldll: Double = Double.MinValue): Unit = {
    if (n > 0) {
      println(x)
      val (nx, ll) = newState(x, oldll, eps)
      metrop5b(n - 1, eps, nx, ll)
    }
  }
//metrop5b: (n: Int, eps: Double, x: Double, oldll: Double)Unit
```

Now, if instead of printing to console, we want to accumulate the set of states visited, we can write that as

```
@tailrec
def metrop6(n: Int = 1000, eps: Double = 0.5,
  x: Double = 0.0, oldll: Double = Double.MinValue,
  acc: List[Double] = Nil): DenseVector[Double] = {
    if (n == 0)
      DenseVector(acc.reverse.toArray)
    else {
      val (nx, ll) = newState(x, oldll, eps)
```

```
      metrop6(n - 1, eps, nx, ll, nx :: acc)
    }
  }
//metrop6: (n: Int, eps: Double, x: Double, oldll: Double, acc: List[Double])breeze.
```

Both of these functions call **newState** to do the real work, and concentrate on what to do with the sequence of states. However, both of these functions repeat the logic of how to iterate over the sequence of states.

MCMC as a stream

Ideally we would like to abstract out the details of how to do state iteration from the code as well. We can do this using Scala's **Stream**, which we saw earlier can embody the logic of how to perform state iteration, allowing us to abstract those details away from our code. To do this, we will restructure our code slightly so that it more clearly maps old state to new state.

```
def nextState(eps: Double)(state: (Double, Double)):
  (Double, Double) = {
    val (x, oldll) = state
    val can = x + Uniform(-eps, eps).draw
    val loglik = Gaussian(0.0, 1.0).logPdf(can)
    val loga = loglik - oldll
    if (math.log(Uniform(0.0, 1.0).draw) < loga)
      (can, loglik)
    else
      (x, oldll)
  }
//nextState: (eps: Double)(state: (Double, Double))(Double, Double)
```

The "real" state of the chain is just **x**, but if we want to avoid recalculation of the old likelihood, then we need to make this part of the chain's state. We can use this nextState function in order to construct a **Stream**.

```
def metrop7(eps: Double = 0.5, x: Double = 0.0,
  oldll: Double = Double.MinValue): Stream[Double] =
    Stream.iterate((x,oldll))(nextState(eps)) map (_._1)
//metrop7: (eps: Double, x: Double, oldll: Double)Stream[Double]
```

The result of calling this is an infinite stream of states. Obviously it isn't computed – that would require infinite computation, but it captures the logic of iteration and computation in a **Stream**, that can be thought of as a **lazy List**. We can get values out by converting the **Stream** to a regular collection, being careful to truncate the **Stream** to one of finite length beforehand! eg. **metrop7().drop(1000).take(10000).toArray** will do a burn-in of 1,000 iterations followed by a main monitoring run of length 10,000, capturing the results in an **Array**. Note that **metrop7().drop(1000).take(10000)** is a **Stream**, and so the main monitoring run of length 10k is not actually computed until the toArray is encountered. Conversely, if printing to console is required, just replace the **.toArray** with **.foreach(println)**.

The above stream-based approach to MCMC iteration is clean and elegant, and deals nicely with issues like burn-in and thinning (which can be handled similarly). This is how I typically write MCMC codes these days. However, functional programming purists would still have issues with this approach, as it isn't quite pure functional. The problem is that the code isn't pure – it has a side-effect, which is to mutate the state of the under-pinning pseudo-random number generator. If the code was pure, calling nextState with the same

inputs would always give the same result. Clearly this isn't the case here, as we have specifically designed the function to be stochastic, returning a randomly sampled value from the desired probability distribution. So **nextState** represents a function for randomly sampling from a conditional probability distribution.

A pure functional approach

Now, ultimately all code has side-effects, or there would be no point in running it! But in functional programming the desire is to make as much of the code as possible pure, and to push side-effects to the very edges of the code. So it's fine to have side-effects in your main method, but not buried deep in your code. Here the side-effect is at the very heart of the code, which is why it is potentially an issue.

To keep things as simple as possible, at this point we will stop worrying about carrying forward the old likelihood, and hard-code a value of **eps**. Generalisation is straightforward. We can make our code pure by instead defining a function which represents the conditional probability distribution itself. For this we use a *probability monad*, which in Breeze is called **breeze.stats.distributions.Rand**. We can couple together such functions using monadic binds (**flatMap** in Scala), expressed most neatly using for-comprehensions. So we can write our transition kernel in the form of a simple *probabilistic program* as

```
def kernel(x: Double): Rand[Double] = for {
  innov <- Uniform(-0.5, 0.5)
  can = x + innov
  oldll = Gaussian(0.0, 1.0).logPdf(x)
  loglik = Gaussian(0.0, 1.0).logPdf(can)
  loga = loglik - oldll
  u <- Uniform(0.0, 1.0)
} yield if (math.log(u) < loga) can else x
//kernel: (x: Double)breeze.stats.distributions.Rand[Double]
```

This is now pure – the same input x will always return the same probability distribution – the conditional distribution of the next state given the current state, and no RNG state will be mutated. We can draw random samples from this distribution if we must, but it's probably better to work as long as possible with pure functions. So next we need to encapsulate the iteration logic. Breeze has a **breeze.stats.distributions.MarkovChain** object which can take kernels of this form and return a stochastic **breeze.stats.distributions.Process** object representing the iteration logic, as follows.

```
//def apply[T](init: T)(resample: (T)⇒=> Rand[T]): Process[T]
//    Given an initial state and an arbitrary Markov transition, return a sampler fo
MarkovChain(0.0)(kernel).
  steps.
  drop(1000).
  take(10000).
  foreach(println)
//0.9492705429060206
//0.9377359076633873
//...
MarkovChain(0.0)(kernel).
  steps.
  drop(1000).
  take(10).
```

```
  foreach(println(_))
//-0.27090069321235166
//-0.6871619185012461
//...
```

The **steps** method contains the logic of how to advance the state of the chain. But again note that no computation actually takes place until it is triggered, here by **.drop** – this is when the sampling starts and sideeffects happen.

Metropolis-Hastings is a common use-case for Markov chains, so Breeze actually has a helper method built-in that will construct a MH sampler directly from an initial state, a proposal kernel, and a (log) target.

```
 //def metropolisHastings[T](init: T, proposal: (T) => ContinuousDistr[T]
 //    ) (logMeasure: (T) ⇒ Double): Process[T]
 //    Performs Metropolis-Hastings distributions on a random variable.
 //init the initial parameter
 //proposal the proposal distribution generator
 //logMeasure the distribution we want to sample from
 //
MarkovChain.
 metropolisHastings(0.0,
  (x: Double) => Uniform(x - 0.5, x + 0.5) ) (
  x => Gaussian(0.0, 1.0).logPdf(x)).
  steps.
  drop(1000).
  take(10000).
  toArray
//res0: Array[Double] = Array(-0.11821725720711274, -0.5479778998637836,...
```

Summary

Viewing MCMC algorithms as infinite streams of state is useful for writing elegant, generic, flexible code. Streams occur everywhere in programming, and so there are lots of libraries for working with them. Here I used the simple **Stream** from the Scala standard library, but there are much more powerful and flexible stream libraries for Scala, including **fs2** and **akka-streams**. But whatever libraries you are using, the fundamental concepts are the same. The most straightforward approach to implementation is to define impure stochastic streams to consume. However, a pure functional approach is also possible, and the Breeze library defines some useful functions to facilitate this.

5.2.3 A Gibbs sampler

Gibbs sampling is another commonly used MCMC procedure. Gibbs sampling samples from a multivariate distribution by sequentially sampling from full-conditional distributions. Like MH, Gibbs sampling is typically used in statistics in order to sample from a Bayesian posterior distribution, but it doesn't have to be. Here we'll illustrate the ideas with a very simple bivariate distribution. The target distribution has density

$$f(x,y) \quad = \quad kx^2 \exp\{-xy^2 - y^2 + 2y - 4x\}, \quad x > 0, y \in \mathcal{R}, \tag{5.8}$$

with unknown normalising constant $k > 0$, ensuring that the density integrates to one. The two full-conditionals for this distribution can be calculated using elementary algebra, and are given as follows using

the parametrisation of normal and gamma distributions most commonly adopted by statisticians.

$$x|y \sim \Gamma(3, y^2 + 4) \tag{5.9}$$

$$y|x \sim N(\frac{1}{1+x}, \frac{1}{2(1+x)}) \tag{5.10}$$

The second parameter of the normal represents variance, and the expectation of a $\Gamma(\alpha, \beta)$ is α/β. In the **breeze.stats.distributions**, the second argument for **Gaussian** is the standard deviation rather than the variance, and the second argument for **Gamma** is the scale ($\theta = 1/\beta$) rather than the rate (β).

Exercise Prove Eq. 5.10.

One iteration of a Gibbs sampler consists of iterating through the sampling of each full–conditional in turn, using the latest available value of each component.

The state of our Gibbs sampler will be the tuple (x, y). We could implement our sampler using a Scala tuple **(x, y)**, but this is another good opportunity to use Scala *case classes*. We will use a case class to represent the state of our Gibbs sampler.

```
case class State(x: Double, y: Double)
//defined class State
```

Case classes are a convenient way to build data structures, and are used extensively in Scala code. See page 181.

So our case class has two attributes, **x** and **y**, which are both **Double**s. We can create, access, copy, and update objects of our case class as follows:

```
val s = State(1.0,2.0)
//s: State = State(1.0,2.0)
s.x
//res0: Double = 1.0
s.y
//res1: Double = 2.0
s.copy()
//res2: State = State(1.0,2.0)
s.copy(y=3)
//res3: State = State(1.0,3.0)
```

Note that the final example shows how to create a copy of an existing state with just one attribute updated. This can be useful for constructing Gibbs samplers, though we won't actually use it in this very simple example. Since the Gibbs sampler is an MCMC algorithm, the iteration pattern is exactly the same as for the MH algorithm. We will begin by implementing an impure stochastic stream approach. First we need a function to execute one iteration of the Gibbs sampler.

```
import breeze.stats.distributions._
//import breeze.stats.distributions._
def nextState(state: State): State = {
  val sy = state.y
  val x = Gamma(3.0,1.0/(sy*sy+4)).draw
```

```
  val y = Gaussian(1.0/(x+1),1.0/math.sqrt(2*x+2)).draw
  State(x,y)
}
//nextState: (state: State)State
```

Note how we have translated the parameters of the full-conditionals to the form that Breeze expects. Once we have this state updating function, creating a **Stream** for our Markov chain is trivial.

```
//def iterate[A](start: A)(f: (A) => A): Stream[A]
//    An infinite stream that repeatedly applies a given function to a start value.
val gs = Stream.iterate(State(1.0,1.0))(nextState)
//gs: scala.collection.immutable.Stream[State] = Stream(State(1.0,1.0), ?)
val output = gs.drop(1000).take(100000).toArray
//output: Array[State] = Array(State(0.5208657591856419,0.7259475091639989), ...
```

If we want to look at the output, we can do it with code such as the following.

```
import breeze.linalg._
//import breeze.linalg._
val xv = DenseVector(output map (_.x))
//xv: breeze.linalg.DenseVector[Double] = DenseVector(0.5208657591856419, ...
val yv = DenseVector(output map (_.y))
//yv: breeze.linalg.DenseVector[Double] = DenseVector(0.7259475091639989, ...

import breeze.plot._
//import breeze.plot._
val fig = Figure("Bivariate Gibbs sampler")
//fig: breeze.plot.Figure = breeze.plot.Figure@384b3fe7
fig.subplot(2,2,0)+=hist(xv,50)
//res4: breeze.plot.Plot = breeze.plot.Plot@22ea09b7
fig.subplot(2,2,1)+=hist(yv,50)
//res5: breeze.plot.Plot = breeze.plot.Plot@249fa2cb
fig.subplot(2,2,2)+=plot(xv,yv, '.')
//res6: breeze.plot.Plot = breeze.plot.Plot@701c092c
```

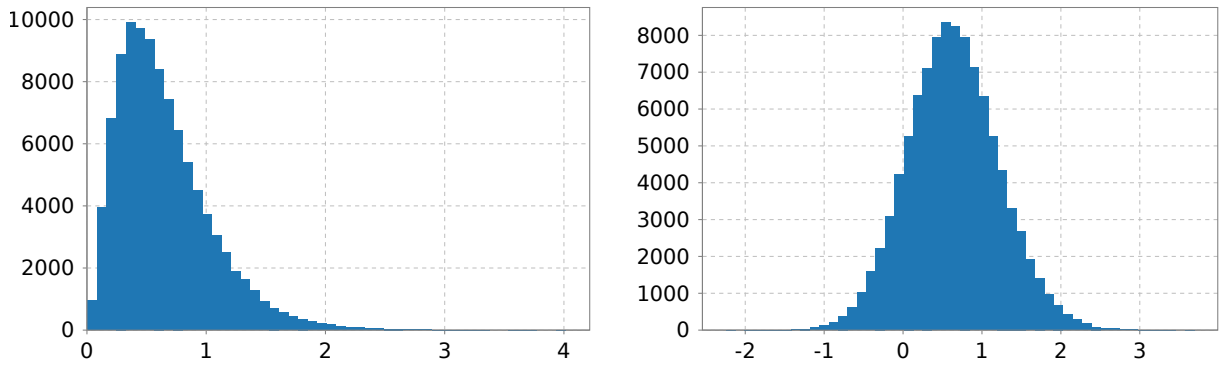


Figure 5.1

Thinning MCMC streams

For Markov chains with strong dependence, as well as truncating a burnin period (using **drop**), we may also want to *thin* the output, keeping just one in every n iterations, for some suitably chosen $n > 1$. Typical values might be $n = 10$ or $n = 100$.

This is done primarily to reduce storage requirements associated with very long MCMC runs. Unfortunately there is no **thin** method pre-defined on the Scala **Stream**, but we can define our own function for thinning **Stream** as

```
def thin[T](s: Stream[T], th: Int): Stream[T] = {
  val ss = s.drop(th - 1)
  if (ss.isEmpty)
    Stream.empty
  else
    ss.head #:: thin(ss.tail, th)
}
//thin: [T](s: Stream[T], th: Int)Stream[T]
```

which we could use as

```
thin(gs.drop(1000),10).take(10000).toArray
//res1: Array[State] = Array(State(0.5990467987686352,...
```

This will give us a final sample of size 10,000 following a burn-in of 1,000 iterations and a thinning factor of 10. Ideally we might like to define our own method **thin** on the existing Stream class. In fact this is perfectly possible to do in Scala, and would allow us to write code like

```
// gs.drop(1000).thin(10).take(10000)
```

which would be more consistent. However, this requires advanced features of the Scala language (*type classes* and *implicits*) that we are not yet ready to discuss, but we will see how to implement exactly this in the final chapter.

A monadic approach

A pure functional monadic approach to defining a Gibbs sampler is not much more effort, as the code below illustrates.

```
def kernel(state: State): Rand[State] = for {
  x <- Gamma(3.0,1.0/(state.y*state.y+4))
  y <- Gaussian(1.0/(x+1),1.0/math.sqrt(2*x+2))
  ns = State(x,y)
} yield ns
//kernel: (state: State)breeze.stats.distributions.Rand[State]

val out3 = MarkovChain(State(1.0,1.0))(kernel).
  steps.
  drop(1000).
  take(10000).
  toArray
//out3: Array[State] = Array(State(1.8740285072385334, ...
```

It is really a matter of taste as to which of these approaches is to be preferred. Also note that the two approaches are not mutually exclusive. For example, starting from the pure monadic approach, one can take a Breeze **Process** object defined, say, by a **MarkovChain** construction, and act on it with **.steps.toStream** to turn the pure **Process** into an impure stochastic **Stream**.

Chapter 6

Statistical modelling

6.1 Linear regression

6.1.1 Introduction

Statistical modelling is a huge topic and we certainly don't have time to discuss it in detail in the context of this course. But *linear modelling*, using *multiple linear regression* is almost certainly the most widely used statistical modelling technique in practice. So it makes sense to start our investigation of statistical modelling with linear regression. We assume n observations of a univariate response $y_i, i = 1, 2, \ldots, n$ with associated covariates x_i, where each x_i is a p-vector. We model y_i as a linear combination of the x_i; $n > p$. That is, we want to find a p-vector β so that $x_i\beta$ is close to y_i, $\forall i$. To avoid explicit consideration of an intercept, we assume that the first element of each covariate vector is 1. Stacking the observations and covariates, we have an n-vector y and an associated $n \times p$ matrix X of covariate (where the ith row of X corresponds to x_i^T) with first column consisting of 1s, and our model is

$$y \sim X\beta \tag{6.1}$$

In other words, we want to find the $\hat{\beta}$ which minimises $\|y - X\hat{\beta}\|_2^2$ The solution of this *least squares problem* is given by the solution of the *normal equations*

$$X^T X\hat{\beta} = X^T y \tag{6.2}$$

Mathematically, we can write this solution as $\hat{\beta} = (X^T X)^{-1} X^T y$ but in numerical linear algebra it is usually better to solve a linear system directly rather than compute a matrix inverse explicitly.

There are many ways to solve linear systems. The normal equations are typically solved using a thin QR-factorisation of the covariate matrix. So if $X = QR$, where Q is $n \times p$ such that $Q^T Q = I$ and R is $p \times p$ upper triangular, then the normal equations simplify to

$$R\hat{\beta} = Q^T y \tag{6.3}$$

and this can be solved for b with a single back-solve.

> **Note:** Probabilistic interpretation for the LMS(Least Mean Square) cost/loss function, $L \equiv \sum_i (y_i - h_\beta(X)_i)^2$, where the predicted value h is defined here to be $h_\beta \equiv X\beta$
>
> Assume that the difference between the observed value and the predicted value, $\epsilon_i \equiv y_i - X_i\hat{\beta}$,

obeys a Gaussian distribution.

$$y \equiv h(X,\beta) + \epsilon \tag{6.4}$$
$$\epsilon = \mathcal{N}(0,\sigma^2) \tag{6.5}$$

Then, the log-likelihood is equal to

$$\mathcal{L}(\beta|X) = -\frac{1}{2}\sum_i (y_i - h_\beta(X)_i)^2 + \text{constant} \tag{6.6}$$
$$= -L(\beta) + \text{constant} \tag{6.7}$$

Thus maximizing the log-likelihood corresponds to minimizing the LMS.

Note: LMS update rule in GD (gradient descent) algorithm to minimize a loss function in machine learning

$$\frac{\partial L(\beta)}{\partial \beta_\mu} = \sum_i (y_i - \sum_\nu X_{i\nu}\beta_\nu)X_{i\mu} \tag{6.8}$$
$$\beta^{(t+1)} = \beta^{(t)} + \alpha\sum_i (y_i - \sum_\nu X_{i\nu}\beta_\nu)X_{i\mu} \tag{6.9}$$

Note

$$(Q_1R_1)^*(Q_1R_1)\hat{\beta} = (Q_1R_1)^* y \tag{6.10}$$
$$(R_1)^*R_1\hat{\beta} = (R_1)^*(Q_1)^* y \tag{6.11}$$
$$\hat{\beta} = [(R_1)^*R_1]^{-1}(R_1)^*(Q_1)^* y \tag{6.12}$$
$$\hat{\beta} = (R_1)^{-1}(Q_1)^* y \tag{6.13}$$
$$R_1\hat{\beta} = (Q_1)^* y \tag{6.14}$$

where $(Q_1)^* = (\overline{Q_1})^T$, the transverse of the complex conjugate of Q_1. Notice that $(Q_1)^*Q_1 = I$ and R_1 is invertible. Then R_1 is equal to the upper trianglar matrix of the Cholesky decomposition of X^*X.

$$X^*X = R_1^*R_1 \tag{6.15}$$

6.1.2 Linear regression in Scala

Let's start by creating a small toy dataset.

```
import breeze.linalg._
//import breeze.linalg._
val y = DenseVector(1.0,2.0,3.0,2.0,1.0)
//y: breeze.linalg.DenseVector[Double] = DenseVector(1.0, 2.0, 3.0, 2.0, 1.0)
val Xwoi = DenseMatrix((2.1,1.4),(1.5,2.2),
                (3.0,3.1),(2.5,2.2),(2.0,1.0))
//Xwoi: breeze.linalg.DenseMatrix[Double] =
```

```
//2.1  1.4
//1.5  2.2
//3.0  3.1
//2.5  2.2
//2.0  1.0
val X = DenseMatrix.horzcat(DenseVector.ones[Double](
             Xwoi.rows).toDenseMatrix.t, Xwoi)
//X: breeze.linalg.DenseMatrix[Double] =
//1.0  2.1  1.4
//1.0  1.5  2.2
//1.0  3.0  3.1
//1.0  2.5  2.2
//1.0  2.0  1.0
```

Note how the final line demonstrates how to prepend a column of ones onto a covariate matrix.

There is a simple least squares function included in Breeze which can be used to solve this problem.

```
import breeze.stats.regression._
val mod = leastSquares(X,y)
//mod: breeze.stats.regression.LeastSquaresRegressionResult = <function1>
mod.coefficients
//res0: breeze.linalg.DenseVector[Double] = DenseVector(-0.2804082840767408, 0.05363
mod.rSquared
//res1: Double = 0.0851907328842681
```

This solves the problem, but doesn't really offer much over a more direct solution:

```
X \ y
//res2: breeze.linalg.DenseVector[Double] = DenseVector(-0.2804082840767408, 0.05363
```

because as previously discussed, the Breeze linear solver uses a QR- based solver by default for over-determined systems.

It is useful to understand exactly how a QR-based solution works, as we can then build on it. First we need to define a **backSolve** function, since one is not exported by the Breeze API.

```
import com.github.fommil.netlib.BLAS.{ getInstance => blas }
//import com.github.fommil.netlib.BLAS.{getInstance=>blas}
def backSolve(A: DenseMatrix[Double],
  y: DenseVector[Double]): DenseVector[Double] = {
    val yc = y.copy
    blas.dtrsv("U", "N", "N", A.cols, A.toArray,
      A.rows, yc.data, 1)
    yc
}
//backSolve: (A: breeze.linalg.DenseMatrix[Double], y: breeze.linalg.DenseVector[Dou
```

We can now use this with a QR-factorisation (QR-decomposition) of the model matrix to solve the least squares problem.

```
val QR = qr.reduced(X)
//QR: breeze.linalg.qr.DenseQR =
//QR(-0.44721359549995787  -0.10656672499880869  0.36158387049672475
//-0.44721359549995787  -0.6394003499928489   -0.6206396829776012
//-0.44721359549995787  0.6926837124922532    -0.35194956342098893
//-0.44721359549995787  0.24865569166388585   0.010942669765032248
//-0.44721359549995787  -0.1953723291644815   0.6000627061368331    ,
//-2.23606797749979  -4.964070910049532  -4.427414595449582
//0.0                1.12605506082074    0.9431155162394529
//0.0                0.0                 -1.32609695084047   )
val q = QR.q
//q: breeze.linalg.DenseMatrix[Double] =
//-0.44721359549995787  -0.10656672499880869  0.36158387049672475
//-0.44721359549995787  -0.6394003499928489   -0.6206396829776012
//-0.44721359549995787  0.6926837124922532    -0.35194956342098893
//-0.44721359549995787  0.24865569166388585   0.010942669765032248
//-0.44721359549995787  -0.1953723291644815   0.6000627061368331
val r = QR.r
//r: breeze.linalg.DenseMatrix[Double] =
//-2.23606797749979  -4.964070910049532  -4.427414595449582
//0.0                1.12605506082074    0.9431155162394529
//0.0                0.0                 -1.32609695084047
backSolve(r, q.t * y)
//res0: breeze.linalg.DenseVector[Double] = DenseVector(-0.2804082840767405, 0.05363
```

If we are only interested in regression coefficients, then we are done. But in practice regression modelling involves diagnostics, such as indicators of the relative importance of different covariates, etc. For people used to regression modelling in R, such diagnostics are essential. It's easy enough to write a function to provide such diagnostics, as required. We will try and make the code reasonably efficient by using a QR-based approach.

```
case class Lm(y: DenseVector[Double],
  X: DenseMatrix[Double],names: List[String]) {
  require(y.size == X.rows)               // scala.Predef.require
  require(names.length == X.cols)
  require(X.rows >= X.cols)
  require(X.rows >= X.cols)
  val QR = qr.reduced(X)
  val q = QR.q
  val r = QR.r
  val qty = q.t * y
  val coefficients = backSolve(r,qty)
  import breeze.stats._
  import breeze.stats.distributions._
  //import org.apache.commons.math3.special.Beta
  def tCDF(t: Double, df: Double): Double = {
    import org.apache.commons.math3.special.Beta
```

```
  val xt = df / (t * t + df)
  1.0 - 0.5 * Beta.regularizedBeta(xt,0.5*df,0.5)
 /*
  val st = new StudentsT(df)
  st.cdf(t.abs)                          // CDF of Student's t distribution; both s
 */
}
def fCDF(x: Double, d1: Double, d2: Double) = {
  import org.apache.commons.math3.special.Beta
  val xt = x * d1 / (x * d1 + d2)
  Beta.regularizedBeta(xt, 0.5 * d1, 0.5 * d2)
 /*
  val f = new FDistribution(d1, d2)
  f.cdf(x)                           // CDF of F distribution
 */
}
lazy val fitted = q * qty
lazy val residuals = y - fitted
lazy val n = X.rows
lazy val pp = X.cols
lazy val df = n - pp                          // df   degree of freedom
lazy val rss = sum(residuals ^:^ 2.0)         // ^:^  elementwise power operator
lazy val rse = math.sqrt(rss / df)
lazy val ri = inv(r)                          // inv  inverse
lazy val xtxi = ri * (ri.t)       // (x^T x)^{-1}
lazy val se = breeze.numerics.sqrt(diag(xtxi))*rse    // diag  Create view of matr
lazy val t = coefficients / se                        // t value of t-test
lazy val p = t.map {1.0 - tCDF(_, df)}.map {_ * 2}    // p-value of t-test
lazy val ybar = mean(y)
lazy val ymyb = y - ybar
lazy val ssy = sum(ymyb ^:^ 2.0)
lazy val rSquared = (ssy - rss) / ssy
lazy val adjRs = 1.0 - ((n-1.0)/(n-pp))*(1-rSquared)
lazy val k = pp-1
lazy val f = (ssy - rss) / k / (rss/df)     // F-statistics
lazy val pf = 1.0 - fCDF(f,k,df)            // p-value for F-test
def summary: Unit = {
  println(
  "Estimate\t S.E.\t t-stat\ttp-value\tVariable")
  println(
  "-----------------------------------------------------------")
  (0 until pp).foreach(i => printf(
    "%8.4f\t%6.3f\t%6.3f\t%6.4f %s\t%s\n",
    coefficients(i),se(i),t(i),p(i),
    if (p(i) < 0.05) "*" else " ", names(i)))
  printf(
```

```
        "\nResidual standard error: %8.4f on %d degrees",
        rse, df)
        printf("of freedom\n")
      printf("Multiple R-squared: %6.4f, ", rSquared)
      printf("Adjusted R-squared: %6.4f\n", adjRs)
      printf("F-statistics: %6.4f on %d and %d DF, ",
        f, k, df)
      printf("p-value: %6.5f\n\n", pf)
    }
  }
  //defined class Lm
```

There are few things worth noting about this code. First, it is a case class, so when we "call" it we will actually get back an *object*. Next, note the use of **require** to make explicit our expectations (*pre-conditions*). Also note the use of **val** versus **lazy val**. Everything up to and including the regression coefficients will be computed eagerly when the object is first constructed, but all of the "optional extras"will be computed lazily if and when they are required. Finally note the **summary** method which will print to the console a textual summary of the regression fit that will look very familiar to R users. Note that all of the numerical output matches up exactly with the corresponding output from R.

We can create a model fit using the default case class constructor:

```
val mod2 = Lm(y, X, List("Intercept", "Var1", "var2"))
//mod2: Lm =
//Lm(DenseVector(1.0, 2.0, 3.0, 2.0, 1.0),
//1.0  2.1  1.4
//1.0  1.5  2.2
//1.0  3.0  3.1
//1.0  2.5  2.2
//1.0  2.0  1.0  ,List(Intercept, Var1, var2))
```

All of the attributes of the model fit can be accessed in the obvious way.

```
mod2.coefficients
//res7: breeze.linalg.DenseVector[Double] = DenseVector(-0.2804082840767405, 0.05363
mod2.p
//res8: breeze.linalg.DenseVector[Double] = DenseVector(0.5711265415017199, 0.833715
mod2.rse
//res9: Double = 0.20638644926965033
mod2.rSquared
//res10: Double = 0.9695747382556187
mod2.adjRs
//res11: Double = 0.9391494765112374
mod2.f
//res12: Double = 31.8674247209942
mod2.pf
//res13: Double = 0.030425261744381427
```

An R-style model fit summary can also be printed to the console.

```
mod2.summary
//Estimate         S.E.    t-stat tp-value        Variable
//-------------------------------------------------------
// -0.2804         0.418  -0.671  0.5711          Intercept
//  0.0536         0.225   0.238  0.8337          Var1
//  0.9906         0.156   6.365  0.0238 *        var2
//
//Residual standard error:   0.2064 on 2 degreesof freedom
//Multiple R-squared: 0.9696, Adj R-squared: 0.9391
//F-statistics: 31.8674 on 2 and 2 DF, p-value: 0.03043
```

Now we understand how it works, we can try it out on a bigger, synthetic data example. First, let's simulate some data from a linear regression model.

```
val N = 1000
//N: Int = 1000
val P = 2
//P: Int = 2
val gau=breeze.stats.distributions.Gaussian(0.0,1.0)
//gau: Gaussian = Gaussian(0.0, 1.0)
val XX = new DenseMatrix(N,P,gau.sample(P*N).toArray) // DenseMatrix: column-major
//XX: breeze.linalg.DenseMatrix[Double] =
//-0.8457812850803169     -0.03606247160833908
//0.03489437269350573     0.5172514585940354
-1.7957955377711499     -0.6168159759776303
0.21782767508344233     0.7937616458286805
//...
val X = DenseMatrix.horzcat(
   DenseVector.ones[Double](N).toDenseMatrix.t,XX)
//X: breeze.linalg.DenseMatrix[Double] =
//1.0  -0.8457812850803169     -0.03606247160833908
//1.0  0.03489437269350573     0.5172514585940354
//1.0  -1.7957955377711499     -0.6168159759776303
//1.0  0.21782767508344233     0.7937616458286805
//...
val b0 = linspace(1.0,2.0,P+1)
//b0: DenseVector[Double] = DenseVector(1.0,1.5,2.0)
val y0 = X * b0
//y0: breeze.linalg.DenseVector[Double] = DenseVector(-0.3407968708371536, ...
val y = y0 + DenseVector(gau.sample(1000).toArray)
//y: breeze.linalg.DenseVector[Double] = DenseVector(-0.5570686724758596, ...
```

Now we have our synthetic data, we can do a "quick-and-dirty" regression fit with

```
val b = X \ y
//b: breeze.linalg.DenseVector[Double] = DenseVector(0.9891677524645601, 1.529095690
```

Alternatively, we can use our regression class to get more information about the fit.

```
val mod3 = Lm(y,X, List("Intercept", "Var1", "Var2"))
//mod3: Lm =
//Lm(DenseVector(-0.5570686724758596, 1.694882020911852, ...
mod3.summary
//Estimate         S.E.    t-stat tp-value        Variable
//-------------------------------------------------------
//  0.9892         0.032  31.244  0.0000 *        Intercept
//  1.5291         0.032  48.369  0.0000 *        Var1
//  2.0407         0.032  62.919  0.0000 *        Var2
//
//Residual standard error:   0.9999 on 997 degreesof freedom
//Multiple R-squared: 0.8639, Adjusted R-squared: 0.8636
//F-statistics: 3163.2684 on 2 and 997 DF, p-value: 0.00000
```

Finally, we can plot some diagnostic information.

```
import breeze.plot._
//import breeze.plot._
val fig = Figure("Regression diagnostics")
//fig: breeze.plot.Figure = breeze.plot.Figure@3583c919
fig.width = 1000
//fig.width: Int = 1000
fig.height = 800
//fig.height: Int = 800
val p = fig.subplot(1,1,0)
//p: breeze.plot.Plot = breeze.plot.Plot@4ebf476d
p += plot(mod3.fitted,mod3.residuals,'.')
//res7: breeze.plot.Plot = breeze.plot.Plot@4ebf476d
p.xlabel = "Fitted values"
//p.xlabel: String = Fitted values
p.ylabel = "Residuals"
//p.ylabel: String = Residuals
p.title = "Residuals against fitted values"
//p.title: String = Residuals against fitted values
```

This gives the plot shown in Fig6.1.

6.1.3 Case study: linear regression for a real dataset

We will conclude this section on linear regression with the analysis of a real dataset. We will use the "yacht hydrodynamics dataset" from the ML repository:

```
http://archive.ics.uci.edu/ml/datasets/Yacht+Hydrodynamics
```

We will begin by downloading the dataset to local disk (if it hasn't been already), converting the white-space separated file to CSV in the process.

```
val url = "http://archive.ics.uci.edu/ml/" + (
```

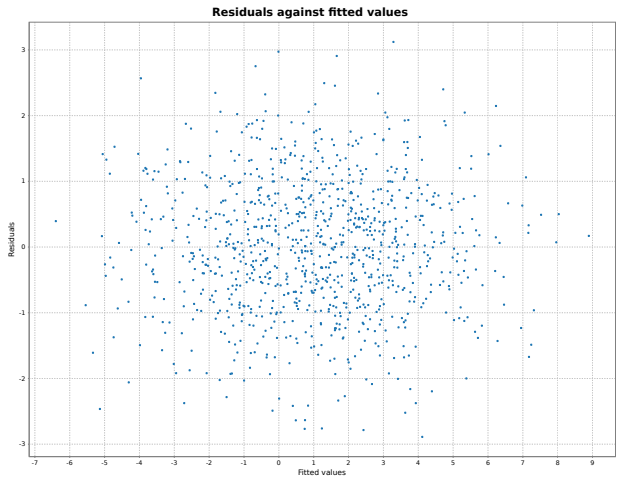


Figure 6.1. Linear regression diagnostics

```
  "machine-learning-databases/00243/") + "yacht_hydrodynamics.data"
//url: String = http://archive.ics.uci.edu/ml/machine-learning-databases/00243/yacht
val fileName = "yacht.csv"
//fileName: String = yacht.csv
val file = new java.io.File(fileName)
//file: java.io.File = yacht.csv
  if(!file.exists) {
    val s = new java.io.PrintWriter(file)
    val data = scala.io.Source.fromURL(url).getLines
    data.foreach(l => s.write(l.trim.split(' ').
filter(_ != "").mkString("",",","\n")))
    s.close
  }
// mkString    makes a collection to a string with prefix, separator, suffix
```

Since the Breeze CSV parser is rather rudimentary, it's best to make the parsing job as simple as possible. Now that the file is on disk, we can parse it and extract the key information as follows.

```
import breeze.linalg._
//import breeze.linalg._
val mat = breeze.linalg.csvread(new java.io.File(fileName))
//mat: breeze.linalg.DenseMatrix[Double] =
//-2.3  0.568  4.78  3.99  3.17  0.125  0.11
//-2.3  0.568  4.78  3.99  3.17  0.15   0.27
//-2.3  0.568  4.78  3.99  3.17  0.175  0.47
println("Dim: " + mat.rows + " " + mat.cols)
//Dim: 308 7
val y = mat(::, 6)     // response in the final column
//y: breeze.linalg.DenseVector[Double] = DenseVector(0.11, 0.27, 0.47, ...
val x = mat(::, 0 to 5)
//x: breeze.linalg.DenseMatrix[Double] =
//-2.3  0.568  4.78  3.99  3.17  0.125
```

```
//-2.3  0.568  4.78  3.99  3.17  0.15
//...
```

Now we have our response and covariate matrix, we can fit the linear model (without an intercept).

```
Lm(y,x,List("LongPos","PrisCoef","LDR","BDR","LBR","Froude")).summary
//Estimate         S.E.    t-stat tp-value        Variable
//-------------------------------------------------------
//  0.1943         0.338   0.575  0.5655          LongPos
//-35.6159        16.005  -2.225  0.0268 *        PrisCoef
// -4.1631         7.779  -0.535  0.5929          LDR
//  1.3747         3.297   0.417  0.6770          BDR
//  3.3232         8.911   0.373  0.7095          LBR
//121.4745         5.054  24.034  0.0000 *        Froude
//
//Residual standard error:   8.9522 on 302 degreesof freedom
//Multiple R-squared: 0.6570, Adjusted R-squared: 0.6513
//F-statistics: 115.6888 on 5 and 302 DF, p-value: 0.00000
//
```

This suggests that two covariates are significant. However, fitting without an intercept doesn't really make sense here, so we can refit including an intercept.

```
val X = DenseMatrix.horzcat(
  DenseVector.ones[Double](x.rows).toDenseMatrix.t,x)
//X: breeze.linalg.DenseMatrix[Double] =
//1.0  -2.3  0.568  4.78  3.99  3.17  0.125
//1.0  -2.3  0.568  4.78  3.99  3.17  0.15
//1.0  -2.3  0.568  4.78  3.99  3.17  0.175
//...
val mod = Lm(y,X,List("(Intercept)","LongPos",
  "PrisCoef","LDR","BDR","LBR","Froude"))
//mod: Lm =
//Lm(DenseVector(0.11, 0.27, 0.47, ...
mod.summary
//Estimate         S.E.    t-stat tp-value        Variable
//-------------------------------------------------------
//-19.2367        27.113  -0.709  0.4786          (Intercept)
//  0.1938         0.338   0.573  0.5668          LongPos
// -6.4194        44.159  -0.145  0.8845          PrisCoef
//  4.2330        14.165   0.299  0.7653          LDR
// -1.7657         5.521  -0.320  0.7493          BDR
// -4.5164        14.200  -0.318  0.7507          LBR
//121.6676         5.066  24.018  0.0000 *        Froude
//
//Residual standard error:   8.9596 on 301 degreesof freedom
//Multiple R-squared: 0.6576, Adjusted R-squared: 0.6507
//F-statistics: 96.3327 on 6 and 301 DF, p-value: 0.00000
```

//

This time only the **Froude** variable is significant.

6.2 Generalised linear models

6.2.1 Introduction

At its most basic level (computing regression coefficients), linear regression can be viewed purely as a least squares problem, independently of any modelling assumptions. But in practice, diagnostic statistics rely on certain assumptions, including iid Gaussian errors. This means that linear regression doesn't work very well when errors aren't approximately Gaussian, and this can be a particular problem for a binary response or count data. *Generalised linear models* (GLMs) extend linear modelling theory to cover response variables in the *exponential family*. The theory of GLMs is out of scope for this course, but we just need some basics. Typically GLM theory is presented for a two-parameter (or, in fact, a scaled one-parameter) exponential family. But for the most commonly used cases: Bernoulli (logistic), Binomial, and Poisson regression, a simple one-parameter exponential family is sufficient. So we will assume that our response variable has a probability mass (or density) function of the form

$$f(y|\theta) \quad = \quad \exp\{\theta y - b(\theta) + c(y)\} \tag{6.16}$$

where θ is the *canonical* parameter of the distribution. Then setting up a regression model with a linear predictor for the canonical parameter (a *canonical link*), we can construct a Newton-Raphson scheme for finding the maximum likelihood estimator, which takes the form of an *iteratively reweighted least squares* (IRLS) algorithm, with updates of the form

$$\beta_{n+1} \quad = \quad \beta_n + [X^T W_n X]^{-1} X^T z_n \tag{6.17}$$

where

$$W_n \quad = \quad \text{diag}\{b''(X\beta_n)\} \quad \text{and} \quad z_n = y - b'(X\beta_n) \tag{6.18}$$

There are efficient QR-based solutions to this problem [9], but here we will present a simple illustrative implementation.

Note: The total log-likelihood of (y_i, θ_i) over all i is

$$\sum_i \log f(y_i|\theta_i) \quad = \quad \sum_i \{\theta_i y_i - b(\theta_i) + c(y_i)\} \tag{6.19}$$

and the maximum log-likelihood for $\theta_i = \sum_\nu X_{i\nu}\beta_\nu$ is attained at $\partial \sum_i \log f(y_i|\theta_i)/\partial\beta_\mu = 0$; that is,

$$\sum_i (X^T)_{\mu i} z_i \quad = \quad 0 \tag{6.20}$$

$$z_i \quad \equiv \quad (y_i - b'(\sum_\nu X_{i\nu}\beta_\nu)) \tag{6.21}$$

Here, the Newton-Raphson method is used to solve the above equation. The partial derivative of

the $X^T z$ by β_κ is

$$-\sum_i (X^T)_{\mu i} b''(\sum_\nu X_{i\nu}\beta_\nu) X_{i\kappa} = -[X^T \text{diag}(b''(\sum_\nu X_{i\nu}\beta_\nu))X]_{\mu\kappa} \quad (6.22)$$

Note:

$$(X^T W_n X)^{-1} X^T z_n = (X^T W_n^{1/2} W_n^{1/2} X)^{-1} X^T z_n \quad (6.23)$$
$$= (W_n^{1/2} X)^T (W_n^{1/2} X))^{-1} (W_n^{1/2} X)^T W_n^{-1/2} z_n \quad (6.24)$$

IF $W_n^{1/2} X = QR$, where $Q^T Q = I$ and R is an upper triangular matrix,

$$(X^T W_n X)^{-1} X^T z_n = (R^T R)^{-1} (QR)^T W_n^{-1/2} z_n \quad (6.25)$$
$$= R^{-1} Q^T W_n^{-1/2} z_n \quad (6.26)$$

Thus, the a **backSolve** function in page 101 can be used to solve the above equation.

6.2.2 Logistic regression

Arguably the simplest, and most widely used non-Gaussian GLM is *logistic regression*, with a binary response variable. Here we assume that

$$y_i \sim \text{Bern}(p_i) \quad (6.27)$$

for some predictor p_i, and so the response distribution is

$$f(y|p) = p^y (1-p)^{(1-y)} \quad (6.28)$$
$$= (\frac{p}{1-p})^y (1-p) \quad (6.29)$$
$$= \exp\{y \log(\frac{p}{1-p}) + \log(1-p)\} \quad (6.30)$$
$$= \exp\{\theta y - b(\theta) + c(y)\} \quad (6.31)$$

where

$$\theta = \log\{p/(1-p)\} \equiv \text{logit}(p) \quad (6.32)$$
$$b(\theta) = \log(1 + e^\theta) \quad (6.33)$$
$$c(\theta) = 0 \quad (6.34)$$

Thus, the response distribution is one-parameter exponential with canonical parameter θ.

Let's now see how this works in practice. We begin by simulating some synthetic data from a logistic regression model.

```
import breeze.linalg._
//import breeze.linalg._
import breeze.stats.distributions.{Gaussian,Binomial}
//import distributions.{Gaussian, Binomial}
val N = 2000
```

```
//N: Int = 2000
val beta = DenseVector(0.1,0.3)
//beta: DenseVector[Double] = DenseVector(0.1, 0.3)
val ones = DenseVector.ones[Double](N)
//ones: DenseVector[Double] = DenseVector(1.0, ...
val x = DenseVector(Gaussian(1.0,3.0).sample(N).
  toArray)
//x: breeze.linalg.DenseVector[Double] = DenseVector(-0.978069320205412, ...
val X = DenseMatrix.vertcat(ones.toDenseMatrix,
  x.toDenseMatrix).t
//X: breeze.linalg.DenseMatrix[Double] =
//1.0  -0.978069320205412
//1.0  0.30609565039259234
//...
val theta = X * beta
//theta: breeze.linalg.DenseVector[Double] = DenseVector(-0.19342079606162357, ...
def expit(x: Double): Double = 1.0/(1.0+math.exp(-x))
//expit: (x: Double)Double
val pr = theta map expit
//pr: breeze.linalg.DenseVector[Double] = DenseVector(0.4517949929153158, ...
val y = pr map (pi => (new Binomial(1,pi)).draw) map ( _.toDouble)
//y: breeze.linalg.DenseVector[Double] = DenseVector(0.0, 1.0, ...
```

Note that the linear predictor is constructed for the canonical parameter **theta**, and that this is pushed through the **expit** function (often called the *sigmoid*, or *logistic* function, the inverse of the *logit*) to get a probability. Now we have our synthetic data, we need an implementation of the IRLS(iteratively reweighted least squares) algorithm for finding the optimal regression coefficients. A very simple implementation is given below.

```
@annotation.tailrec
def IRLS(
  b: Double => Double,
  bp: Double => Double,
  bpp: Double => Double,
  y: DenseVector[Double],
  X: DenseMatrix[Double],
  bhat0: DenseVector[Double],
  its: Int,
  tol: Double = 0.0000001
): DenseVector[Double] = if (its == 0) {
  println("WARNING: IRLS did not converge")
  bhat0
} else {
  val eta = X * bhat0
  val W = diag(eta map bpp)
  val z = y - (eta map bp)
  val bhat = bhat0 + (X.t * W * X) \ (X.t * z)
  if (norm(bhat-bhat0) < tol) bhat else
```

```
    IRLS(b,bp,bpp,y,X,bhat,its-1,tol)
}
//IRLS: (b: Double => Double, bp: Double => Double, bpp: Double => Double,
//  y: breeze.linalg.DenseVector[Double], X: breeze.linalg.DenseMatrix[Double],
//  bhat0: breeze.linalg.DenseVector[Double], its: Int,
//  tol: Double)breeze.linalg.DenseVector[Double]
```

Though very simple, this function works for any one-parameter exponential family GLM, accepting as arguments the function **b**= $b(\theta)$ (which isn't actually required), its derivative **bp** and second derivative **bpp**. We can write a simple wrapper for this for the case of logistic regression as follows.

```
def logReg(
  y: DenseVector[Double],
  X: DenseMatrix[Double],
  its: Int = 30
): DenseVector[Double] = {
  def expit(x: Double): Double = 1.0/(1.0+math.exp(-x))
  def b(x: Double): Double = math.log(1.0+math.exp(x))
  def bp(x: Double): Double = expit(x)
  def bpp(x: Double): Double = {
    val e = math.exp(-x)
      e/((1.0+e)*(1.0+e))
  }
  val bhat0 = DenseVector.zeros[Double](X.cols)
  IRLS(b,bp,bpp,y,X,bhat0,its)
}
//logReg: (y: breeze.linalg.DenseVector[Double],
//  X: breeze.linalg.DenseMatrix[Double],
//  its: Int)breeze.linalg.DenseVector[Double]
```

We can now call this to fit a logistic regression model as follows.

```
val betahat = logReg(y,X)
//betahat: breeze.linalg.DenseVector[Double] =
//  DenseVector(0.018413492658916366, 0.30531759133926595)
```

For diagnostic purposes, we could plot our fitted probabilities along with the observed data.

```
import breeze.plot._
// import breeze.plot._
val fig = Figure("Logistic regression")
//fig: breeze.plot.Figure = breeze.plot.Figure@196e046d
fig.width = 1000
//fig.width: Int = 1000
fig.height = 800
//fig.height: Int = 800
var p = fig.subplot(1,1,0)
//p: breeze.plot.Plot = breeze.plot.Plot@23d99d1c
```

```
p += plot(x,y,'+')
//res0: breeze.plot.Plot = breeze.plot.Plot@23d99d1c
p += plot(x,x map (xi =>
  expit(betahat(0)+betahat(1)*xi)),
  '.',colorcode="red")
//res1: breeze.plot.Plot = breeze.plot.Plot@23d99d1c
p.xlabel = "x"
//p.xlabel: String = x
p.ylabel = "y"
//p.ylabel: String = y
p.title = "Logistic regression"
//p.title: String = Logistic regression
fig.refresh
```

This gives the plot shown in Fig6.2.

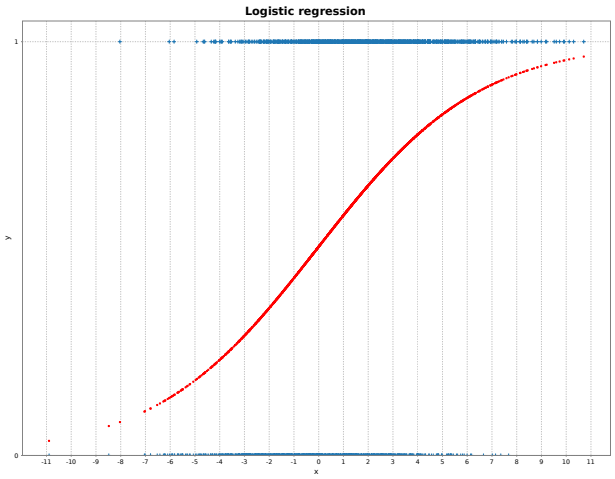


Figure 6.2. Logistic regression

6.2.3 Poisson regression

Poisson regression is often used when the response variable is a nonnegative integer "count" of some sort. This is also exponential family, as we model the response variable as

$$y_i = \text{Pois}(\mu_i) \tag{6.35}$$

for some predictor μ_i, and so the response distribution is

$$f(y|\mu) = \frac{\mu^x e^{-\mu}}{y!} = \exp\{y \log \mu - \mu - \log y!\} \tag{6.36}$$

$$f(y|\theta) = \exp\{\theta y - b(\theta) + c(y)\} \tag{6.37}$$

where

$$\theta = \log \mu \tag{6.38}$$
$$b(\theta) = e^{\theta} \tag{6.39}$$
$$c(y) = -\log y! \tag{6.40}$$

Thus, the mass or density function f is exponential family. So here we have a log link, which means that the linear predictor is exponentiated to get the mean of the Poisson distribution

Again we will begin by simulating some synthetic data consistent with the assumptions of a Poisson regression model.

```
import breeze.linalg._
//import breeze.linalg._
import breeze.numerics._
//import breeze.numerics._
import breeze.stats.distributions.{Gaussian,Poisson}
//import breeze.stats.distributions.{Gaussian, Poisson}
val N = 2000
//N: Int = 2000
val beta = DenseVector(-3.0,0.1)
//beta: breeze.linalg.DenseVector[Double] = DenseVector(-3.0, 0.1)
val ones = DenseVector.ones[Double](N)
//ones: breeze.linalg.DenseVector[Double] = DenseVector(1.0,1.0,...
val x = DenseVector(Gaussian(50.0,10.0).sample(N).toArray)
//x: breeze.linalg.DenseVector[Double] = DenseVector(57.470401333447626, ...
val X = DenseMatrix.vertcat(ones.toDenseMatrix, x.toDenseMatrix).t
//X: breeze.linalg.DenseMatrix[double] =
//1.0  57.470401333447626
//1.0  51.87227699389494
//1.0  51.94965234826458
//...
val theta = X * beta
//theta: breeze.linalg.DenseVector[Double] = DenseVector(2.747040133344763, ...
val mu = exp(theta)
//mu: breeze.linalg.DenseVector[Double] = DenseVector(15.596400233187346, ...
val y = mu map (mui => new Poisson(mui).draw) map (_.toDouble)
//y: breeze.linalg.DenseVector[Double] = DenseVector(14.0, 9.0, 6.0, ...
```

Now we have a synthetic dataset we can define a function for fitting a Poisson regression model, reusing our **IRLS** function from previously.

```
def poiReg(
  y: DenseVector[Double],
  X: DenseMatrix[Double],
  its: Int = 30
): DenseVector[Double] = {
  val bhat0 = DenseVector.zeros[Double](X.cols)
  IRLS(math.exp,math.exp,math.exp,y,X,bhat0,its)
```

```
}
//poiReg: (y: breeze.linalg.DenseVector[Double],
//  X: breeze.linalg.DenseMatrix[Double], its: Int)
//  breeze.linalg.DenseVector[Double]
```

We can now use this function to fit a model.

```
poiReg(y,X)
//WARNING: IRLS did not converge
//res0: breeze.linalg.DenseVector[Double] = DenseVector(-75.96335856948667, 1.159977
```

This runs, but prints a warning to the console indicating that the maximum number of iterations was reached. The estimated parameters are terrible. However, we can easily re-fit the model, bumping up the maximum iteration count.

```
val betahat = poiReg(y,X,its=100)
//betahat: breeze.linalg.DenseVector[Double] =
//  DenseVector(-2.9829989727876494, 0.0995945111252215)
```

This time there is no warning and the estimated parameters look good.

Finally, we can plot the data and the fitted means.

```
import breeze.plot._
//import breeze.plot._
val fig = Figure("Poisson regression")
//fig: breeze.plot.Figure = breeze.plot.Figure@31b3537b
fig.width = 1000
//fig.width: Int = 1000
fig.height = 800
//fig.height: Int = 800
var p = fig.subplot(1,1,0)
//p: breeze.plot.Plot = breeze.plot.Plot@54a23ced
p += plot(x,y,'+')
//res0: breeze.plot.Plot = breeze.plot.Plot@54a23ced
p += plot(x,x map (xi => math.exp( betahat(0)+betahat(1)*xi)),
  '.',colorcode="red")
//res1: breeze.plot.Plot = breeze.plot.Plot@54a23ced
p.xlabel = "x"
//p.xlabel: String = x
p.ylabel = "y"
//p.ylabel: String = y
p.title = "Poisson regression"
//p.title: String = Poisson regression
```

This gives the plot shown in Fig. 6.3.

6.3 The scala-glm library

While writing these notes it became apparent that it would be quite useful to have the code from this Chapter for fitting linear and generalised linear models tidied, improved, documented and packaged as a

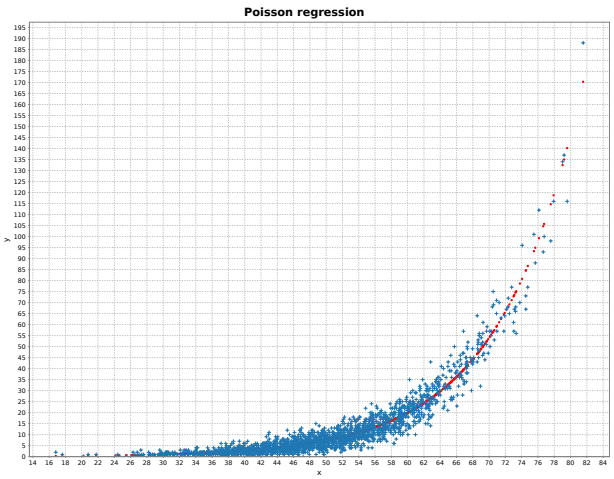


Figure 6.3. Poisson regression

Scala library, for convenience. I've now done this, and the library is available on-line, called **scala-glm**, https://github.com/darrenjw/scala-glm/. We will explore this (small) library as part of the end-of-chapter exercises.

6.4 Data frames and tables in Scala

Introduction

To statisticians and data scientists used to working in R, the concept of a data frame is one of the most natural and basic starting points for statistical computing and data analysis. It always surprises me that data frames aren't a core concept in most programming languages' standard libraries, since they are essentially a representation of a relational database table, and relational databases are pretty ubiquitous in data processing and related computing. For statistical modelling and data science, having functions designed for data frames is more convenient than using functions designed to work directly on vectors and matrices, for example. Trivial things like being able to refer to columns by a readable name rather than a numeric index makes a huge difference, before we even get into issues like columns of heterogeneous types, coherent handling of missing data, etc. This is why modelling in R is typically nicer than in certain other languages, where libraries for scientific and numerical computing existed for a long time before libraries for data frames were added to the language ecosystem. To build good libraries for statistical computing in Scala, it would be helpful to build those libraries using a good data frame implementation. There are a few different examples in existence, but unfortunately none have gained significant traction as yet.

For this course, we are not going to spend significant time working with the existing data frame libraries, since they are all rather experimental. Nevertheless, it seems sensible to briefly describe some of the available libraries, and to compare and contrast their features.

A simple data manipulation task

In order to illustrate the issues, I'm going to consider a very simple data manipulation task: first reading in a CSV file from disk into a data frame object, then filtering out some rows, then adding a derived column, then finally writing the data frame back to disk as a CSV file. We will start by looking at how this would be done in R. First we need an example CSV file. Since many R packages contain example datasets, we will

use one of those. We could export **Cars93** from the R MASS package with the following R commands in an interactive R session:

```
> library(MASS)
```

If MASS isn't installed, it can be installed as follows.

```
> install.packages("MASS").
```

The above code snippet generates a CSV file to be used for the example. Typing **?Cars93** will give some information about the dataset, including the original source. Our analysis task is going to be to load the file from disk, filter out cars with **EngineSize** larger than 4 (litres), add a new column to the data frame, **WeightKG**, containing the weight of the car in KG, derived from the column **Weight** (in pounds), and then write back to disk in CSV format. This is the kind of thing that R is good at:

```
> library(MASS)
> # ?Cars93 for information about the dataset, Cars93
> write.csv(Cars93,"cars93.csv",row.names=FALSE)
> df = read.csv("cars93.csv")
> print(dim(df))
[1] 93 27
> df = df[df$EngineSize<=4.0,]
> print(dim(df))
[1] 84 27
> df$WeightKG = df$Weight*0.453592
> print(dim(df))
[1] 84 28
> write.csv(df,"cars93m.csv",row.names=FALSE)
```

That's how the task could be accomplished using R. Now let's see how a similar task could be accomplished using some different Scala data frames implementations.

6.4.1 Spark DataFrames

The three data frames, Saddle, Scala-database, and Famian, considered so far are all standard single-machine, non-distributed, in-memory objects. However, a **DataFrame** object has A **DataFrame** object has recently been added to Apache Spark. Spark is a Scala framework for the distributed processing and analysis of huge datasets on a cluster. We will examine it in more detail tomorrow. If you have a legitimate need for this kind of set-up, then Spark is a pretty impressive piece of technology. However, for datasets that can be analysed on a single machine, Spark is a rather slow and clunky sledgehammer to crack a nut. So, for datasets in the terabyte range and above, Spark DataFrames are great, but for datasets smaller than a few gigs, it's probably not the best solution. With those caveats in mind, here's an example of how to solve our problem using Spark DataFrames in the Spark Shell:

```
// spark is a predefined val in the spark-shell.
spark.getClass
//res0: Class[_ <: org.apache.spark.sql.SparkSession] =
//  class org.apache.spark.sql.SparkSession
val df = spark.read.
```

```
          option("header", "true").
          option("inferSchema","true").
          csv("cars93.csv")
//df: org.apache.spark.sql.DataFrame = [Manufacturer: string, Model: string ... 25 m
val df2=df.filter("EngineSize <= 4.0")
//df2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Manufacturer: strin
val col=df2.col("Weight")*0.453592
//col: org.apache.spark.sql.Column = (Weight * 0.453592)
val df3=df2.withColumn("WeightKG",col)
//df3: org.apache.spark.sql.DataFrame = [Manufacturer: string, Model: string ... 26
df3.write.format("com.databricks.spark.csv").
                          option("header","true").
                          save("out-csv")
//18/04/11 13:23:41 WARN Utils: Truncated the string representation of a plan
//since it was too large. This behavior can be adjusted by setting
//'spark.debug.maxToStringFields' in SparkEnv.conf.
```

6.4.2 Summary

If you really need a distributed data frame library, then you will probably want to use Spark. However, for the vast majority of statistical modelling and data science tasks, Spark is likely to be unnecessarily complex and heavyweight. The other three libraries considered all have pros and cons. They are all largely one-person hobby projects, quite immature, and not currently under very active development. In particular, at the time of writing Saddle is the only library that has been updated for Scala 2.12 (and it still hasn't published artifacts). Saddle is fine for when you just want to add column headings to a matrix. Scala-datatable is lightweight and immutable, if you don't care about missing values. On balance, I think Framian is probably the most full-featured "batteries included" R-like data frame, and so is likely to be most attractive to statisticians and data scientists. However, it's very immature, appears to be abandoned, and the dependence on Shapeless may be of concern to those who prefer libraries to be lean and lightweight.

Chapter 7

Tools and libraries: Scala Tools

7.1 SBT tips and tricks

We've seen how easy it is to use SBT to add dependencies on third party libraries by adding them into a **build.sbt** file. But when you just want to quickly experiment with a library/API, creating a new project with an appropriate build file can seem like a bit of a chore. Fortunately, it is very easy to interactively add dependencies into a running SBT session, and this can be very convenient. To illustrate this, start SBT from an empty directory, or some kind of /tmp/ directory which doesn't contain any files ending .sbt or .scala. This will give you a "vanilla" SBT session. We know that if we type console we will get a Scala REPL, but which version of Scala will it be? Before typing console, we can set the Scala version from the SBT prompt with

```
sbt:tmp> set scalaVersion := "2.12.6"
```

Now when we type console we will get a 2.12.1 REPL. Obviously you can set this to any version of Scala you like, so this is also a good way to experiment with different versions of Scala. Having set an appropriate version of Scala, we can next add any dependencies (again, this should be done from the SBT prompt, not the Scala REPL). For example, we can add a dependency on Breeze with:

```
sbt:tmp> set libraryDependencies += "org.scalanlp" %% "breeze" % "0.132"
sbt:tmp> set libraryDependencies += "org.scalanlp" %% "breeze-natives" % "0.13.2"
```

Now when we type console we will get a REPL with a Breeze dependency. This approach to adding in dependencies interactively can be very useful for "quick" and "dirty" experiments with new libraries in the REPL.

7.1.1 sbt new

Recent versions of SBT support the command line option new, allowing quick-and-easy creation of SBT project templates pre-configured for particular applications, utilising the giter8 templating system. For example, a minimal seed project can be created by typing:

```
$ sbt new scala/scala-seed.g8
```

at your OS command prompt, from a directory where you want to create a new SBT project as a sub-directory. Templates typically query a project name, and can also query other setup information. There are templates for many applications. For example, if you are interested in the ScalaFX GUI library for building graphical applications, you can create a minimal project template with:

```
$ sbt new scalafx/scalafx.g8
```

After creating the template, entering the project directory and typing **sbt run** will compile and run a minimal GUI application.

A list of available templates can be found at: https://github.com/foundweekends/giter8/wiki/giter8-templates

I have created a giter8 template for Breeze projects, which can be used with

```
$ sbt new darrenjw/breeze.g8
```

This is arguably easier and better than copying the app-template directory in the GitHub repo associated with this course, though it does require an internet connection.

To search for a template using, eg., Google, just search for the name of the library and add g8 onto the end. eg. To search for an akka-streams template, search for akka-streams g8. But beware that this can often return very old templates.

This is arguably easier and better than copying the app-template directory

7.1.2 Increasing heap memory

SBT accepts some command line options on start-up, though these are not well-documented. If you have issues with SBT crashing due to running out of heap memory, you can start it with more by starting it with a command like:

```
sbt -mem 8000
```

The integer argument is given in megabytes, so the above command starts SBT with around 8 Gb of heap space.

7.1.3 Building standalone applications (assembly JARs)

So far we have been building and running our code using SBT. But sometimes we would like to be able to run our code without using SBT. Indeed, we would sometimes like to be able to run our code on systems where SBT isn't installed. For a project with (multiple) dependencies on third party libraries, the simplest way to do this is to build a "fat" (assembly) Java archive (JAR) file containing all of the compiled code associated with the project in addition to the Scala standard library and all third party libraries required to run the application. This assembly JAR can then be run on any machine with a (compatible) JVM. This is simplest to illustrate with an example. Suppose we have an SBT project containing a single Scala source file as follows.

```
object Metropolis {
  import breeze.stats.distributions._
  val chain = MarkovChain.
    metropolisHastings(0.0,
    (x: Double) => Uniform(x-0.5, x+0.5))(x =>
      Gaussian(0.0, 1.0).logPdf(x)).steps

  def main(args: Array[String]): Unit = {
    val n = if (args.size == 0) 10 else args(0).toInt
```

```
      chain.take(n).toArray.foreach(println)
    }

}
```

We can build and run this by typing sbt run from the OS command prompt in the top level directory of the SBT project. If we want to run the chain for 20 iterations, we can do so with **sbt** "run 20". To build an assembly JAR we need to use the **sbt-assembly** SBT plugin. To add this to a project, create a file called **project/plugins.sbt** and add the single line

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.4")
```

(The manual of sbt/sbt-assembly indicates **project/assembly.sbt** as the filename.) This adds a new SBT task, **assembly**. So we can now build an assembly JAR by running **sbt assembly** from the OS command prompt. This should build the JAR and put it in **target/scala-2.12/**. We can run this without SBT with a command like

```
$ java -jar target/scala-2.12/metropolis-assembly-assembly-0.1.jar
```

or

```
$ java -jar target/scala-2.12/metropolis-assembly-assembly-0.1.jar 20
```

So we can now deploy and run this on any system with a (compatible) JVM. This is exceptionally convenient in the context of cluster/Cloud computing environments. Also, we shall see shortly that it is also convenient if you want to call Scala code from other systems and languages, such as R.

7.2 Interfacing Scala with R

Unfortunately, for the time being R remains the "default" language for statistical computing and data science, and has thousands of packages (on CRAN) implementing a huge variety of statistical methods and algorithms. Also, the statistical plotting libraries available in R are still somewhat superior to those easily available in Scala. Consequently, R is likely to remain part of the statistical computing ecosystem for the foreseeable future (for better or worse), and so having mechanisms for using R and Scala together in statistical applications is clearly valuable.

There are obviously two different ways that one could use R and Scala together. One would be to call R functionality from a Scala application (embedding R in Scala), perhaps to fit a model using a sophisticated model-fitting package like **lme4**, not routinely available as a Scala library, or alternatively to call a plotting function that will produce a publication-quality plot, possibly using R's **ggplot** library. The other way to use R and Scala together would be to call a Scala function from an R script (embedding Scala in R). Here a typical use–case would be that you have a pre-existing R application that is too slow, and so you want to re-write the slow parts in Scala and then call the Scala version from your R script. Both of these ways of combining R and Scala are addressed by the rscala package [10].

7.2.1 Calling R from Scala

We will begin by seeing how easy it is to use **rscala** [10] to embed R in a Scala application. Start by running SBT from an empty/temp directory, and add a dependency on **rscala** by entering:

```
set scalaVersion := "2.12.6"
set libraryDependencies+="org.ddahl"%%"rscala"%"2.5.3"
console
```

For this to work, you must have a system-wide installation of a recent version of R on your computer.

to get an appropriately configured Scala REPL.

An R session can be started from within Scala with

```
val R = org.ddahl.rscala.RClient(serializeOutput=true,rowMajor=true,debug=false)
//R: org.ddahl.rscala.RClient = org.ddahl.rscala.RClient@5ba23b66
```

The return value (here **R**) is a handle on the R session. Note that this starts R using the default R command, which can be obtained with

```
org.ddahl.rscala.RClient.defaultRCmd
//res2: String = R
```

If you start up R with a non-standard command (for example, because you have multiple versions of R installed on your system), this can be passed into the **RClient** method. Now we have a handle on an R session, we can evaluate R expressions and have the results returned back to Scala. For example, we can evaluate a function returning a scalar **Double** with

```
val d0 = R.evalD0("rnorm(1)")
//d0: Double = 0.025083626418306904
```

Here the D stands for **Double** and the 0 for 0-dimensional, ie. a scalar. So we can evaluate a function returning a vector of **Double**s with

```
val d1 = R.evalD1("rnorm(5)")
//d1: Array[Double] = Array(-1.0694128054523035, 0.5125469424381456, ...)
```

Note how the Scala return type is **Array[Double]**. Similarly, an R expression returning a matrix of **Double**s can be evaluated with

```
val d2 = R.evalD2("matrix(rnorm(6), nrow=2)")
//d2: Array[Array[Double]] = Array(Array(-0.04814148330742599, ...)
```

Note how the return type is **Array[Array[Double]]**. We can send data from Scala to R by creating R variables containing the data. For example, we can create an R variable called **vec** with

```
R.vec = (1 to 10).toArray
//R.vec: (Any, String) = ([I@7c9bdee9,Array[Int])
R.evalI1("vec")
//res1: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Here, **I** is for **Int**. We can evaluate multi-line expressions (and discard any return value) as follows.

```
R.eval("""
vec2 = rep(vec, 3)
vec3 = vec2 + 1
mat1 = matrix(vec3, ncol=5)
""")
//
```

If we want to just "get" an R variable, we can do so:

```
R.getI2("mat1")   // get data back from R
//res13: Array[Array[Int]] = Array(Array(2, 8, 4, 10, 6),...)
```

We will finish this section with a more realistic (Breeze-based) example, where we have data in Scala that we want to fit using a model-fitting routine available in R. We start by simulating some data from Poisson regression model (in Scala).

```
import breeze.stats.distributions._
import breeze.linalg._
import org.ddahl.rscala.RClient
val x = Uniform(50,60).sample(1000)
//x: IndexedSeq[Double] = Vector(50.01006231472438, ...
val eta = x map (xi => (xi * 0.1) - 3)
//eta: IndexedSeq[Double] = Vector(2.001006231472438, ...
val mu = eta map math.exp
//mu: IndexedSeq[Double] = Vector(7.396494941700207, ...
val y = mu map (Poisson(_).draw)
//y: IndexedSeq[Int] = Vector(8, 16, 14, 12, 11,
```

Now we have our data in Scala, we can create a connection to R and use it to fit our model.

```
val R = RClient() // initialize an R interpreter
//R: org.ddahl.rscala.RClient = org.ddahl.rscala.RClient@5e98032e
R.x = x.toArray    // send x to R
//R.x: (Any, String) = ([D@48b2dbc4,Array[Double])
R.y = y.toArray // send y to R
//R.y: (Any, String) = ([I@11f9b95a,Array[Int])
R.eval("mod = glm(y~x, family=poisson())")     // fit
//
val result = DenseVector[Double](R.evalD1("mod$coefficients"))
//result: breeze.linalg.DenseVector[Double] = DenseVector(-2.8120575725829613, 0.096
R.eval("""print(summary(mod))""")
```

7.2.2 Calling Scala from R

It can also sometimes be useful to embed Scala in iR so that Scala code can be called from R. To do this we need to install the CRAN package **rscala** in R. Assuming that both R and a JVM is installed on your system, installing this should be as simple as typing

```
R> install.packages("rscala")
```

at the R command prompt. Calling

```
R> library(rscala)
```

will check that it has worked. The package will do a sensible search for a Scala installation and use it if it can find one. If it can't find one it will fail. In this case you can download and install a Scala installation specifically for rscala using the command

```
R> scalaInstall()
```

This option is likely to be attractive to sbt (or IDE) users who don't like to rely on a system-wide scala installation.

Getting started

A connection to a Scala interpreter can be constructed using

```
R> sc = scala()
```

It's possible to find out information about the interpreter with

```
R> scalaInfo(verbose=TRUE)

Searching for a suitable Scala installation.
  * FAILURE: 'scala.home' argument is NULL
  * FAILURE: 'rscala.scala.home' () global option
  * FAILURE: SCALA_HOME () environment variable
  * ATTEMPT: Found a candidate (/localhome/usrlocal-amd64-linux/newbin/scala.sh)
    - scala-library.jar is not in 'lib' directory of assumed Scala home (/localhome/
    - Looking for 'scala.home' property in 'scala' script
  * SUCCESS: 'scala' in the shell's search path

$cmd
[1] "/localhome/usrlocal-amd64-linux/newbin/scala.sh"

$home
[1] "/usr/share/scala-2.11"

$version
[1] "2.11.8"

$major.version
[1] "2.11"
>
```

As usual in R, it is possible to get help on commands in the usual way.

```
R> help(package="rscala")
//Description:
//
//Package:              rscala
//Type:                 Package
//...
R> ?scala
//Create an Instance of an Embedded Scala Interpreter
//...
```

We can send a command to the Scala interpreter using the

```
R> sc %~% 'println("hello world")'
//hello world
```

If we want, we can send instructions one line at a time.

```
R> sc %~% 'val x = Array (1 ,2 ,3)'
// [1] 1 2 3
R> sc %~% 'val y = x map (_*2)'
// [1] 2 4 6
```

And we can get data back into R using $ notation.

```
R> ry = sc$y  # get data back to R
R> print(ry)
// [1] 2 4 6
```

However, it is often more convenient to send a collection of expressions and use the fact that the result of the final expression will be returned to R.

```
R> s = sc %~% '
+ val x = List(1 ,2 ,3)
+ ( x map (_+1) ).sum
+ '
R> print(s)
// [1] 9
```

It is similarly straightforward to send data from R to Scala.

```
R> rx = rnorm(5)
R> print(rx)
// [1]  0.7163580  2.0139352 -1.0789291 -2.1124382 -0.9968101
R> sc$sx = rx # send data to Scala
R> sc %~% 'val sy = sx map (xi => xi * xi)'
// [1] 0.5131687 4.0559350 1.1640879 4.4623950 0.9936304
R> ry = sc$sy # get data back to R
R> print(ry)
// [1] 0.5131687 4.0559350 1.1640879 4.4623950 0.9936304
```

Note that for R to understand the result, you should typically use expressions evaluating to primitive numeric types or **Array**s of primitive types. There is also a convenient string interpolation notation @ , which is useful for passing arguments from R into Scala code.

```
R> n = 10
R> sc %~% '(1 to @{n}).toArray'
// [1]  1  2  3  4  5  6  7  8  9 10
```

Finally, when one is finished with the Scala interpreter, it should be closed to free up resources.

```
R> close(sc)
R>
```

Calling into compiled Scala code (with dependencies)

Typically one will want to run Scala code from compiled SBT projects, usually having third-party library dependencies (such as Breeze). In this case, the simplest way to call such code from R via rscala is via an assembly JAR, constructed as described in section 7.1.3. Using the example from section 7.1.3, we can call this from an R session (with working directory the top level of the SBT project) roughly as follows.

```
library(rscala)
sc = scala(
"target/sclaa-2.11/metropolis-assembly-assembly-0.1.jar",
scala.home="~/.rscala/scala-2.11.8"
)
met = sc %~% 'Metropolis.chain.take(10000).toArray'
```

The first argument to the **scala** function is the path to the assembly JAR. This should be the only argument required if the system-wide installation of Scala is compatible with the version used to build the assembly JAR. If not, the **scala.home** option must be set to a Scala installation which is. Here I point at an installation provided by **scalaInstall()**. We can then run the compiled code by evaluating an appropriate expression.

To check that this has worked as expected, we can use the **mcmcSummary** function from my **smfsb** CRAN package.

```
R> # install.packages("smfsb")   # to install
R> library(smfsb)
R> mcmcSummary(matrix(met,ncol=1))
```

This should provide some plots and summary statistics indicating that the chain is targeting a standard normal distribution.

Summary

The CRAN package rscala makes it very easy to embed a Scala interpreter within an R session and vice versa. However, for most non-trivial statistical computing problems, the Scala code will have dependence on external scientific libraries such as Breeze. The standard way to easily manage external dependencies in the Scala ecosystem is SBT. Given an SBT-based Scala project, it is easy to generate an assembly JAR in order to initialise the rscala Scala interpreter with the classpath needed to call arbitrary Scala functions. This provides very convenient inter-operability between R and Scala for many statistical computing applications.

7.3 CSV parsing libraries

As we have seen, the CSV parsing function built in to Breeze is rather rudimentary. When it proves inadequate, a more customisable parsing library is needed. There are many parsing libraries for Scala, and we unfortunately do not have time to explore them in the context of this course, but there are a couple of popular CSV parsing libraries worth mentioning here for future reference. The most popular library seems to be scala-csv. This is very simple, but flexible. Other popular Scala libraries for CSV parsing include kantan.csv and PureCSV, but there are many others. However, some prefer to use some of the more sophisticated Java parsing libraries, such as univocity.

7.4 Testing

Testing code to check that it behaves as expected is very important, but is not often taken sufficiently seriously in the scientific and statistical computing communities. Fortunately the Scala and SBT ecosystem is full of tools to make testing code less of a chore. We will begin with some very simple features built into the Scala standard library.

7.4.1 Assert and require

We have already seen the use of the **require** function in section 6.1 This is typically used to document consistency requirements of function arguments. An expression returning a **Boolean** is provided to the function, with **true** corresponding to satisfying the requirements. If the expression evaluates to false, the function will throw an exception (an **IllegalArgumentException**). Note that **require** provides testing at runtime, and therefore there will always be some runtime penalty associated with using **require**. It should therefore be used sparingly for functions that are likely to be called frequently in an inner-loop of some sort.

A useful alternative to **require** is **assert**, which can be used in many different contexts within a code-base. Again, it has a Boolean expression with true corresponding to the assertion and false corresponding to a violation of the assertion, typically resulting in the throwing of an exception, this time an **AssertionError**. If the assertion is used near the start of a function, it can be used to specify pre-conditions, similarly to require. If it is used near the end of a function, in can be used to specify post-conditions.

> Pre- and post-conditions are an important part of a design-by-contract approach to software engineering, and are built-in to some languages (such as Eiffel).

In fact, assert can be used anywhere in code to document the programmer's expectations about the way that things should be at that point. Like **require**, **assert** is called at runtime, and so there is potentially a runtime penalty associated with using **assert**. However, there is an important difference with **assert**, in that assertions can be disabled at compile time, so that they are not checked, and therefore incur no runtime penalty. The idea is that they can be used for development, testing and debugging, but then turned off in production code for performance reasons. This means that one can be much more relaxed about the liberal use of assertions in code, safe in the knowledge that they can be disabled when runtime performance is critical. Some simple examples are given below.

```
require(1 == 1) // satisfied
// require(1 == 2) // throws exception
assert (1 == 1) // satisfied
// assert ( 1 == 2) // throws exception
```

A more interesting example is given below, which uses **require** for checking a pre-condition and **assert** for checking a post-condition.

```
def sqrt(x: Double): Double = {
  require(x >= 0.0) // pre-condition
  val ans = math.sqrt(x)
  assert(math.abs(x-ans*ans) < 0.00001) // post-condition
  ans
}

sqrt(2.0) // works as expected
//res1: Double = 1.4142135623730951
```

Note that assertions can be disabled simply by adding the line

```
scalacOptions += "-Xdisable-assertions"
```

to the project's SBT build file.

There is also a function called **assume**, which behaves identically to **assert**, but is semantically different. The intention is that **assume** corresponds to an axiom and **assert** corresponds to a statement that is required to be proved. This difference is important mainly in the context of static code analysis.

7.4.2 scalatest

scalatest is a widely used unit-testing framework for Scala. The idea behind unit-testing is that every function you write should be tested with a few test cases to ensure that it behaves as expected. These test cases can be kept in a test-suite, and re-run frequently to ensure that the code hasn't been changed in such a way as to cause the test cases to fail. Unit-tests can be run frequently, potential every time a source-file is saved. But the philosophy is quite consistent with that of statically-typed compiled languages in general – do as much work and checking as possible at compile-time to save processor cycles at runtime.

Within an SBT project, unit test code is put in **src/test/scala** (rather than **src/main/scala**), and can be run with the test task in SBT. To rerun all unit tests whenever a source code file changes, run the test task.

scalatest supports the development of tests in a number of different styles. The FlatSpec style seems to be the most popular. A complete FlatSpec test file is given below.

Note: see styles

```
import org.scalatest.FlatSpec

class SetSpec extends FlatSpec {

  "An empty Set" should "have size 0" in {
    assert(Set.empty.size == 0)
  }

  it should "throw an exception with head" in {
    assertThrows[NoSuchElementException] {
      Set.empty.head
    }
  }
}
```

An example of a more statistical test might be:

```
  "A Gamma(3.0, 4.0)" should "have mean 12.0" in {
    import breeze.stats.distributions.Gamma
    val g = Gamma(3.0,4.0)
    val m = g.mean
    assert(math.abs(m - 12.0) < 0.000001)
  }
```

As well as testing your own code, tests can be useful for testing your understanding of the API of a third-party library.

7.4.3 scalacheck

scalacheck (www.scalacheck.org) is an interesting alternative to conventional unit-test based frameworks, which instead advocates property-based testing. The idea is that rather than focusing on a few carefully chosen test-cases, one instead specifies the expected behaviour of a function (corresponding loosely to a post-condition), and the test-suite randomly generates a large number of test-cases to check that the requirements are satisfied. Time constraints prevent us from exploring this in detail within the context of this course, but I recommend exploring the use of this library as time permits.

7.4.4 Benchmarking and profiling

omitted

7.4.5 Documentation using ScalaDoc

Specially formatted comments can be used to document code. The example below shows how to document a function. Note how the comment starts /** rather than the usual /* for a multi-line comment.

```
/**
  * Take every th value from the stream  s of type T
  *
  * @param s  A Stream to be thinned
  * @param th Thinning interval
  *
  * @return The thinned stream, with values of
  * the same type as the input stream
  */
def thinStream[T](s: Stream[T],th: Int): Stream[T] = {
  val ss = s.drop(th-1)
  if (ss.isEmpty) Stream.empty else
  ss.head #:: thinStream(ss.tail, th)
}
```

HTML documentation can be generated for an application by running the **doc** SBT task, and this will be written to **target/scala-2.11/api/**. Pointing a web browser at this directory will then allow browsing of the interactive API documentation. IDEs will often assist with the formatting of ScalaDoc comments.

7.5 Note: How to execute (exec) external system commands in Scala [3)]

7.5.1 The methods, ! and !!, added to the String class by sys.process.stringToProcess

```
import scala.sys.process.stringToProcess
//import scala.sys.process.stringToProcess
val ls = "ls -la /etc".! // Its completion code is returned
//total 1884
//drwxr-xr-x 174 root     root       12288 Jun 24 18:13 ./
//...
```

```
//ls: Int = 0
ls
//res0: Int = 0
val list = "ls -la /etc".!! // Full output from the command is returned.
//list: String =
//"total 1884
//drwxr-xr-x 174 root     root       12288 Jun 24 18:13 .
//...
list
//res1: String =
//"total 1884
//drwxr-xr-x 174 root     root       12288 Jun 24 18:13 .
//...
```

You can also explicitly create a Process object to execute an external command, if you prefer:

```
import scala.sys.process.Process
//import scala.sys.process.Process
val ls = Process("ls -la /etc").! // Its completion code is returned
//...
//ls: Int = 0
```

7.5.2 Also, sys.process.stringSeqToProcess can be used.

```
import sys.process.stringSeqToProcess
//import sys.process.stringSeqToProcess
val exitCode = Seq("ls", "-a", "-l", "/tmp").!
//...
//exitCode: Int = 0
val files = Seq("ls", "-a", "-l", "/tmp").!!
//files: String =
//...
```

7.5.3 "|" and "cd" are a shell built-in command.

```
import scala.sys.process._
//import scala.sys.process._
val result = "ls -la /tmp | wc -l".!
// ...
//ls: |: No such file or directory
//ls: grep: No such file or directory
//ls: pdf: No such file or directory
result: Int = 2
```

7.5.4 How can a pipe shell-command be mimiced in Scala?

```
import scala.sys.process._
```

```
//import scala.sys.process._
val result = ("ls -al /tmp" #| "wc -l").!!
//result: String =
//"36
//"
println(result)
//36
```

7.5.5 Scala operators mimic normal shell operators

A great thing about Scala in this regard is that not only does Scala offer a pipeline operator, it also offers other traditional shell operators. You just have to precede them with the "#" character, like this:

```
#<   Redirect STDIN
#>   Redirect STDOUT
#>>  Append to STDOUT
#&&  Create an AND list
#!!  Create an OR list
```

Chapter 8

Apache Spark

8.1 Getting started with the Spark Shell

8.1.1 Introduction

Apache Spark is a Scala library for analysing "big data". It can be used for analysing huge (internet-scale) datasets distributed across large clusters of machines. The analysis can be anything from the computation of simple descriptive statistics associated with the datasets, through to rather sophisticated machine learning pipelines involving data preprocessing, transformation, nonlinear model fitting and regularisation parameter tuning (via methods such as cross-validation). A relatively impartial overview can be found in the Apache Spark Wikipedia page.

Although Spark is really aimed at data that can't easily be analysed on a laptop, it is actually very easy to install and use (in standalone mode) on a laptop, and a good laptop with a fast multicore processor and plenty of RAM is fine for datasets up to a few gigabytes in size. This chapter will walk through getting started with Spark, installing it locally (not requiring admin/root access) doing some simple descriptive analysis, and moving on to fit simple regression models to some simulated data. After this it should be relatively easy to take things further by reading the Spark documentation, which is generally pretty good.

Anyone who is interested in learning more about setting up and using Spark clusters may want to have a quick look over on my personal blog [11)] (mainly concerned with the Raspberry Pi), where I have previously considered installing Spark on a Raspberry Pi 2, setting up a small Spark cluster, and setting up a larger Spark cluster. Although these posts are based around the Raspberry Pi, most of the material there is quite generic, since the Raspberry Pi is just a small (Debian-based) Linux server.

8.1.2 Getting started – installing Spark

The only pre-requisite for installing Spark is a recent Java installation. Once you have Java installed, you can download and install Spark in any appropriate place in your file-system. If you are running Linux, or a Unix-alike, unpack and test your download with the following commands:

```
tar xvfz spark-2.2.0-bin-hadoop2.7.tgz
cd spark-2.2.0-bin-hadoop2.7
bin/run-example SparkPi 10
```

If all goes well, the last command should run an example. Don't worry if there are lots of INFO and WARN messages – we will sort that out shortly. On other systems it should simply be a matter of downloading and

unpacking Spark somewhere appropriate, then running the example from the top-level Spark directory. Get Spark from the downloads page. You should get version 2.2.0 built for Hadoop 2.7. It doesn't matter if you don't have Hadoop installed — it is not required for single-machine use.

The INFO messages are useful for debugging cluster installations, but are too verbose for general use. On a Linux system you can turn down the verbosity with:

```
sed 's/rootCategory=INFO/rootCategory=WARN/g' < \
conf/log4j.properties.template > \
conf/log4j.properties
```

On other systems, copy the file log4j.properties.template in the conf sub-directory to log4j.properties and edit the file, replacing INFO with WARN on the relevant line. Check it has worked by re-running the SparkPi example – it should be much less verbose this time. You can also try some other examples:

```
bin/run-example SparkLR
ls examples/src/main/scala/org/apache/spark/examples/
```

The "4" refers to the number of worker threads to use. Four is probably fine for most decent laptops. Ctrl-D or :quit will exit the Spark shell and take you back to your OS shell. It is more convenient to have the Spark bin directory in your path. If you are using bash or a similar OS shell, you can temporarily add the Spark bin to your path with the OS shell command:

```
export PATH=$PAT:‘pwd‘/bin
```

You can make this permanent by adding a line like this (but with the full path hard-coded) to your .profile or similar start-up dot-file. I prefer not to do this, as I typically have several different Spark versions on my laptop and want to be able to select exactly the version I need. If you are not running bash, Google how to add a directory to your path. Check the path update has worked by starting up a shell with:

```
$ spark-shell --master "local[4]"
```

Note that if you want to run a script containing Spark commands to be run in "batch mode", you could do it with a command like:

```
$ spark-shell --driver-memory 25g --master "local[4]" < \
   spark-script.scala | tee script-out.txt
```

There are much better ways to develop and submit batch jobs to Spark clusters, but we will discuss these later. Note that while Spark is running, diagnostic information about the "cluster" can be obtained by pointing a web browser port 4040 on the master, which here is just http://localhost:4040/ – this is extremely useful for debugging purposes.

The Spark shell is essentially just a Scala REPL with a dependency on Spark and a couple of pre-defined values.

> **Note:** To have a spark-shell in the console mode of sbt, the following may be added into the build.sbt file; see Using sbt instead of spark-shell.

```
// See https://sanori.github.io/2017/06/Using-sbt-instead-of-spark-shell/
initialCommands in console := """
  import org.apache.spark.sql.SparkSession
  import org.apache.spark.sql.functions._
```

```
  val spark = SparkSession.builder().
    master("local").
    appName("spark-shell").
    getOrCreate()
  import spark.implicits._
  val sc = spark.sparkContext
"""
cleanupCommands in console := "spark.stop()"
```

8.1.3 First Spark shell commands

Counting lines in a file

We are now ready to start using Spark. From a Spark shell in the toplevel directory, enter:

```
//Spark context Web UI available at http://127.0.0.1:4040
//Spark context available as 'sc' (master = local[*], app id = local-1526182800772).
//Spark session available as 'spark'.
//Welcome to
//      ____              __
//     / __/__  ___ _____/ /__
//    _\ \/ _ \/ _ `/ __/  '_/
//   /___/ .__/\_,_/_/ /_/\_\   version 2.1.1
//      /_/
//
//Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_171)
//Type in expressions to have them evaluated.
//Type :help for more information.
sc.textFile("README.md").count
//res0: Long = 104
```

If all goes well, you should get a count of the number of lines in the file "README.md". The value sc is the "Spark context", containing information about the Spark cluster (here it is just a laptop, but in general it could be a large cluster of machines, each with many processors and each processor with many cores). The **textFile** method loads up the file into an RDD (Resilient Distributed Dataset). **The RDD is the fundamental abstraction provided by Spark.** It is a lazy distributed parallel immutable monadic collection. After loading a text file like this, each element of the collection represents one line of the file. The point is that although RDDs are potentially huge and distributed over a large cluster, using them is very similar to using any other monadic collection in Scala. We can unpack the previous command slightly as follows:

```
val rdd1 = sc.textFile("README.md")
//rdd1: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[3] at textFile
rdd1
//res1: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[3] at textFile
rdd1.count
//res2: Long = 104
```

Note that RDDs are "lazy" (unlike most other Scala collections), and this is important for optimising complex pipelines. So here, after assigning the value **rdd1**, no data is actually loaded into memory. All of the actual

computation is deferred until an "action" is called – **count** is an example of such an action, and therefore triggers the loading of data into memory and the counting of elements.

Counting words in a file

We can now look at a very slightly more complex pipeline – counting the number of words in a text file rather than the number of lines. This can be done as follows:

```
sc.textFile("README.md").
  map(_.trim).
  flatMap(_.split( ' ' )).
  count
//res3: Long = 536
```

Note that map and flatMap are both lazy ("transformations" in Spark terminology), and so no computation is triggered until the final action, count is called. The call to map will just trim any redundant white-space from the line ends. So after the call to map the RDD will still have one element for each line of the file. However, the call to flatMap splits each line on white-space, so after this call each element of the RDD will correspond to a word, and not a line. So, the final count will again count the number of elements in the RDD, but here this corresponds to the number of words in the file.

Counting character frequencies in a file

A final example before moving on to look at quantitative data analysis: counting the frequency with which each character occurs in a file. This can be done as follows:

```
val cc = sc.textFile("README.md").
  map(_.toLowerCase).
  flatMap(_.toCharArray).
  map{(_,1)}.
  reduceByKey(_+_)
//cc: org.apache.spark.rdd.RDD[(Char, Int)] = ShuffledRDD[35] at reduceByKey at <con
cc.collect
//res4: Array[(Char, Int)] = Array((d,97), (z,3), ...
```

The first call to **map** converts upper case characters to lower case, as we don't want separate counts for upper and lower case characters. The call to **flatMap** then makes each element of the RDD correspond to a single character in the file. The second call to **map** transforms each element of the RDD to a key-value pair, where the key is the character and the value is the integer 1. RDDs have special methods for key-value pairs in this form – the method **reduceByKey** is one such – it applies the reduction operation (here just +) to all values corresponding to a particular value of the key. Since each character has the value 1, the sum of the values will be a character count. Note that the reduction will be done in parallel, and for this to work it is vital that the reduction operation is associative. Simple addition of integers is clearly associative, so here we are fine. Note that **reduceByKey** is a (lazy) transformation, and so the computation needs to be triggered by a call to the action **collect**.

On most Unix-like systems there is a file called words that is used for spell-checking. The example below applies the character count to this file. Note the calls to **filter**, which filter out any elements of the RDD not matching the predicate. Here it is used to filter out special characters. We also sort the results by value (rather than by key), using **false** to indicate descending order.

```
val cf = sc.textFile("/usr/share/dict/words").
  map(_.toLowerCase).
  flatMap(_.toCharArray).
  filter(_ > '/').
  filter(_ < '}').
  map{(_,1)}.
  reduceByKey(_+_).
  sortBy(_._2, false)
//cf: org.apache.spark.rdd.RDD[(Char, Int)] = MapPartitionsRDD[37] at sortBy at <con
cf.collect
//res10: Array[(Char, Int)] = Array((s,90113), (e,88833), (i,66986), (a,64439), (r,5
```

8.1.4 Caching RDDs with persist

Before continuing further, it is worth dwelling briefly on issues of resilience, re-computation and persistence. The RDD is resilient in the sense that it knows how to re-compute itself if (say) one node of the Spark cluster fails. Spark computations are largely sequences of lazy transformations, and these computations can be "re-played" if necessary. But care must be taken not to unnecessarily re-do computations that have already been done when it is not necessary. For example, suppose that **rdd** is an **RDD[Double]**, resulting from a long chain of transformations, starting with loading a huge file from disk. Creating the rdd is near-instantaneous, since it is a sequence of transformations, and so no actual computation will take place. But now suppose that there are a couple of different actions we want to call on this RDD. First, we would like to know how many elements are in the RDD, so we call **rdd.count**. At this point the computation is triggered, and it could take a long time to get our result. But now suppose that we also want to know the sum of the elements in the RDD, and call **rdd.sum**. We have again triggered a computation by calling an action, but potentially, we will re-execute the entire computational pipeline which results in **rdd**, which we have already done to get our count. This is clearly very unsatisfactory.

The lazy transformation **persist** is used to cache a snapshot of the current state of an RDD, precisely to avoid this kind of unnecessary recomputation. So if in forming our **rdd**, the final transformation used was **persist**, then when the actual content of the RDD is computed when count triggers it, the state of **rdd** will be cached, so that when the new action **sum** is called, computation of **rdd** does not have to begin again from scratch. These lazy evaluation semantics take a bit of getting used to, since they are somewhat different to regular Scala collections. Knowing when or whether to persist an RDD is a common source of confusion for Spark beginners. But note that the lazy semantics are very powerful, as they allow analysis and optimisation of the entire computational pipeline, and can help to speed up computations by minimising shuffling between nodes of a Spark cluster.

8.2 Commonly used RDD methods

Remember that **transformations** are lazy and **actions** are strict. An action must be called to trigger computation. Methods returning an RDD are transformations and methods returning other types are actions.

8.2.1 Transformations of an RDD[T]

```
def ++(other: RDD[T]): RDD[T]
def cartesian[U](other: RDD[U]): RDD[(T, U)]
```

```
def distinct: RDD[T]
def filter(f: (T) => Boolean): RDD[T]
def flatMap[U](f: (T) => TraversableOnce[U]): RDD[U]
def map[U](f: (T) => U): RDD[U]
def persist: RDD[T]
def sample(withReplacement: Boolean,
  fraction: Double): RDD[T]
def sortBy[K](f: (T) => K,
  ascending: Boolean = true): RDD[T]
def zip[U](other: RDD[U]): RDD[(T, U)]
```

8.2.2 Actions on an RDD[T]

```
def aggregate[U](zeroValue: U)(seqOp: (U, T) => U,
  combOp: (U, U) => U): U
def collect: Array[T]
def count: Long
def fold(zeroValue: T)(op: (T, T) => T): T
def foreach(f: (T) => Unit): Unit
def reduce(f: (T, T) => T): T
def take(num: Int): Array[T]
```

8.2.3 Transformations of a Pair RDD, RDD[(K,V)]

```
def aggregateByKey[U](zeroValue: U)(seqOp: (U, V) => U,
  combOp: (U, U) => U): RDD[(K, U)]
def groupByKey(): RDD[(K, Iterable[V])]
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
def keys: RDD[K]
def mapValues[U](f: (V) => U): RDD[(K, U)]
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
def values: RDD[V]
```

8.2.4 Actions on a RDD[(K,V)]

```
def countByKey: Map[K, Long]
```

8.2.5 DataFrames

```
def toDF(colNames: String*): DataFrame
def col(colName: String): Column
def drop(col: Column): DataFrame
def select(cols: Column*): DataFrame
def show(numRows: Int): Unit
def withColumn(colName: String, col: Column): DataFrame
def withColumnRenamed(existingName: String,
```

```
    newName: String): DataFrame
```

8.3 Analysis of quantitative data

8.3.1 Descriptive statistics

We first need some quantitative data, so let's simulate some using Breeze. Spark has a dependence on Breeze, and therefore can be used from inside the Spark shell – this is very useful. So, we start by using Breeze to simulate a vector of normal random quantities:

```
import breeze.stats.distributions._
//import breeze.stats.distributions._
val x = Gaussian(1.0,2.0).sample(10000)
//x: IndexedSeq[Double] = Vector(2.941854758174765, -1.8525483998731618, ...
```

Note, though, that x is just a regular Breeze Vector, a simple serial collection all stored in RAM on the master thread. To use it as a Spark RDD, we must convert it to one, using the parallelize function:

```
val xRdd = sc.parallelize(x)
//xRdd: org.apache.spark.rdd.RDD[Double] = ParallelCollectionRDD[0] at parallelize a
```

Now **xRdd** is an RDD, and so we can do Spark transformations and actions on it. There are some special methods for RDDs containing numeric values:

```
xRdd.mean
//res0: Double = 0.981790180441237
xRdd.sampleVariance
//res1: Double = 4.041647277060633
```

Each summary statistic is computed with a single pass through the data, but if several summary statistics are required, it is inefficient to make a separate pass through the data for each summary, so the **stats** method makes a single pass through the data returning a **StatsCounter** object that can be used to compute various summary statistics.

```
val xStats = xRdd.stats
//xStats: org.apache.spark.util.StatCounter = (count: 10000, mean: 0.981790,
//  stdev: 2.010284, max: 8.798628, min: -8.102180)
xStats.mean
//res2: Double = 0.9817901804412371
xStats.sampleVariance
//res3: Double = 4.041647277060633
xStats.sum
//res4: Double = 9817.901804412371
```

The StatsCounter methods are: count, mean, sum, max, min, variance, sampleVariance, stdev, sampleStdev.

8.3.2 Linear regression

Moving beyond very simple descriptive statistics, we will next look at a simple linear regression model, which will also allow us to introduce Spark **DataFrame**s – a high level abstraction layered on top of RDDs which makes working with tabular data much more convenient, especially in the context of statistical modelling.

We start with some standard (non-Spark) Scala Breeze code to simulate some data from a simple linear regression model. We use the **x** already simulated as our first covariate. Then we simulate a second covariate, **x2**. Then, using some residual noise, **eps**, we simulate a regression model scenario, where we know that the "true" intercept is 1.5 and the "true" covariate regression coefficients are 2.0 and 1.0.

```
val x2 = Gaussian(0.0,1.0).sample(10000)
//x2: IndexedSeq[Double] = Vector(0.7832284218438778, 0.16793714271280957, ...
val xx = x zip x2
//xx: IndexedSeq[(Double, Double)] = Vector((2.941854758174765,0.7832284218438778),
val lp = xx map {p => 2.0*p._1 + 1.0*p._2 + 1.5}
//lp: IndexedSeq[Double] = Vector(8.166937938193406, -2.0371596570335138, ...
val eps = Gaussian(0.0,1.0).sample(10000)
//eps: IndexedSeq[Double] = Vector(1.8532152388587169, -0.14118377628968212, ...
val y = (lp zip eps) map (p => p._1 + p._2)
//y: IndexedSeq[Double] = Vector(10.020153177052123, -2.178343433323196, ...
val yx = (y zip xx) map (p => (p._1,p._2._1,p._2._2))
//yx: IndexedSeq[(Double, Double, Double)] = Vector((10.020153177052123, ...
val rddLR = sc.parallelize(yx)
//rddLR: org.apache.spark.rdd.RDD[(Double, Double, Double)] = ParallelCollectionRDD[
```

Note that the last line converts the regular Scala Breeze collection into a Spark RDD using **parallelize**. We could, in principle, do regression modelling using raw RDDs, and early versions of Spark required this. However, statisticians used to statistical languages such as R know that data frames are useful for working with tabular data. We can convert an RDD of tuples to a Spark **DataFrame** as follows:

```
val dfLR = rddLR.toDF("y", "x1" ,"x2")
//dfLR: org.apache.spark.sql.DataFrame = [y: double, x1: double ... 1 more field]
dfLR.show
//18/05/13 13:44:03 WARN TaskSetManager: Stage 3 contains a task of very large size
//+-------------------+-------------------+--------------------+
//|                  y|                 x1|                  x2|
//+-------------------+-------------------+--------------------+
//| 10.020153177052123|   2.941854758174765|  0.7832284218438778|
//| -2.178343433323196| -1.8525483998731618| 0.16793714271280957|
//| 3.3038672103110405|  1.6070647939250189|  0.3633737381166781|
//...
dfLR.show(5)
//18/05/13 13:46:17 WARN TaskSetManager: Stage 4 contains a task of very large size
//+-------------------+-------------------+-------------------+
//|                  y|                 x1|                 x2|
//+-------------------+-------------------+-------------------+
//| 10.020153177052123|   2.941854758174765| 0.7832284218438778|
```

```
//...
//only showing top 5 rows
//
```

Note that **show** shows the first few rows of a **DataFrame**, and giving it a numeric argument specifies the number to show. This is very useful for quick sanity-checking of **DataFrame** contents.

Note that there are other ways of getting data into a Spark **DataFrame**. One of the simplest ways to get data into Spark from other systems is via a CSV file. A properly formatted CSV file with a header row can be read into Spark with a command like:

```
// Don't run unless you have an appropriate csv file ...
val df = spark.read.
  option("header","true").
  option("inferSchema","true").
  csv("myCsvFile.csv")
```

This requires two passes over the data – one to infer the schema and one to actually read the data. The result is a Spark DataFrame. For very large datasets it is better to declare the schema and not use automatic schema inference. However, for very large datasets, CSV probably isn't a great choice of format anyway. Spark supports many more efficient data storage formats. Note that Spark also has functions for querying SQL (and other) databases, and reading query results directly into **DataFrame** objects. For people familiar with databases, this is often the most convenient way of ingesting data into Spark. See the Spark DataFrames guide and the API docs for DataFrameReader for further information.

Spark has an extensive library of tools for the development of sophisticated machine learning pipelines. Included in this are functions for fitting linear regression models, regularised regression models (Lasso, ridge, elastic net), generalised linear models, including logistic regression models, etc., and tools for optimising regularisation parameters, for example, using cross-validation. Let's start by seeing how to fit a simple OLS (ordinary least squares) linear regression model: see the ML pipeline documentation for further information, especially the docs on classification and regression.

We start by creating an object for fitting linear regression models:

```
import org.apache.spark.ml.regression.LinearRegression
//import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.linalg._
//import org.apache.spark.ml.linalg._
val lm = new LinearRegression
//lm: org.apache.spark.ml.regression.LinearRegression = linReg_3b11eeb155c4
lm.getStandardization
//res8: Boolean = true
lm.setStandardization(false)
//res9: lm.type = linReg_3b11eeb155c4
lm.getStandardization
//res10: Boolean = false
lm.explainParams
//aggregationDepth: suggested depth for treeAggregate (>= 2) (default: 2)
//elasticNetParam: the ElasticNet mixing parameter, in range [0, 1].
//  For alpha = 0, the penalty is an L2 penalty.
//  For alpha = 1, it is an L1 penalty (default: 0.0)
```

```
//featuresCol: features column name (default: features)
//fitIntercept: whether to fit an intercept term (default: true)
//labelCol: label column name (default: label)
//maxIter: maximum number of iterations (>= 0) (default: 100)
//predictionCol: prediction column name (default: prediction)
//regParam: regularization parameter (>= 0) (default: 0.0)
//solver: the solver algorithm for optimization.
//  If this is not set or empty, default value is 'auto' (default: auto)
//standardization: whether to standardize the training features
//  before fitting the model (defa...
```

Note also that **fitIntercept** defaults to **true**, and so you do not need to prepend your design matrix with a column of ones.

Note that there are many parameters associated with the fitting algorithm, including regularisation parameters. These are set to defaults corresponding to no regularisation (simple OLS). Note, however, that the algorithm defaults to standardising covariates to be mean zero variance one. We can turn that off before fitting the model if desired.

Also note that the model fitting algorithm assumes that the **DataFrame** to be fit has (at least) two columns, one called **label** containing the response variable, and one called **features**, where each element is actually a **Vectors** of covariates. So we first need to transform our **DataFrame** into the required format. We can do this directly as follows.

In Spark, **Vector**s is just a type alias for a Breeze **Vector**

```
val dflr = (dfLR map {row => (row.getDouble(0),
  Vectors.dense(row.getDouble(1),
  row.getDouble(2)))}).toDF("label","features")
//dflr: org.apache.spark.sql.DataFrame = [label: double, features: vector]
dflr.show(5)
//+-------------------+--------------------+
//|              label|            features|
//+-------------------+--------------------+
//| 10.020153177052123|[2.94185475817476...|
//| -2.178343433323196|[-1.8525483998731...|
//| 3.3038672103110405|[1.60706479392501...|
//|-1.4752986008440727|[-0.1268283519730...|
//|  3.479401710647615|[0.40096090205000...|
//+-------------------+--------------------+
//only showing top 5 rows
```

However, the simplest way to coerce the data into this format is using an R-style model formula, **RFormula**, as follows.

```
import org.apache.spark.ml.feature.RFormula
val dflr2 = new RFormula().
  setFormula("y ~ x1 + x2"). \\or setFormula("y ~ .").
  fit(dfLR).transform(dfLR).
  select("label","features")
```

```
//dflr2: org.apache.spark.sql.DataFrame = [label: double, features: vector]
dflr2 show 5
//18/05/13 14:33:26 WARN TaskSetManager: Stage 6 contains a task of very large size
//+------------------+--------------------+
//|             label|            features|
//+------------------+--------------------+
//| 10.020153177052123|[2.94185475817476...|
//| -2.178343433323196|[-1.8525483998731...|
//| 3.3038672103110405|[1.60706479392501...|
//|-1.4752986008440727|[-0.1268283519730...|
//|  3.479401710647615|[0.40096090205000...|
//+------------------+--------------------+
//only showing top 5 rows
```

Now we have the data in the correct format, it is simple to fit the model and look at the estimated parameters.

```
val fit = lm.fit(dflr)
//fit: org.apache.spark.ml.regression.LinearRegressionModel = linReg_3b11eeb155c4
fit.intercept
//res14: Double = 1.502320773356445
fit.coefficients
//res15: org.apache.spark.ml.linalg.Vector = [1.9976234367900432,1.0086728598782204]
```

You should see that the estimated parameters are close to the "true" parameters that were used to simulate from the model. More detailed diagnostics can be obtained from the fitted summary object.

```
val summ = fit.summary
//summ: org.apache.spark.ml.regression.LinearRegressionTrainingSummary = org.apache.
summ.r2
//res16: Double = 0.9446201070352793
summ.rootMeanSquaredError
//res17: Double = 1.0047661770289757
summ.coefficientStandardErrors
//res18: Array[Double] = Array(0.004999018041574145, 0.009974614589424745, 0.0111836
summ.pValues
//res19: Array[Double] = Array(0.0, 0.0, 0.0)
summ.tValues
//res20: Array[Double] = Array(399.60316609719814, 101.12399339696117, 134.332202590
summ.predictions
//res21: org.apache.spark.sql.DataFrame = [label: double, features: vector ... 1 mor
summ.residuals
//res22: org.apache.spark.sql.DataFrame = [residuals: double]
```

So, that's how to fit a simple OLS linear regression model.

8.3.3 Logistic regression

To illustrate the fitting of logistic regression models, we will again use synthetic data, and start from exactly the same linear predictor as we used in the linear regression analysis. But this time we will push it through a sigmoid and simulate some binary outcomes.

```
val p = lp map (x => 1.0/(1.0+math.exp(-x)))
//p: IndexedSeq[Double] = Vector(0.9997161946050895, 0.11535627025535045, ...
val yl = p map (pi => new Binomial(1,pi).draw) map (_.toDouble)
//yl: IndexedSeq[Double] = Vector(1.0, 0.0, 1.0, 1.0, 1.0, ...
val yxl = (yl zip xx) map (p => (p._1,p._2._1,p._2._2))
//yxl: IndexedSeq[(Double, Double, Double)] = Vector((1.0,2.941854758174765, ...
val rddLogR = sc.parallelize(yxl)
//rddLogR: org.apache.spark.rdd.RDD[(Double, Double, Double)] =
//  ParallelCollectionRDD[39] at parallelize at <console>:46
val dfLogR = rddLogR.toDF("y", "x1", "x2").persist
//dfLogR: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [y: double, x1: d
dfLogR.show(5)
//+---+-------------------+-------------------+
//|  y|                 x1|                 x2|
//+---+-------------------+-------------------+
//|1.0|   2.941854758174765| 0.7832284218438778|
//|0.0| -1.8525483998731618|0.16793714271280957|
//|1.0|  1.6070647939250189| 0.3633737381166781|
//|1.0|-0.12682835197305442|-1.8476876099117105|
//|1.0| 0.40096090205000967|  0.819815598651938|
//+---+-------------------+-------------------+
//only showing top 5 rows
```

Now that we have our synthetic logistic regression data, we can fit it in a fairly similar way to simple linear regression.

```
import org.apache.spark.ml.classification._
//import org.apache.spark.ml.classification._
val lr = new LogisticRegression
//lr: org.apache.spark.ml.classification.LogisticRegression = logreg_2ee870c66fe8
lr.setStandardization(false)
//res24: lr.type = logreg_2ee870c66fe8
lr.explainParams
//res25: String =
//aggregationDepth: suggested depth for treeAggregate (>= 2) (default: 2)
//  elasticNetParam: the ElasticNet mixing parameter, in range [0, 1].
//  For alpha = 0, the penalty is an L2 penalty.
//  For alpha = 1, it is an L1 penalty (default: 0.0)
//family: The name of family which is a description of the label distribution
//  to be used in the model. Supported options: auto, binomial, multinomial. (defaul
//featuresCol: features column name (default: features)
//fitIntercept: whether to fit an intercept term (default: true)
//labelCol: label column name (default: label)
//maxIter: maximum number of iterations (>= 0) (default: 100)
//predictionCol: prediction column name (default: prediction)
//probabilityCol: Column name for predicted class conditional probabilities.
//  Note: Not all models outp...
```

```
//
val dflogr = new RFormula().
  setFormula("y ~ x1 + x2").
  fit(dfLogR).transform(dfLogR).
  select("label", "features")
//dflogr: org.apache.spark.sql.DataFrame = [label: double, features: vector]
dflogr.show(5)
//+-----+--------------------+
//|label|            features|
//+-----+--------------------+
//|  1.0|[2.94185475817476...|
//|  0.0|[-1.8525483998731...|
//|  1.0|[1.60706479392501...|
//|  1.0|[-0.1268283519730...|
//|  1.0|[0.40096090205000...|
//+-----+--------------------+
//only showing top 5 rows
//
val logrfit = lr.fit(dflogr)
//logrfit: org.apache.spark.ml.classification.LogisticRegressionModel = logreg_2ee87
logrfit.intercept
//res27: Double = 1.5234572152639736
logrfit.coefficients
//res28: org.apache.spark.ml.linalg.Vector = [2.0426091319230233,1.018954193115733]
```

So we see that we are able to recover the regression coefficients that we used to simulate the data fairly well. Additional diagnostics can be obtained, similar to the linear regression case.

8.4 Tuning regularisation parameters using crossvalidation

We have seen that there are many possible tuning constants associated with linear and logistic regression models in Spark. In particular, the models allow regularisation using elastic net penalties (which includes ridge and lasso regression as special cases). But then there is a question as to how to set the tuning parameters. One way is to search over a grid of possible tuning parameters and see which give the best predictive performance under k-fold cross-validation. The example below illustrates this for a ridge penalty in our logistic regression example, searching over a grid of 60 different ridge parameters and evaluating with 8-fold cross-validation.

```
import breeze.linalg.linspace
val lambdas = linspace(-12,4,60).
  toArray.
  map{math.exp(_)}
//lambdas: Array[Double] = Array(6.14421235332821E-6, 8.058254732499574E-6,
import org.apache.spark.ml.tuning._
//import org.apache.spark.ml.tuning._
import org.apache.spark.ml.evaluation._
```

```
//import org.apache.spark.ml.evaluation._
val paramGrid = new ParamGridBuilder().
  addGrid(lr.regParam,lambdas).
  build()
//paramGrid: Array[org.apache.spark.ml.param.ParamMap] =
//Array({
//        logreg_7fd75830611e-regParam: 6.14421235332821E-6
// ...
val cv = new CrossValidator().
  setEstimator(lr).
  setEvaluator(new BinaryClassificationEvaluator).
  setEstimatorParamMaps(paramGrid).
  setNumFolds(8)
//cv: org.apache.spark.ml.tuning.CrossValidator = cv_7351d5b7e155
val cvMod = cv.fit(dflogr)
//cvMod: org.apache.spark.ml.tuning.CrossValidatorModel = cv
cvMod.explainParams
//res31: String =
//estimator: estimator for selection (current: logreg_7fd75830611e)
//estimatorParamMaps: param maps for the estimator
//  (current: [Lorg.apache.spark.ml.param.ParamMap;@517f6d94)
//evaluator: evaluator used to select hyper-parameters that
//  maximize the validated metric (current: binEval_0af44f4a45ad)
//numFolds: number of folds for cross validation (>= 2)
//  (default: 3, current: 8)
//seed: random seed (default: -1191137437)
cvMod.bestModel.explainParams
//res32: String =
//aggregationDepth: suggested depth for treeAggregate (>= 2) (default: 2)
//elasticNetParam: the ElasticNet mixing parameter, in range [0, 1].
//  For alpha = 0, the penalty is an L2 penalty.
//  For alpha = 1, it is an L1 penalty (default: 0.0)
//family: The name of family which is a description of the label distribution
//  to be used in the model. Supported options: auto, binomial, multinomial. (defaul
//featuresCol: features column name (default: features)
//fitIntercept: whether to fit an intercept term (default: true)
//labelCol: label column name (default: label)
//maxIter: maximum number of iterations (>= 0) (default: 100)
//predictionCol: prediction column name (default: prediction)
//probabilityCol: Column name for predicted class conditional
//  probabilities. Note: Not all models outp...
cvMod.bestModel.extractParamMap
//res33: org.apache.spark.ml.param.ParamMap =
//{
//        logreg_7fd75830611e-aggregationDepth: 2,
//        logreg_7fd75830611e-elasticNetParam: 0.0,
```

```
//        logreg_7fd75830611e-family: auto,
//        logreg_7fd75830611e-featuresCol: features,
//        logreg_7fd75830611e-fitIntercept: true,
//        logreg_7fd75830611e-labelCol: label,
//        logreg_7fd75830611e-maxIter: 100,
//        logreg_7fd75830611e-predictionCol: prediction,
//        logreg_7fd75830611e-probabilityCol: probability,
//        logreg_7fd75830611e-rawPredictionCol: rawPrediction,
//        logreg_7fd75830611e-regParam: 6.14421235332821E-6,
//        logreg_7fd75830611e-standardization: false,
//        logreg_7fd75830611e-threshold: 0.5,
//        logreg_7fd75830611e-tol: 1.0E-6
//}
val lambda = cvMod.bestModel.
  extractParamMap.
  getOrElse(cvMod.bestModel.
  getParam("regParam"),0.0).
  asInstanceOf[Double]
//lambda: Double = 6.14421235332821E-6
```

These big grid-search jobs can take a long time to run.

The most important output is the optimal regularisation parameter, which for this dataset was around 2.74×10^{-4}. Generally speaking it is better to standardise the predictors for tuning and fitting elastic net models (since you want the shrinkage to be on the same scale for each variable). It also is important to fit the intercept as we have done here, since you don't want to shrink the intercept along with the other regression coefficients.

8.5 Compiling Spark jobs

Working interactively in the Spark Shell can be very convenient for learning about how Spark works, similarly to how it can be convenient to explore Scala by using the Scala REPL interactively. But for significant work, there are many advantages to writing code in an intelligent IDE and compiling using SBT. Exactly the same is true for Spark, so once you have the idea of how it works, you will want to start developing Spark code as you do other Scala applications, using the same tool chain, and regarding Spark to some extent as just another third-party Scala library.

A complete example of how to set up an SBT project for building Spark applications is provided in the course repo. But essentially, the SBT build file should look something like:

```
name := "Spark-template"

version := "0.1"

libraryDependencies  ++= Seq(
 "org.apache.spark" %% "spark-core" % "2.3.0",
 "org.apache.spark" %% "spark-sql" % "2.3.0"
)
```

```
scalaVersion := "2.11.12"
```

Note that Spark 2.2.0 is not compatible with Scala 2.12. Also note that if you are using MLLib, then you will need an additional dependence on "spark-mllib"; spark.mllib is obsolete, so instead use spark.ml.

The following is a complete example application:

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SparkApp {

  def main(args: Array[String]): Unit = {

    val sparkConf = new SparkConf().
      setAppName("Spark Application").
      setMaster("local[4]")
    val sc = new SparkContext(sparkConf)

    sc.textFile("/usr/share/dict/words").
      map(_.trim).
      map(_.toLowerCase).
      flatMap(_.toCharArray).
      filter(_ > '/' ).
      filter(_ < 'g' ).
      map{(_,1)}.
      reduceByKey(_+_).
      sortBy(_._2,false).
      collect.
      foreach(println)

    sc.stop()
  }
}
//(e,88833)
//(a,64439)
//(c,31872)
//(d,28531)
//(b,15526)
//(f,10675)
```

So, you just need a few imports, then you must start by creating a **SparkConf** and giving your application an appropriate name, and then create the **SparkContext, sc**. This Spark context can then be used in the same way as the one that is created for you when you start up a Spark shell.

You can use SBT to build your application with **sbt** package. This will place the compiled JAR in **target/scala-2.11/**. This JAR can be submitted to a Spark cluster using **spark-submit**. For example, this application could be submitted to the local standalone Spark cluster with a command like:

```
$ spark-submit --class "SparkAppl" \
--master "local[4]" \
target/scala-2.11/spark-template_2.11-0.1.jar
```

Note that you can include additional third party libraries in your build in the usual way, but in this case you will need to build and submit an assembly JAR to the Spark cluster.

8.5.1 Further reading

As previously mentioned, once you are up and running with a Spark shell, the official Spark documentation is reasonably good. First go through the quick start guide, then the programming guide, then the ML guide, and finally, consult the API docs [12)].

Chapter 9

Advanced topics

9.1 Typeclasses and implicits

Scala supports an advanced programming feature called "implicits", which is essentially a mechanism for automatically (implicitly) providing arguments for functions without requiring the function caller to pass the value explicitly. The value used must be be of the correct type, and must have already been declared and "in scope" (exactly how implicits are resolved is non-trivial, and beyond the scope of this course). There are many possible applications of implicits, but in Scala they are most often used as the mechanism by which the "typeclass" pattern can be implemented. Typeclasses are a powerful device for implementing "ad hoc polymorphism" popularised by Haskell; note. They are a (arguably superior) alternative to traditional inheritance based O-O approaches to polymorphism. Unlike Haskell, Scala does not have first class language support for typeclasses, but the available implicit mechanisms are powerful enough to support the pattern indirectly. This probably all sounds like terrifying CS theory. In fact it is all very practical and useful, but is most easily understood by looking at a few examples.

9.1.1 Using implicits to add a method to a pre-existing class

Scala's **.sum** method magically works on any Scala collection parametrised by a numeric type.

```
Vector(1,2,3).sum
//res0: Int = 6
List(1.0,5.0).sum
//res1: Double = 6.0
```

This is great, but there isn't a corresponding **.mean** method.

```
Vector(1,2,3).mean
//<console>:12: error: value mean is not a member of
//  scala.collection.immutable.Vector[Int]
```

Is it possible to add one? The answer is "yes", but before diving in to the details, think about how difficult this would be to do cleanly in a typical O-O language like Java. You can't go in to the Scala source code for the standard library and retrospectively include a **.mean** method. You would probably be forced to define a new abstract class (or interface) which does implement a **.mean** method, and then define new classes inheriting from the old collections that also implement the new interface, and then you'd be forced to work with the new subclass wherever you needed the **.mean** methods. So there'd be lots of converting and casting back and

forth between the original and new classes, and it would all be very unsatisfactory and not very extensible or inter-operable ...

We have already seen (in Chapter 3) how to define a mean function which accepts a numeric collection as input and returns a mean as output. Let's define this again now, but wrap it in an object to keep it out of scope.

```
object Meanable {
  def mean[T: Numeric](it: Iterable[T]): Double =
    it.map(implicitly[Numeric[T]].toDouble(_)).sum / it.size
}
//defined object Meanable
```

We will now try to better understand how this works and how to go one step further and define a **.mean** method (on the **Iterable** collection). The *context bound* on the type parameter **T** (the **:** notation) is just syntactic sugar for the addition of an implicit parameter. So the above could have been written equivalently as:

```
object Meanable {
  def mean[T: Numeric](it: Iterable[T]) (
    implicit num: Numeric[T]): Double =
    it.map(num.toDouble(_)).sum / it.size
}
//defined object Meanable
```

Recall that we could implement this a bit more cleanly using *Spire*, but we will stick to vanilla Scala here

Note: By using a typeclass Numeric that is defined in scala.math and an implicit class, we can more cleanly describe the method.

```
implicit class NumericSyntax[T](numeric: T) {
  def toDouble(implicit num: Numeric[T]) = num.toDouble(numeric)
}
//defined class NumericSyntax
object Meanable {
  def mean[T: Numeric](it: Iterable[T]): Double = {
it.map(x => x.toDouble).sum / it.size
  }
//mean: [T](it: Iterable[T])(implicit evidence$1: Numeric[T])Double
}
//defined object Meanable
```

Note: Actually, such an implicit class is defined in the spire librray.

```
import spire.math._
//import spire.math._
import spire.implicits._
//import spire.implicits._
```

```
object Meanable {
  def mean[T: Numeric](it: Iterable[T]): Double = {
    it.map(x => x.toDouble).sum / it.size
}
//defined object Meanable
```

So when the mean function is called, there will be a search (at compile time) for an implicit value of type **Numeric[T]**. If such a value can be found, it will be passed in implicitly, and made available in the body of the function as num. In the first version of the code, we don't bind a name to the implicit value, but we can refer to it using implicitly. For every numeric type **T**, a function **.toDouble()** is defined, and so the code will compile. We can use this as follows:

```
import Meanable._
mean(Vector(1,2,3))
//res4: Double = 2.0
mean(List(1.0,5.0))
//res5: Double = 3.0
```

So far so good, but we have already seen how to do this. What about adding a method? For this, we need to use an *implicit class*:

```
implicit class MeanableInstance[T: Numeric](
  it: Iterable[T]) {
  def mean[T] = Meanable.mean(it)
}
```

This says that any instance of an iterable class over a numeric type should be regarded as an instance of the class **MeanableInstance**. But **MeanableInstance** has a **.mean** method defined in terms of the **Meanable.mean** function we have already defined. We can then use this as follows:

```
Vector(1,2,3).mean
//res4: Double = 2.0
List(1.0,3.0,5.0,7.0).mean
//res5: Double = 4.0
```

So this shows that implicits can do powerful things, but in this example we've somewhat skirted around the typeclass issue, so it's worth walking through another example, this time explicitly considering the typeclass.

9.1.2 Defining and using a simple typeclass

In statistical computing we quite often want to output data in CSV format. Suppose, for example, that we have an MCMC algorithm with multivariate state, and that we want to serialise the MCMC iterations as rows in a CSV file. It would be useful to define a **.toCsv** method on the state which serialises the state as a **String** representing one row of a CSV file. If we always define our state type ourselves, we could ensure that our state always included a **.toCsv** method. But what if we sometimes like to use a **Vector[Double]** as our state (or a Breeze **DenseVector[Double]**)? We are then back to the same problem as we just considered. A good way to deal with this kind of problem is using typeclasses. First, define the typeclass itself.

```
trait CsvRow[T] {
  def toCsv(row: T): String
}
```

This is our typeclass. Superficially, this is a parametrised type with a **toCsv** function, **T =>String**. But as a typeclass, we interpret this as meaning that the type **T** belongs to the **CsvRow** typeclass if it has a **toCsv** method. As before, we have the issue that in the typeclass we have a method which accepts a value of type **T** and returns a String, but we actually just want a method on **T** which returns a **String**. We solve this using an implicit class, as before.

```
implicit class CsvRowSyntax[T](row: T) {
  def toCsv(implicit inst: CsvRow[T]) = inst.toCsv(row)
}
```

Now that we have our typeclass and its associated syntax, we can define functions that are generic over any type **T** belonging to our typeclass.

```
def printRows[T: CsvRow](it: Iterable[T]): Unit =
  it.foreach(row => println(row.toCsv))
```

The context bound notation often leads to more readable code. Here we just read this as the type T belonging to the**CsvRow** typeclass, and can ignore the fact that this is actually shorthand for the addition of an implicit parameter. Let's see how we can define instances of this typeclass. First let's suppose that we have our own state type, **MyState**.

```
case class MyState(x: Int, y: Double)
```

This doesn't currently have a **.toCsv** method, but we can declare one at the same time as declaring it to be an instance of the **CsvRow** typeclass.

```
implicit val myStateCsvRow = new CsvRow[MyState] {
  def toCsv(row: MyState) = row.x.toString+","+row.y
}
```

Now we have declared it to be a member of the typeclass, we can use it as follows.

```
MyState(1,2.0).toCsv
//res6: String = 1,2.0
printRows(List(MyState(1,2.0),MyState(2,3.0)))
//1,2.0
//2,3.0
```

Although we can do this, we could have just defined a **.toCsv** method directly on our **MyState** class, which obviously would have been a bit easier. But we can't easily add methods to classes in the Scala standard library. So now suppose that we would like to use a **Vector[Double]** to represent state. We can just as easily declare a typeclass instance for this, and use it as follows.

```
implicit val vectorDoubleCsvRow =
  new CsvRow[Vector[Double]] {
  def toCsv(row: Vector[Double]) = row.mkString(",")
}
//vectorDoubleCsvRow: CsvRow[Vector[Double]] = $anon$1@333a44f2
Vector(1.0,2.0,3.0).toCsv
//res8: String = 1.0,2.0,3.0
printRows(List(Vector(1.0,2.0),Vector(4.0,5.0),
```

```
  Vector(3.0,3.0)))
//1.0,2.0
//4.0,5.0
//3.0,3.0
```

So typeclasses provide a nice basis for the development of flexible, generic code.

9.1.3 A parametrised typeclass, using higher-kinded types

We have just seen an example of how to declare a **Vector[Double]** as belonging to a typeclass. But what if we have an example where we want to declare **Vector[T]** as belonging to a typeclass irrespective of the type parameter **T**. In other words, we want to declare a typeclass over a parametrised type. Since typeclasses are by definition parametrised types, we will need to declare a generic type with a type parameter corresponding to a generic type. It would be impossible to do this in a language like Java, but in Scala we can do it, using another advanced language feature: higher-kinded types. Higher-kinded types are generic types with type parameters that are also generic types. Very few languages support them: Scala and Haskell are the best known languages which do. Again, this probably sounds like advanced CS theory, but again, examples reveal that they are actually very useful in practice.

To illustrate this problem, we will reconsider the problem of "thinning" a **Stream[T]**. We saw previously how to write a function that would thin the Stream, irrespective of the type parameter T, but this was a function taking a **Stream[T]** as input and returning another **Stream[T]** as an output. We did not define it as a new method on the existing Stream class. We will define a typeclass, Thinnable, representing parametrised classes which can be thinned. We do this as follows.

```
import scala.language.higherKinds
trait Thinnable[F[_]] {
  def thin[T](f: F[T], th: Int): F[T]
}
```

Note that we need to explicitly import higherKinds in order to prevent compiler warnings. In declaring the trait we use an underscore to indicate that **F** is generic, but use a type parameter **T** in the declaration of thin, to indicate that the type of the input and output are exactly the same. As usual, we can declare method syntax using an implicit class.

```
implicit class ThinnableSyntax[T,F[T]](value: F[T]) {
  def thin(th: Int)(implicit inst: Thinnable[F]): F[T] =
  inst.thin(value,th)
}
```

Finally, we can declare an instance of our **Thinnable** typeclass for **Stream**.

```
implicit val streamThinnable: Thinnable[Stream] =
  new Thinnable[Stream] {
    def thin[T](s: Stream[T],th: Int): Stream[T] = {
      val ss = s.drop(th-1)
      if (ss.isEmpty) Stream.empty else
        ss.head #:: thin(ss.tail, th)
    }
  }
```

Now that evidence has been provided that Stream can be regarded as belonging to the **Thinnable** typeclass, we can use our new **.thin** method syntax in an intuitive way.

```
Stream.iterate(0)(_ + 1).
  drop(10).
  thin(2).
  take(5).
  toArray
//
```

Typeclasses are a powerful framework for developing flexible generic code. But using them in Scala via implicits involves a little bit of "boiler- plate" code. In particular, the definition of the implicit class providing method syntax for the typeclass seems as though it ought to be possible to automatically generate. In fact it is possible, and this functionality is provided by the library *Simulacrum* [13)].

```
https://github.com/mpilquist/simulacrum
```

This library makes typeclasses more of a first class concept in Scala, and eliminates quite a bit of code. Once you understand typeclasses and implicits, I recommend learning about how to use Simulacrum to make working with typeclasses simpler and more consistent. We don't have time to cover it in detail within this course, but the end-of-chapter exercises will get you started.

9.1.4 Case study: A scalable particle filter in Scala

Introduction

Many modern algorithms in computational Bayesian statistics have at their heart a particle filter or some other sequential Monte Carlo (SMC) procedure. In particular, particle MCMC algorithms use a particle filter in the inner-loop in order to compute a (noisy, unbiased) estimate of the marginal likelihood of the data. These algorithms are often highly computationally intensive, either because the forward model used to propagate the particles is expensive, or because the likelihood associated with each particle/observation is expensive (or both). In this case it is desirable to parallelise the particle filter to run on all available cores of a machine, or in some cases, it would even be desirable to distribute the the particle filter computation across a cluster of machines. Parallelisation is difficult when using the conventional imperative programming languages typically used in scientific and statistical computing, but is much easier using modern functional languages such as Scala. In Scala it is possible to describe algorithms at a higher level of abstraction, so that exactly the same algorithm can run in serial, run in parallel across all available cores on a single machine, or run in parallel across a cluster of machines, all without changing any code. Doing so renders parallelisation a non-issue. Here I'll walk through how to do this for a simple bootstrap particle filter, but the same principle applies for a large range of statistical computing algorithms.

Typeclasses and monadic collections

Many computational tasks in statistics can be accomplished using a sequence of operations on monadic collections. We would like to write code that is independent of any particular implementation of a monadic collection, so that we can switch to a different implementation without changing the code of our algorithm (for example, switching from a serial to a parallel collection). But in strongly typed languages we need to know at compile time that the collection we use has the methods that we require. Typeclasses provide a nice solution to this problem. We can describe a simple typeclass for our monadic collection as follows:

```
trait GenericColl[C[_]] {
  def map[A, B](ca: C[A])(f: A => B): C[B]
  def reduce[A](ca: C[A])(f: (A, A) => A): A
  def flatMap[A, B, D[B] <: GenTraversable[B]](
    ca: C[A])(f: A => D[B]): C[B]
  def zip[A, B](ca: C[A])(cb: C[B]): C[(A, B)]
  def length[A](ca: C[A]): Int
}
```

In the typeclass we just list the methods that we expect our generic collection to provide, but do not say anything about how they are implemented. For example, we know that operations such as **map** and **reduce** can be executed in parallel, but this is a separate concern. We can now write code that can be used for any collection conforming to the requirements of this typeclass. The full code for this example is provided in the associated github repo, and includes the obvious syntax for this typeclass, and typeclass instances for the Scala collections **Vector** and **ParVector**, that we will exploit later in the example.

SIR step for a bootstrap filter

We can now write some code for a single observation update of a bootstrap particle filter.

```
def update[S: State, O: Observation, C[_]: GenericColl](
  dataLik: (S, O) => LogLik, stepFun: S => S
  )(x: C[S], o: O): (LogLik, C[S]) = {
  import breeze.stats.distributions.Poisson
  val xp = x map (stepFun(_))
  val lw = xp map (dataLik(_, o))
  val max = lw reduce (math.max(_, _))
  val rw = lw map (lwi => math.exp(lwi - max))
  val srw = rw reduce (_ + _)
  val l = rw.length
  val z = rw zip xp
  val rx = z flatMap { case (rwi, xpi) =>
    Vector.fill(Poisson(rwi * l / srw).draw)(xpi) }
  (max + math.log(srw / l), rx)
}
```

This is a very simple bootstrap filter, using Poisson resampling for simplicity and data locality, but does include use of the log-sum-exp trick to prevent over/underflow of raw weight calculations, and tracks the marginal (log-)likelihood of the observation. With this function we can now pass in a "prior" particle distribution in any collection conforming to our typeclass, together with a propagator function, an observation (log-)likelihood, and an observation, and it will return back a new collection of particles of exactly the same type that was provided for input. Note that all of the operations we require can be accomplished with the standard monadic collection operations declared in our typeclass.

Filtering as a functional fold

Once we have a function for executing one step of a particle filter, we can produce a function for particle filtering as a functional fold over a sequence of observations:

```
def pFilter[S: State, O: Observation,
  C[_]: GenericColl, D[O] <: GenTraversable[O]](
  x0: C[S], data: D[O],
  dataLik: (S, O) => LogLik, stepFun: S => S
  ): (LogLik, C[S]) = {
    val updater = update[S, O, C](dataLik, stepFun) _
    data.foldLeft((0.0, x0))((prev, o) => {
      val (oll, ox) = prev
      val (ll, x) = updater(ox, o)
      (oll + ll, x)
    })
}
```

Folding data structures is a fundamental concept in functional programming, and is exactly what is required for any kind of filtering problem

Marginal likelihoods and parameter estimation

So far we haven't said anything about parameters or parameter estimation, but this is appropriate, since parametrisation is a separate concern from filtering. However, once we have a function for particle filtering, we can produce a function concerned with evaluating marginal likelihoods trivially:

```
def pfMll[S: State, P: Parameter, O: Observation,
    C[_]: GenericColl, D[O] <: GenTraversable[O]](
  simX0: P => C[S], stepFun: P => S => S,
  dataLik: P => (S, O) => LogLik, data: D[O]
): (P => LogLik) = (th: P) =>
    pFilter(simX0(th), data, dataLik(th), stepFun(th))._1
```

Note that this higher-order function does not return a value, but instead a function which will accept a parameter as input and return a (log-)likelihood as output. This can then be used for parameter estimation purposes, perhaps being used in a PMMH pMCMC algorithm, or something else. Again, this is a separate concern.

Example

Here I'll just give a completely trivial toy example, purely to show how the functions work. Here we will just look at inferring the auto-regression parameter of a linear Gaussian AR(1)-plus-noise model using the functions we have developed. First we can simulate some synthetic data from this model, using a value of 0.8 for the auto-regression parameter:

```
val inNoise = Gaussian(0.0, 1.0).sample(99)
val state = DenseVector(inNoise.scanLeft(0.0)(
  (s, i) => 0.8 * s + i).toArray)
val noise = DenseVector(
  Gaussian(0.0, 2.0).sample(100).toArray)
val data = (state + noise).toArray.toList
```

Now assuming that we don't know the auto-regression parameter, we can construct a function to evaluate the likelihood of different parameter values as follows:

```
val mll = pfMll(
  (th: Double) => Gaussian(0.0, 10.0).
    sample(10000).toVector.par,
  (th: Double) => (s: Double) =>
    Gaussian(th * s, 1.0).draw,
  (th: Double) => (s: Double, o: Double) =>
    Gaussian(s, 2.0).logPdf(o),
  data
)
```

Note that the 4 characters **.par** at the end of the third line are the only difference between running this code serially or in parallel! Now we can run this code by calling the returned function with different values. So, hopefully **mll(0.8)** will return a larger log-likelihood than (say) **mll(0.6)** or **mll(0.9)**. The example code in the GitHub repo plots the results of calling **mll()** for a range of values. In this particular example, both the forward model and the likelihood are very cheap operations, so there is little to be gained from parallelisation. Nevertheless, I still get a speedup of more than a factor of two using the parallel version on my laptop.

> Note that if that was the genuine usecase, then it would be much better to parallelise the parameter range than the particle filter, due to providing better parallelisation granularity, but many other examples require parallelisation of the particle filter itself.

Conclusion

This short case study has illustrated how typeclasses can be used in Scala to write code that is parallelisation-agnostic. Code written in this way can be run on one or many cores as desired. We've illustrated the concept with a scalable particle filter, but nothing about the approach is specific to that application. It would be easy to build up a library of statistical routines this way, all of which can effectively exploit available parallel hardware. Further, although we haven't demonstrated it here, it is straightforward to extend this idea to allow code to be distributed over a cluster of parallel machines if necessary. For example, if an Apache Spark cluster is available, it is possible to define a Spark RDD instance for our generic collection typeclass, that will then allow us to run our (unmodified) particle filter code over a Spark cluster. This also highlights the fact that Spark can be useful for distributing computation as well as just processing "big data".

> I've started work on a library for parallel Monte Carlo simulation in Scala, based on these ideas: github.com/darrenjw/monte-scala – but at the time of writing it isn't yet ready for general use

Chapter 10

Appendix

10.1 Scala API for tensorflow

TensorFlow_scala is a scala API for a deep learning framework, TensorFlow.
An example for object detection is found here with sorce code.

Part II

Tour of Scala [2)]

Chapter 11

Scala Tour

11.1 Introduction

11.1.1 Welcome to the tour

This tour contains bite-sized introductions to the most frequently used features of Scala. It is intended for newcomers to the language.

This is just a brief tour, not a full language tutorial. If you want that, consider obtaining a book or consulting other resources [14)].

11.1.2 What is Scala?

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages.

11.1.3 Scala is object-oriented

Scala is a pure object-oriented language in the sense that *every value* is an object. Types and behavior of objects are described by **class**es and **trait**s. Classes are extended by subclassing and a flexible *mixin-based composition* mechanism as a clean replacement for multiple inheritance.

11.1.4 Scala is functional

Scala is also a functional language in the sense that *every function is a value*. Scala provides a *lightweight syntax* for defining anonymous functions, it supports *higher-order functions*, it allows functions to be nested, and supports *currying*. Scala's **case class**es and its built-in support for *pattern matching* model algebraic types used in many functional programming languages. *Singleton objects* provide a convenient way to group functions that aren't members of a class.

Furthermore, Scala's notion of pattern matching naturally extends to the *processing of XML data* with the help of *right-ignoring sequence patterns*, by way of general extension via *extractor objects*. In this context, *for comprehensions* are useful for formulating queries. These features make Scala ideal for developing applications like web services.

11.1.5 Scala is statically typed

Scala is equipped with an expressive type system that enforces statically that abstractions are used in a safe and coherent manner. In particular, the type system supports:

- *generic classes*
- *variance annotations*
- *upper* and *lower* type bounds,
- *inner classes* and *abstract types* as object members
- *compound types*
- *explicitly typed self references*
- *implicit parameters* and *conversions*
- *polymorphic methods*

Type inference means the user is not required to annotate code with redundant type information. In combination, these features provide a powerful basis for the safe reuse of programming abstractions and for the type-safe extension of software.

11.1.6 Scala is extensible

In practice, the development of domain-specific applications often requires domain-specific language extensions. Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in the form of libraries.

In many cases, this can be done without using meta-programming facilities such as macros. For example,

- *Implicit class*es allow adding extension methods to existing types.
- *String interpolation* is user-extensible with custom interpolators.

11.1.7 Scala interoperates

Scala is designed to interoperate well with the popular Java Runtime Environment (JRE). In particular, the interaction with the mainstream object-oriented Java programming language is as smooth as possible. Newer Java features like SAMs, *lambdas, annotations*, and *generics* have direct analogues in Scala.

Those Scala features without Java analogues, such as *default* and *dnamed parameters*, compile as close to Java as they can reasonably come. Scala has the same compilation model (separate compilation, dynamic class loading) like Java and allows access to thousands of existing high-quality libraries.

11.2 Basic

11.2.1 Trying Scala

Trying Scala in the Scala standard REPL

In the following, the Scala standard REPL is used for the interactive use of Scala.

11.2.2 Expressions

Expressions are computable statements.

```
1 + 1 //res0: Int = 2
```

You can output results of expressions using **println**.

```
println(1 + 1)                          //2
println("Hello," + " world!")           //Hello, world!
```

11.2.3 Values

You can name results of expressions with the **val** keyword.

```
val x = 1 + 1                           //x: Int = 2
println(x)                              //2
x = 3                                   //<console>:12: error: reassignment to val
                                        //              x = 3
                                        //                ^
```

Named results, such as **x** here, are called values. Referencing a value does not re-compute it. Values cannot be re-assigned.

Types of values can be inferred, but you can also explicitly state the type, like this:

```
val x: Int = 1 + 1                      //x: Int = 2
```

Notice how the type declaration Int comes after the identifier **x**. You also need a **:**.

11.2.4 Variables

Variables are like values, except you can re-assign them. You can define a variable with the **var** keyword.

```
var x = 1 + 1                           //x: Int = 2
x = 3                                   //x: Int = 3
```

As with values, you can explicitly state the type if you want:

```
var x: Int = 1 + 1                      //x: Int = 2
```

11.2.5 Blocks

You can combine expressions by surrounding them with . We call this a block. The result of the last expression in the block is the result of the overall block, too.

```
println({
  val x = 1 + 1
  x + 1
})                                      // 3
```

11.2.6 Functions

Functions are expressions that take parameters. You can define an anonymous function (i.e. no name) that returns a given integer plus one:

```
(x: Int) => x + 1                          //res0: Int => Int = <function1>
```

On the left of **=>** is a list of parameters. On the right is an expression involving the parameters. You can also name functions.

```
val addOne = (x: Int) => x + 1             //addOne: Int => Int = <function1>
println(addOne(1))                         //2
```

Functions may take multiple parameters.

```
val add = (x: Int, y: Int) => x + y        //add: Int => Int = <function2>
println(addOne(1,2))                       //3
```

Or it can take no parameters.

```
val getTheAnswer = () => 42                //getTheAnswer: () => Int = <function0>
println(getTheAnswer())                    // 42
```

11.2.7 Methods

Methods look and behave very similar to functions, but there are a few key differences between them. Methods are defined with the **def** keyword. **def** is followed by a name, parameter lists, a return type, and a body.

```
def add(x: Int, y: Int): Int = x + y       //add: (x: Int, y: Int)Int
println(add(1, 2))                         // 3
```

Notice how the return type is declared after the parameter list and a colon **: Int**. Methods can take multiple parameter lists.

```
def addThenMultiply(x: Int, y: Int)(multiplier: Int): Int = (x + y) * multiplier
                          //addThenMultiply: (x: Int, y: Int)(multiplier: Int)Int
println(addThenMultiply(1, 2)(3))          // 9
```

Or no parameter lists at all.

```
def name: String = System.getProperty("user.name")   //name: String
println("Hello, " + name + "!")            //Hello, miyazawa!
```

There are some other differences, but for now, you can think of them as something similar to functions. Methods can have multi-line expressions as well.

```
def getSquareString(input: Double): String = {
  val square = input * input
  square.toString
}                                          //getSquareString: (input: Double)String
```

The last expression in the body is the method's return value. (Scala does have a **return** keyword, but it's rarely used.)

11.2.8 Classes

You can define **class**es with the class keyword followed by its name and constructor parameters.

```
class Greeter(prefix: String, suffix: String) {
  def greet(name: String): Unit =
    println(prefix + name + suffix)
}                                          //defined class Greeter
```

The return type of the method **greet** is **Unit**, which says there's nothing meaningful to return. It's used similarly to **void** in Java and C. (A difference is that because every Scala expression must have some value, there is actually a singleton value of type Unit, written **()**. It carries no information.) You can make an instance of a class with the new keyword.

```
val greeter = new Greeter("Hello, ", "!")
greeter.greet("Scala developer")        //Hello, Scala developer!
```

We will cover classes in depth later.

11.2.9 Case Classes

Scala has a special type of class called a *case class*. By default, case classes are immutable and compared by value. You can define case classes with the **case class** keywords.

```
case class Point(x: Int, y: Int)        //defined class Point
```

You can instantiate case classes without **new** keyword.

```
val point = new Point(1, 2)             //point: Point = Point(1,2)
val anotherPoint = Point(1, 2)          //anotherPoint: Point = Point(1,2)
val yetAnotherPoint = Point(2, 2)       //yetAnotherPoint: Point = Point(2,2)
```

And they are compared by value.

```
if (point == anotherPoint) {
  println(point + " and " + anotherPoint + " are the same.")
} else {
  println(point + " and " + anotherPoint + " are different.")
}
                                        //Point(1,2) and Point(1,2) are the same.
if (point == yetAnotherPoint) {
  println(point + " and " + yetAnotherPoint + " are the same.")
} else {
  println(point + " and " + yetAnotherPoint + " are different.")
}
                                        //Point(1,2) and Point(2,2) are different.
```

There is a lot more to case classes that we'd like to introduce, and we are convinced you will fall in love with them! We will cover them in depth later.

11.2.10 Objects

Objects are single instances of their own definitions. You can think of them as *singletons* of their own classes. You can define objects with the **object** keyword.

```
object IdFactory {
  private var counter = 0
  def create(): Int = {
    counter += 1
    counter
  }
}                                          //defined object IdFactory
```

You can access an object by referring to its name.

```
val newId: Int = IdFactory.create()
println(newId)                             //1
val newerId: Int = IdFactory.create()
println(newerId)                           //2
```

We will cover objects in depth later.

11.2.11 Traits

Traits are types containing certain fields and methods. Multiple traits can be combined. You can define traits with **trait** keyword.

```
trait Greeter {
  def greet(name: String): Unit
}                                          //defined trait Greeter
```

Traits can also have default implementations.

```
trait Greeter {
  def greet(name: String): Unit =
    println("Hello, " + name + "!")
}                                          //defined trait Greeter
```

You can extend **trait**s with the **extends** keyword and override an implementation with the **override** keyword.

```
class DefaultGreeter extends Greeter    //defined class DefaultGreeter

class CustomizableGreeter(prefix: String, postfix: String) extends Greeter {
  override def greet(name: String): Unit = {
    println(prefix + name + postfix)
  }
}                                          //defined class CustomizableGreeter

val greeter = new DefaultGreeter()  //greeter: DefaultGreeter = DefaultGreeter@2f162
greeter.greet("Scala developer")           //Hello, Scala developer!
```

```
val customGreeter = new CustomizableGreeter("How are you, ", "?")
                //customGreeter: CustomizableGreeter = CustomizableGreeter@c9413d8
customGreeter.greet("Scala developer")  //How are you, Scala developer?
```

Here, **DefaultGreeter** extends only a single trait, but it could extend multiple traits. We will cover traits in depth later.

11.2.12 Main Method

The main method is an entry point of a program. The Java Virtual Machine requires a main method to be named **main** and take one argument, an array of strings. Using an object, you can define a main method as follows:

```
object Main {
  def main(args: Array[String]): Unit =
    println("Hello, Scala developer!")
}                                          //defined object Main
```

11.3 Unified Types

In Scala, all values have a type, including numerical values and functions. The diagram below illustrates a subset of the type hierarchy.

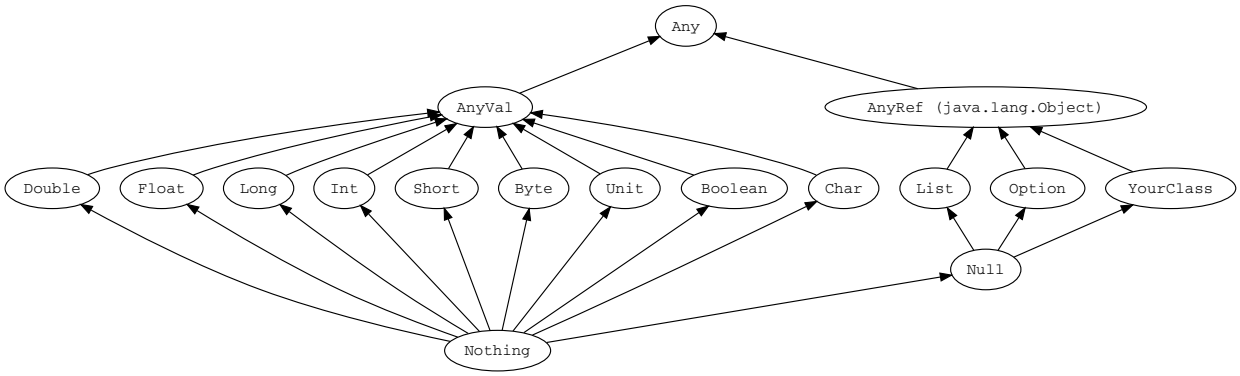


Figure 11.1. Unified types diagram

11.3.1 Scala Type Hierarchy

Any is the supertype of all types, also called the top type. It defines certain universal methods such as **equals**, **hashCode**, and **toString**. Any has two direct subclasses: **AnyVal** and **AnyRef**.

AnyVal represents value types. There are nine predefined value types and they are non-nullable: **Double, Float, Long, Int, Short, Byte, Char, Unit**, and **Boolean**. **Unit** is a value type which carries no meaningful information. There is exactly one instance of Unit which can be declared literally like so: **()**. All functions must return something so sometimes Unit is a useful return type.

AnyRef represents reference types. All non-value types are defined as reference types. Every user-defined type in Scala is a subtype of **AnyRef**. If Scala is used in the context of a Java runtime environment, **AnyRef** corresponds to java.lang.Object.

Here is an example that demonstrates that strings, integers, characters, boolean values, and functions are all objects just like every other object:

```
val list: List[Any] = List(
  "a string",
  732,              // an integer
  'c',              // a character
  true,             // a boolean value
  () => "an anonymous function returning a string"
)                  //list: List[Any] = List(a string, 732, c, true, <function0>)

list.foreach(element => println(element))
//a string
//732
//c
//true
//<function0>
```

It defines a variable list of type **List[Any]**. The list is initialized with elements of various types, but they all are instance of **scala.Any**, so you can add them to the list.

11.3.2 Type Casting

Value types can be cast in the following way:

Byte → **Short** → **Int** → **Long** → **Float** → **Double**
Char → **Int**

For example:

```
val x: Long = 987654321       //x: Long = 987654321
val y: Float = x              //y: Float = 9.8765434E8
                              //(note that some precision is lost in this case)
val face: Char = 9786
val number: Int = face        //number: Int = 9786
```

Casting is unidirectional. This will not compile:

```
val x: Long = 987654321       //x: Long = 987654321
val y: Float = x              //y: Float = 9.8765434E8
val z: Long = y               //<console>:13: error: type mismatch;
                              // found   : Float
                              // required: Long
                              //       val z: Long = y  // Does not conform
                              //                     ^
```

You can also cast a reference type to a subtype. This will be covered later in the tour.

11.3.3 Nothing and Null

Nothing is a subtype of all types, also called the bottom type. There is no value that has type **Nothing**. A common use is to signal non-termination such as a thrown exception, program exit, or an infinite loop (i.e., it is the type of an expression which does not evaluate to a value, or a method that does not return normally).

Null is a subtype of all reference types (i.e. any subtype of **AnyRef**). It has a single value identified by the keyword literal **null**. **Null** is provided mostly for interoperability with other JVM languages and should almost never be used in Scala code. We'll cover alternatives to **null** later in the tour.

11.4 Classes

Classes in Scala are blueprints for creating objects. They can contain methods, values, variables, types, objects, traits, and classes which are collectively called members. Types, objects, and traits will be covered later in the tour.

11.4.1 Defining a class

A minimal class definition is simply the keyword **class** and an identifier. Class names should be capitalized.

```
class User                                  //val user1 = new User
val user1 = new User                        //user1: User = User@6f75e721
```

The keyword **new** is used to create an instance of the class. **User** has a default constructor which takes no arguments because no constructor was defined. However, you'll often want a constructor and class body. Here is an example class definition for a point:

```
class Point(var x: Int, var y: Int) {

  def move(dx: Int, dy: Int): Unit = {
    x = x + dx
    y = y + dy
  }

  override def toString: String =
    s"($x, $y)"
}                                           //defined class Point

val point1 = new Point(2, 3)                //point1: Point = (2, 3)
point1.x                                    //res0: Int = 2
println(point1)                             //(2, 3)
//point1.toString is called.
```

This Point class has four members: the variables **x** and **y** and the methods **move** and **toString**. Unlike many other languages, the primary constructor is in the class signature **(var x: Int, var y: Int)**. The **move** method takes two integer arguments and returns the Unit value **()**, which carries no information. This corresponds roughly with **void** in Java/C-like languages. **toString**, on the other hand, does not take any arguments but returns a **String** value. Since **toString** overrides **toString** from AnyRef, it is tagged with the **override** keyword.

Primary constructor parameters with val and var are public. However, because **val**s are immutable, you can't write the following.

```
class Point(val x: Int, val y: Int)   //defined class Point
val point = new Point(1, 2)           //point: Point = Point@7e0ea639
point.x = 3                           //<console>:13: error: reassignment to val
                                      //       point.x = 3
                                      //                 ^
```

Parameters without **val** or **var** are private values, visible only within the class.

```
class Point(x: Int, y: Int)       //defined class Point
val point = new Point(1, 2)       //point: Point = Point@1efbd816
point.x                           //<console>:14: error: value x is not a member of
                                  //       point.x
                                  //              ^
```

11.4.2 Constructors

Constructors can have optional parameters by providing a default value like so:

```
class Point(var x: Int = 0, var y: Int = 0)  //defined class Point

val origin = new Point                       //origin: Point = Point@3527942a
println(origin.x, origin.y)                  //(0, 0)
val point1 = new Point(1)                    //point1: Point = Point@740cae06
println(point1.x, point1.y)                  //(1, 0)
```

In this version of the Point class, **x** and **y** have the default value **0** so no arguments are required.

Arguments with a default value must be located after those without a default value. Arguments without a default value must be specified in **new** constructs. If arguments with a default value are not specified, arguments located after that argument must be named in **new** constructs. Any argument can be named in **new** constructs.

```
class Point(val x: Int, val y: Int = 0, val z: Int = 0)  //defined class Point
val pointx = new Point(1)                    //pointx: Point = Point@10bd9df0
val pointxx = new Point(x=1)                 //pointxx: Point = Point@64524dd
val pointy = new Point(0, 1)                 //pointy: Point = Point@7b78ed6a
val pointz = new Point(0, z = 1)             //pointz: Point = Point@2f651f93
println(pointx.x, pointx.y, pointx.z)        //(1,0,0)
println(pointy.x, pointy.y, pointy.z)        //(0,1,0)
println(pointz.x, pointz.y, pointz.z)        //(0,0,1)
```

11.4.3 Private Members and Getter/Setter Syntax

Members are public by default. Use the **private** access modifier to hide them from outside of the class.

```
class Point {
  private var _x = 0
  private var _y = 0
  private val bound = 100

  def x = _x
  def x_= (newValue: Int): Unit = {
    if (newValue < bound) _x = newValue else printWarning
  }

  def y = _y
  def y_= (newValue: Int): Unit = {
    if (newValue < bound) _y = newValue else printWarning
  }

  private def printWarning = println("WARNING: Out of bounds")
}                                       //defined class Point

val point1 = new Point                  //point1: Point = Point@e73f9ac
point1.x = 99                           //point1.x: Int = 99
```

In this version of the **Point** class, the data is stored in private variables **_x** and **_y**. There are methods **def x** and **def y** for accessing the private data. **def x_=** and **def y_=** are for validating and setting the value of **_x** and **_y**. Notice the special syntax for the setters: the method has **_=** appended to the identifier of the getter and the parameters come after.

11.5 Traits

Traits are used to share interfaces and fields between classes. They are similar to Java 8's interfaces. Classes and objects can extend traits but traits cannot be instantiated and therefore have no parameters.

11.5.1 Defining a trait

A minimal trait is simply the keyword **trait** and an identifier:

```
trait HairColor                         //defined trait HairColor
```

Traits become especially useful as generic types and with abstract methods.

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}                                       //defined trait Iterator
```

Extending the **trait Iterator[A]** requires a type **A** and implementations of the methods **hasNext** and **next**.

11.5.2 Using traits

Use the **extends** keyword to extend a trait. Then implement any abstract members of the trait using the **override** keyword:

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}                                       //defined trait Iterator

class IntIterator(to: Int) extends Iterator[Int] {
  private var current = 0
  override def hasNext: Boolean = current < to
  override def next(): Int =  {
    if (hasNext) {
      val t = current
      current += 1
      t
    } else 0
  }
}                                       //defined class IntIterator

val iterator = new IntIterator(10)      //iterator: IntIterator = IntIterator@737996a0
iterator.next()                         //0
iterator.next()                         //1
```

This **IntIterator** class takes a parameter to as an upper bound. It **extends Iterator[Int]** which means that the **next** method must return an **Int**.

11.5.3 Subtyping

Where a given trait is required, a subtype of the trait can be used instead.

```
import scala.collection.mutable.ArrayBuffer   //import scala.collection.mutable.Arra

trait Pet {
  val name: String
}                                             //defined trait Pet

class Cat(val name: String) extends Pet       //defined class Cat
class Dog(val name: String) extends Pet       //defined class Dog

val dog = new Dog("Harry")                    //dog: Dog = Dog@5a42bbf4
val cat = new Cat("Sally")                    //cat: Cat = Cat@7cdbc5d3

val animals = ArrayBuffer.empty[Pet]          //animals: scala.collection.mutable.Ar
animals.append(dog)
animals.append(cat)
```

```
animals.foreach(pet => println(pet.name))      //Harry
                                               //Sally
```

The **trait Pet** has an abstract field **name** which gets implemented by Cat and Dog in their constructors. On the last line, we call **pet.name** which must be implemented in any subtype of the trait **Pet**.

11.6 Class Composition With Mixins

Mixins are traits which are used to compose a class.

```
abstract class A {
  val message: String
}                                              //defined class A
class B extends A {
  val message = "I'm an instance of class B"
}                                              //defined class B
trait C extends A {
  def loudMessage = message.toUpperCase()
}                                              //defined trait C
class D extends B with C                       //defined class D

val d = new D                                  //d: D = D@3e9b1010
println(d.message)                             //I'm an instance of class B
println(d.loudMessage)                         //I'M AN INSTANCE OF CLASS B
```

Class **D** has a superclass **B** and a mixin **C**. Classes can only have one superclass but many mixins (using the keywords **extends** and **with** respectively). The mixins and the superclass may have the same supertype.

Now let's look at a more interesting example starting with an abstract class:

```
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next(): T
}                                          //defined class AbsIterator
```

The class has an abstract type **T** and the standard iterator methods. Next, we'll implement a concrete class (all abstract members **T**, **hasNext**, and **next** have implementations):

```
class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length
  def next() = {
    val ch = s charAt i
    i += 1
    ch
  }
}                                          //defined class StringIterator
```

StringIterator takes a **String** and can be used to iterate over the String (e.g. to see if a String contains a certain character).

Now let's create a trait which also extends AbsIterator.

```
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit): Unit = while (hasNext) f(next())
}                                          //defined trait RichIterator
```

This trait implements **foreach** by continually calling the provided function **f: T => Unit** on the next element (**next()**) as long as there are further elements (while (**hasNext**)). Because RichIterator is a trait, it doesn't need to implement the abstract members of AbsIterator.

We would like to combine the functionality of StringIterator and RichIterator into a single class.

We would like to combine the functionality of **StringIterator** and **RichIterator** into a single class.

```
object StringIteratorTest extends App {
  class RichStringIter extends StringIterator("Scala") with RichIterator
  val richStringIter = new RichStringIter
  richStringIter foreach println
}                                    //defined object StringIteratorTest
```

The new class **RichStringIter** has **StringIterator** as a superclass and **RichIterator** as a mixin.

With single inheritance we would not be able to achieve this level of flexibility.

11.7 Higher Order Functions

Higher order functions take other functions as parameters or return a function as a result. This is possible because functions are first-class values in Scala. The terminology can get a bit confusing at this point, and we use the phrase "higher order function" for both methods and functions that take functions as parameters or that return a function.

One of the most common examples is the higher-order function **map** which is available for collections in Scala.

```
val salaries = Seq(20000, 70000, 40000)  //salaries: Seq[Int] = List(20000, 70000, 4
val doubleSalary = (x: Int) => x * 2     //doubleSalary: Int => Int = <function1>
val newSalaries = salaries.map(doubleSalary)  //newSalaries: Seq[Int] = List(40000,
```

doubleSalary is a function which takes a single Int, **x**, and returns **x * 2**. In general, the tuple on the left of the arrow **=>** is a parameter list and the value of the expression on the right is what gets returned. On line 3, the function **doubleSalary** gets applied to each element in the list of salaries.

To shrink the code, we could make the function anonymous and pass it directly as an argument to **map**:

```
val salaries = Seq(20000, 70000, 40000)    //salaries: Seq[Int] = List(20000, 70000,
val newSalaries = salaries.map(x => x * 2) //newSalaries: Seq[Int] = List(40000, 140
```

Notice how **x** is not declared as an Int in the above example. That's because the compiler can infer the type based on the type of function **map** expects. An even more idiomatic way to write the same piece of code would be:

```
val salaries = Seq(20000, 70000, 40000)    //salaries: Seq[Int] = List(20000, 70000,
val newSalaries = salaries.map(_ * 2) //newSalaries: Seq[Int] = List(40000, 140000,
```

Since the Scala compiler already knows the type of the parameters (a single Int), you just need to provide the right side of the function. The only caveat is that you need to use _ in place of a parameter name (it was **x** in the previous example).

11.7.1 Coercing methods into functions

It is also possible to pass methods as arguments to higher-order functions because the Scala compiler will coerce the method into a function.

```
case class WeeklyWeatherForecast(temperatures: Seq[Double]) {
  private def convertCtoF(temp: Double) = temp * 1.8 + 32
  def forecastInFahrenheit: Seq[Double] = temperatures.map(convertCtoF) // <-- passi
}                                       //defined class WeeklyWeatherForecast
val wwf = WeeklyWeatherForecast (List(-10.0, 0.0, 10.0))
wwf.forecastInFahrenheit                //res0: Seq[Double] = List(14.0, 32.0, 50.0)
```

Here the method **convertCtoF** is passed to **forecastInFahrenheit**. This is possible because the compiler coerces **convertCtoF** to the function **x => convertCtoF(x)** (note: **x** will be a generated name which is guaranteed to be unique within its scope).

11.7.2 Functions that accept functions

One reason to use higher-order functions is to reduce redundant code. Let's say you wanted some methods that could raise someone's salaries by various factors. To simplify, you can extract the repeated code into a higher-order function like so:

```
object SalaryRaiser {

  private def promotion(salaries: List[Double],
        promotionFunction: Double => Double): List[Double] =
    salaries.map(promotionFunction)

  def smallPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * 1.1)

  def bigPromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * math.log(salary))

  def hugePromotion(salaries: List[Double]): List[Double] =
    promotion(salaries, salary => salary * salary)
}
```

The new method, **promotion**, takes the salaries plus a function of type **Double => Double** (i.e. a function that takes a Double and returns a Double) and returns the product.

11.7.3 Functions that return functions

There are certain cases where you want to generate a function. Here's an example of a method that returns a function.

```
def urlBuilder(ssl: Boolean, domainName: String): (String, String) => String = {
  val schema = if (ssl) "https://" else "http://"
  (endpoint: String, query: String) => s"$schema$domainName/$endpoint?$query"
}     //urlBuilder: (ssl: Boolean, domainName: String)(String, String) => String

val domainName = "www.example.com"
def getURL = urlBuilder(ssl=true, domainName) //getURL: (String, String) => String
val endpoint = "users"
val query = "id=1"
val url = getURL(endpoint, query) //url: String = https://www.example.com/users?id=1
```

Notice the return type of **urlBuilder (String, String) => String**. This means that the returned anonymous function takes two Strings and returns a String. In this case, the returned anonymous function is **(endpoint: String, query: String) => s"https://www.example.com/*endpoint*?query"**.

11.8 Nested Methods

In Scala it is possible to nest method definitions. The following object provides a **factorial** method for computing the factorial of a given number:

```
def factorial(x: Int): Int = {
  def fact(x: Int, accumulator: Int): Int = {
    if (x <= 1) accumulator
    else fact(x - 1, x * accumulator)
  }
  fact(x, 1)
}

println("Factorial of 2: " + factorial(2))  //Factorial of 2: 2
println("Factorial of 3: " + factorial(3))  //Factorial of 3: 6
```

11.9 Multiple Parameter Lists (Currying)

Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments. This is formally known as *currying*.

Here is an example, defined in **Traversable** trait from Scala collections:

```
//def foldLeft[B](z: B)(op: (B, A) => B): B
```

foldLeft applies a binary operator **op** to an initial value **z** and all elements of this traversable, going left to right. Shown below is an example of its usage.

Starting with an initial value of 0, **foldLeft** here applies the function **(m, n) => m + n** to each element in the List and the previous accumulated value.

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val res = numbers.foldLeft(0)((m, n) => m + n)  //res: Int = 55
print(res)                                      //55
```

Multiple parameter lists have a more verbose invocation syntax; and hence should be used sparingly. Suggested use cases include:

11.9.1 Single functional parameter

In case of a single functional parameter, like **op** in the case of **foldLeft** above, multiple parameter lists allow a concise syntax to pass an anonymous function to the method. Without multiple parameter lists, the code would look like this:

```
numbers.foldLeft(0, {(m: Int, n: Int) => m + n})
```

Note that the use of multiple parameter lists here also allows us to take advantage of Scala type inference to make the code more concise as shown below; which would not be possible in a non-curried definition.

```
numbers.foldLeft(0)(_ + _)
```

Also, it allows us to fix the parameter **z** and pass around a partial function and reuse it as shown below:

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val numberFunc = numbers.foldLeft(List[Int]())_
        //numberFunc: ((List[Int], Int) => List[Int]) => List[Int] = <function1>
val squares = numberFunc((xs, x) => xs :+ x*x)
        //squares: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
print(squares.toString())    //List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

val cubes = numberFunc((xs, x) => xs :+ x*x*x)
        //cubes: List[Int] = List(1, 8, 27, 64, 125, 216, 343, 512, 729, 1000)
print(cubes.toString())    //List(1, 8, 27, 64, 125, 216, 343, 512, 729, 1000)
```

11.9.2 Implicit parameters

To specify certain parameters in a parameter list as **implicit**, multiple parameter lists should be used. An example of this is:

```
def execute(arg: Int)(implicit ec: ExecutionContext) = ???
```

11.10 Case Classes

Case classes are like regular classes with a few key differences which we will go over. Case classes are good for modeling immutable data. In the next step of the tour, we'll see how they are useful in *pattern matching*.

11.10.1 Defining a case class

A minimal case class requires the keywords **case class**, an identifier, and a parameter list (which may be empty):

```
case class Book(isbn: String)              //defined class Book
val frankenstein = Book("978-0486282114")  //frankenstein: Book = Book(978-048628211
val frankensteinNew = new Book("978-0486282114")  //frankensteinNew: Book = Book(978
```

Notice how the keyword TEXTBFnew was not used to instantiate the **Book** case class. This is because case classes have an **apply** method by default which takes care of object construction.

When you create a case class with parameters, the parameters are *public* **val**s.

```
case class Message(sender: String, recipient: String, body: String)
                                              //defined class Message
val message1 = Message("guillaume@quebec.ca", "jorge@catalonia.es", "Ça va ?")
  //message1: Message = Message(guillaume@quebec.ca,jorge@catalonia.es,Ça va ?)
println(message1.sender)                      //guillaume@quebec.ca
message1.sender = "travis@washington.us"  //<console>:14: error: reassignment to val
                                              //message1.sender = "travis@washington.us"
                                              //                  ^
```

You can't reassign **message1.sender** because it is a **val** (i.e. immutable). It is possible to use **var**s in case classes but this is discouraged.

```
case class MessageVal(val sender: String, val recipient: String, val body: String)
                                             //defined class MessageVal
case class MessageVar(var sender: String, var recipient: String, var body: String)
                                             //defined class MessageVar
```

11.10.2 Comparison

Case classes are compared by structure and not by reference:

```
case class Message(sender: String, recipient: String, body: String)

val message2 = Message("jorge@catalonia.es", "guillaume@quebec.ca", "Com va?")
val message3 = Message("jorge@catalonia.es", "guillaume@quebec.ca", "Com va?")
val messagesAreTheSame = message2 == message3    //messagesAreTheSame: Boolean = tru
```

Even though **message2** and **message3** refer to different objects, the value of each object is equal.

11.10.3 Copying

You can create a (shallow) copy of an instance of a case class simply by using the **copy** method. You can optionally change the constructor arguments.

```
case class Message(sender: String, recipient: String, body: String)
val message4 = Message("julien@bretagne.fr", "travis@washington.us", "Me zo o komz g

val message5 = message4.copy(sender = message4.recipient, recipient = "claire@bourgo
message5.sender                 //res0: String = travis@washington.us
message5.recipient              //res1: String = claire@bourgogne.fr
message5.body                   //res2: String = Me zo o komz gant ma amezeg
```

The recipient of **message4** is used as the sender of **message5** but the body of **message4** was (shallow-)copied directly.

11.11 Pattern Matching

Pattern matching is a mechanism for checking a value against a pattern. A successful match can also deconstruct a value into its constituent parts. It is a more powerful version of the **switch** statement in Java and it can likewise be used in place of a series of if/else statements.

11.11.1 Syntax

A match expression has a value, the **match** keyword, and at least one **case** clause.

```
import scala.util.Random

val x: Int = Random.nextInt(10) //x: Int = 5

x match {
  case 0 => "zero"
  case 1 => "one"
  case 2 => "two"
  case _ => "many"
}                                 //r0: String = many
```

The **val x** above is a random integer between 0 and 10. **x** becomes the left operand of the **match** operator and on the right is an expression with four cases. The last case _ is a "catch all" case for any number greater than 2. Cases are also called alternatives.

Match expressions have a value.

```
def matchTest(x: Int): String = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "many"
}
matchTest(3)                              //many
matchTest(1)                              //one
```

This **match** expression has a type String because all of the cases return String. Therefore, the function **matchTest** returns a String.

11.11.2 Matching on case classes

Case classes are especially useful for pattern matching.

```
abstract class Notification

case class Email(sender: String, title: String, body: String) extends
                                                                Notification

case class SMS(caller: String, message: String) extends Notification

case class VoiceRecording(contactName: String, link: String) extends
                                                                Notification
```

Notification is an abstract super class which has three concrete **Notification** types implemented with case classes **Email**, **SMS**, and **VoiceRecording**. Now we can do pattern matching on these case classes:

```
def showNotification(notification: Notification): String = {
  notification match {
    case Email(email, title, _) =>
      s"You got an email from $email with title: $title"
    case SMS(number, message) =>
      s"You got an SMS from $number! Message: $message"
    case VoiceRecording(name, link) =>
      s"you received a Voice Recording from $name! Click the link to hear it: $link"
  }
}
val someSms = SMS("12345", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

println(showNotification(someSms))
//You got an SMS from 12345! Message: Are you there?

println(showNotification(someVoiceRecording))
// you received a Voice Recording from Tom! Click the link to hear it: voicerecordin
```

The function **showNotification** takes as a parameter the abstract type **Notification** and matches on the type of **Notification** (i.e. it figures out whether it's an **Email**, **SMS**, or **VoiceRecording**). In the case **Email(email, title, _)** the fields **email** and **title** are used in the return value but the **body** field is ignored with _.

11.11.3 Pattern guards

attern guards are simply boolean expressions which are used to make cases more specific. Just add **if <boolean expression>** after the pattern.

```
def showImportantNotification(notification: Notification, importantPeopleInfo: Seq[S
  notification match {
    case Email(email, _, _) if importantPeopleInfo.contains(email) =>
      "You got an email from special someone!"
    case SMS(number, _) if importantPeopleInfo.contains(number) =>
      "You got an SMS from special someone!"
    case other =>
      showNotification(other) // nothing special, delegate to our original showNotif
  }
}

val importantPeopleInfo = Seq("867-5309", "jenny@gmail.com")

val someSms = SMS("867-5309", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")
val importantEmail = Email("jenny@gmail.com", "Drinks tonight?", "I'm free after 5!"
```

```
val importantSms = SMS("867-5309", "I'm here! Where are you?")

println(showImportantNotification(someSms, importantPeopleInfo))
//You got an SMS from special someone!
println(showImportantNotification(someVoiceRecording, importantPeopleInfo))
//you received a Voice Recording from Tom! Click the link to hear it: voicerecording
println(showImportantNotification(importantEmail, importantPeopleInfo))
//You got an email from special someone!
println(showImportantNotification(importantSms, importantPeopleInfo))
//You got an SMS from special someone!
```

In the case **Email(email, _, _)** if **importantPeopleInfo.contains(email)**, the pattern is matched only if the **email** is in the list of important people.

11.11.4 Matching on type only

You can match on the type like so:

```
abstract class Device
case class Phone(model: String) extends Device{
  def screenOff = "Turning screen off"
}
case class Computer(model: String) extends Device {
  def screenSaverOn = "Turning screen saver on..."
}

def goIdle(device: Device) = device match {
  case p: Phone => p.screenOff
  case c: Computer => c.screenSaverOn
}
```

def goIdle has a different behavior depending on the type of **Device**. This is useful when the case needs to call a method on the pattern. It is a convention to use the first letter of the type as the case identifier (**p** and **c** in this case).

11.11.5 Sealed classes

Traits and classes can be marked **sealed** which means all subtypes must be declared in the same file. This assures that all subtypes are known.

```
sealed abstract class Furniture
case class Couch() extends Furniture
case class Chair() extends Furniture

def findPlaceToSit(piece: Furniture): String = piece match {
  case a: Couch => "Lie on the couch"
  case b: Chair => "Sit on the chair"
}                                  //findPlaceToSit: (piece: Furniture)String
```

This is useful for pattern matching because we don't need a "catch all" case.

11.11.6 Notes

Scala's pattern matching statement is most useful for matching on algebraic types expressed via **case class**es. Scala also allows the definition of patterns independently of **case class**es, using **unapply** methods in **extractor object**s.

11.12 Singleton Objects

An object is a class that has exactly one instance. It is created lazily when it is referenced, like a lazy val.

As a top-level value, an object is a singleton.

As a member of an enclosing class or as a local value, it behaves exactly like a lazy val.

11.12.1 Defining a singleton object

An object is a value. The definition of an object looks like a class, but uses the keyword **object**:

```
object Box
```

Here's an example of an object with a method:

```
package logging

object Logger {
  def info(message: String): Unit = println(s"INFO: $message")
}
```

The method **info** can be imported from anywhere in the program. Creating utility methods like this is a common use case for singleton objects.

Let's see how to use **info** in another package:

```
import logging.Logger.info

class Project(name: String, daysToComplete: Int)

class Test {
  val project1 = new Project("TPS Reports", 1)
  val project2 = new Project("Website redesign", 5)
  info("Created projects")                    //INFO: Created projects
}
```

The **info** method is visible because of the import statement, **import logging.Logger.info**.

Imports require a "stable path" to the imported symbol, and an object is a stable path.

Note: If an object is not top-level but is nested in another class or object, then the object is "path-dependent" like any other member. This means that given two kinds of beverages, **class Milk** and **class OrangeJuice**, a class member **object NutritionInfo** "depends" on the enclosing instance, either milk or orange juice. **milk.NutritionInfo** is entirely distinct from **oj.NutritionInfo**.

11.12.2 Companion objects

An object with the same name as a class is called a companion object. Conversely, the class is the object's companion class. A companion class or object can access the private members of its companion. Use a companion object for methods and values which are not specific to instances of the companion class.

```
import scala.math._

case class Circle(radius: Double) {
  import Circle._
  def area: Double = calculateArea(radius)
}

object Circle {
  private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)
}

val circle1 = new Circle(5.0)

circle1.area                                  //res7: Double = 78.53981633974483
```

The **class Circle** has a member **area** which is specific to each instance, and the singleton object Circle has a method **calculateArea** which is available to every instance.

The companion object can also contain factory methods:

```
class Email(val username: String, val domainName: String)

object Email {
  def fromString(emailString: String): Option[Email] = {
    emailString.split('@') match {
      case Array(a, b) => Some(new Email(a, b))
      case _ => None
    }
  }
}

val scalaCenterEmail = Email.fromString("scala.center@epfl.ch")
scalaCenterEmail match {
  case Some(email) => println(
    s"""Registered an email
       |Username: ${email.username}
       |Domain name: ${email.domainName}
     """)
  case None => println("Error: could not parse email")
}
//Registered an email
//Username: scala.center
//Domain name: epfl.ch
```

The **object Email** contains a factory **fromString** which creates an Email instance from a String. We return it as an **Option[Email]** in case of parsing errors.

Note: If a class or object has a companion, both must be defined in the same file. To define companions in the REPL, either define them on the same line or enter **:paste** mode.

11.12.3 Notes for Java programmers

static members in Java are modeled as ordinary members of a companion object in Scala.

When using a companion object from Java code, the members will be defined in a companion class with a **static** modifier. This is called *static forwarding*. It occurs even if you haven't defined a companion class yourself.

11.13 Regular Expression Patterns

Regular expressions are strings which can be used to find patterns (or lack thereof) in data. Any string can be converted to a regular expression using the **.r** method.

```
import scala.util.matching.Regex

val numberPattern: Regex = "[0-9]".r

numberPattern.findFirstMatchIn("awesomepassword") match {
  case Some(_) => println("Password OK")
  case None => println("Password must contain a number")
}
```

In the above example, the **numberPattern** is a **Regex** (regular expression) which we use to make sure a password contains a number.

You can also search for groups of regular expressions using parentheses.

```
import scala.util.matching.Regex

val keyValPattern: Regex = "([0-9a-zA-Z-#() ]+): ([0-9a-zA-Z-#() ]+)".r

val input: String =
  """background-color: #A03300;
    |background-image: url(img/header100.png);
    |background-position: top center;
    |background-repeat: repeat-x;
    |background-size: 2160px 108px;
    |margin: 0;
    |height: 108px;
    |width: 100%;""".stripMargin
                                                    //input: String =
                                                    //background-color: #A03300;
                                                    //background-image: url(img/header10
                                                    //background-position: top center;
```

```
                                                    //background-repeat: repeat-x;
                                                    //background-size: 2160px 108px;
                                                    //margin: 0;
                                                    //height: 108px;
                                                    //width: 100%;

for (patternMatch <- keyValPattern.findAllMatchIn(input))
  println(s"key: ${patternMatch.group(1)} value: ${patternMatch.group(2)}")
```

Here we parse out the keys and values of a String. Each match has a group of sub-matches. Here is the output:

```
key: background-color value: #A03300
key: background-image value: url(img
key: background-position value: top center
key: background-repeat value: repeat-x
key: background-size value: 2160px 108px
key: margin value: 0
key: height value: 108px
key: width value: 100
```

- In single quoted strings, special characters such as a line feed cannot be included. Instead, **n** is interpreted as a line feed, unless the escape character is not prepended, but variables are not replaced by their values.
- Triple quoted strings can include special characters such as line feeds, the escape character, single quotes and dollars.
- To use basic string interpolation in Scala, precede your string with the letter **s** and include your variables inside the string, with each variable name preceded by a **$** character. Any arbitrary expression can be embedded in **$ {}**.
- **raw** interpolators can also include the escape characters except quotes, but still performs variable substitutions.

```
val helloVar = "Hello"                  //helloVar: String = Hello
"$helloVar !\n"                         //res0: String =
                                        //res1: String =
                                        //"$helloVar !
                                        //"
"$helloVar !\\n"                        //res2: String = $helloVar !\n

"""$helloVar, "World"!\n"""             //res3: String = $helloVar, "World"!\n

s"$helloVar !\n"                        //res4: String =
                                        //"Hello !
                                        //"
s"$helloVar !\\n"                       //res5: String = Hello !\n
```

```
s"""$helloVar, "World"!\n"""          //res6: String =
                                      //"Hello, "World"!
                                      //"

raw"$helloVar !\n"                    //res7: String = Hello !\n
raw"""$helloVar, "World"!\n"""        //res8: String = Hello, "World"!\n
```

11.14 Extractor Objects

An extractor object is an object with an **unapply** method. Whereas the **apply** method is like a constructor which takes arguments and creates an object, the **unapply** takes an object and tries to give back the arguments. This is most often used in pattern matching and partial functions.

```
import scala.util.Random

object CustomerID {

  def apply(name: String) = s"$name--${Random.nextLong}"

  def unapply(customerID: String): Option[String] = {
    val name = customerID.split("--").head
    if (name.nonEmpty) Some(name) else None
  }
}

val customer1ID = CustomerID("Sukyoung")  // Sukyoung--23098234908
customer1ID match {
  case CustomerID(name) => println(name)  // prints Sukyoung
  case _ => println("Could not extract a CustomerID")
}
```

The **apply** method creates a **CustomerID** string from a name. The **unapply** does the inverse to get the name back. When we call **CustomerID("Sukyoung")**, this is shorthand syntax for calling **CustomerID.apply("Sukyoung** When we call **case CustomerID(name) => println(name)**, we are calling the **unapply** method.

The **unapply** method can also be used to assign a value.

```
val customer2ID = CustomerID("Nico")
val CustomerID(name) = customer2ID
println(name)                                  //Nico
```

This is equivalent to **val name = CustomerID.unapply(customer2ID).get**. If there is no match, a scala.MatchError is thrown:

```
val CustomerID(name2) = "--asdfasdfasdf"      //
```

The return type of an **unapply** should be chosen as follows:

- If it is just a test, return a **Boolean**. For instance **case even()**.

- If it returns a single sub-value of type T, return an **Option[T]**.
- If you want to return several sub-values T1,...,Tn, group them in an optional tuple **Option[(T1,...,Tn)]**.

Sometimes, the number of sub-values isn't fixed and we would like to return a sequence. For this reason, you can also define patterns through **unapplySeq** which returns **Option[Seq[T]]**. This mechanism is used for instance in pattern **case List(x1, ..., xn)**.

11.15 For Comprehensions

Scala offers a lightweight notation for expressing *sequence comprehensions*. Comprehensions have the form **for (enumerators) yield e**, where **enumerators** refers to a semicolon-separated list of enumerators. An enumerator is either a generator which introduces new variables, or it is a filter. A comprehension evaluates the body **e** for each binding generated by the enumerators and returns a sequence of these values.

Here's an example:

```
case class User(val name: String, val age: Int)

val userBase = List(new User("Travis", 28),
  new User("Kelly", 33),
  new User("Jennifer", 44),
  new User("Dennis", 23))

val twentySomethings = for (user <- userBase if (user.age >=20 && user.age < 30))
  yield user.name                                  //twentySomethings: List[String] =

twentySomethings.foreach(name => println(name))   //Travis
                                                  //Dennis
```

The **for** loop used with a **yield** statement actually creates a List. Because we said **yield user.name**, it's a List[String]. **user <- userBase** is our generator and **if (user.age >=20 && user.age < 30)** is a guard that filters out users who are in their 20s.

Here is a more complicated example using two generators. It computes all pairs of numbers between **0** and **n-1** whose sum is equal to a given value **v**:

```
def foo(n: Int, v: Int) =
   for (i <- 0 until n;
        j <- i until n if i + j == v)
   yield (i, j)
//foo: (n: Int, v: Int)scala.collection.immutable.IndexedSeq[(Int, Int)]

foo(10, 10) foreach {
  case (i, j) =>
    print(s"($i, $j) ")                    //(1, 9) (2, 8) (3, 7) (4, 6) (5, 5)
}
```

Here **n == 10** and **v == 10**. On the first iteration, **i == 0** and **j == 0** so **i + j != v** and therefore nothing is yielded. **j** gets incremented 9 more times before **i** gets incremented to **1**. Without the **if** guard, this would simply print the following:

```
(0, 0) (0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (0, 6) (0, 7) (0, 8) (0, 9) (1, 1) ...
```

Note that comprehensions are not restricted to lists. Every datatype that supports the operations **withFilter**, **map**, and **flatMap** (with the proper types) can be used in sequence comprehensions.

You can omit **yield** in a comprehension. In that case, comprehension will return Unit. This can be useful in case you need to perform side-effects. Here's a program equivalent to the previous one, but without using **yield**:

```
def foo(n: Int, v: Int) =
   for (i <- 0 until n;
        j <- i until n if i + j == v)
   print(s"($i, $j)")

foo(10, 10)                                  //(1, 9)(2, 8)(3, 7)(4, 6)(5, 5)
```

11.16 Generic Classes

Generic classes are classes which take a type as a parameter. They are particularly useful for collection classes.

11.16.1 Defining a generic class

Generic classes take a type as a parameter within square brackets **[]**. One convention is to use the letter **A** as type parameter identifier, though any parameter name may be used.

```
class Stack[A] {
  private var elements: List[A] = Nil
  def push(x: A) { elements = x :: elements }
  def peek: A = elements.head
  def pop(): A = {
    val currentTop = peek
    elements = elements.tail
    currentTop
  }
}
```

This implementation of a **Stack** class takes any type **A** as a parameter. This means the underlying list, **var elements: List[A] = Nil**, can only store elements of type **A**. The procedure **def push** only accepts objects of type **A** (note: **elements = x :: elements** reassigns elements to a new list created by prepending **x** to the current **elements**).

11.16.2 Usage

To use a generic class, put the type in the square brackets in place of **A**.

```
val stack = new Stack[Int]
stack.push(1)
stack.push(2)
```

```
println(stack.pop)                  //2
println(stack.pop)                  //1
```

The instance **stack** can only take **Int**s. However, if the type argument had subtypes, those could be passed in:

```
class Fruit
class Apple extends Fruit
class Banana extends Fruit

val stack = new Stack[Fruit]
val apple = new Apple
val banana = new Banana

stack.push(apple)
stack.push(banana)
```

Class **Apple** and **Banana** both extend **Fruit** so we can push instances **apple** and **banana** onto the stack of Fruit.

*Note: subtyping of generic types is *invariant*. This means that if we have a stack of characters of type **Stack[Char]** then it cannot be used as an integer stack of type **Stack[Int]**. This would be unsound because it would enable us to enter true integers into the character stack. To conclude, **Stack[A]** is only a subtype of **Stack[B]** if and only if **B = A**. Since this can be quite restrictive, Scala offers a type parameter annotation mechanism to control the subtyping behavior of generic types.*

11.17 Variences

Variance is the correlation of subtyping relationships of complex types and the subtyping relationships of their component types. Scala supports variance annotations of type parameters of **generic classes**, to allow them to be covariant, contravariant, or invariant if no annotations are used. The use of variance in the type system allows us to make intuitive connections between complex types, whereas the lack of variance can restrict the reuse of a class abstraction.

```
class Foo[+A]                   // A covariant class
class Bar[-A]                   // A contravariant class
class Baz[A]                    // An invariant class
```

11.17.1 Covariance

A type parameter **A** of a generic class can be made covariant by using the annotation **+A**. For some **class List[+A]**, making **A** covariant implies that for two types **A** and **B** where **A** is a subtype of **B**, then **List[A]** is a subtype of **List[B]**. This allows us to make very useful and intuitive subtyping relationships using generics.

Consider this simple class structure:

```
abstract class Animal {
  def name: String
}
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal
```

Both **Cat** and **Dog** are subtypes of **Animal**. The Scala standard library has a generic immutable sealed abstract class **List[+A]** class, where the type parameter **A** is covariant. This means that a **List[Cat]** is a **List[Animal]** and a **List[Dog]** is also a List[Animal]. Intuitively, it makes sense that a list of cats and a list of dogs are each lists of animals, and you should be able to substitute either of them for a **List[Animal]**.

In the following example, the method **printAnimalNames** will accept a list of animals as an argument and print their names each on a new line. If **List[A]** were not covariant, the last two method calls would not compile, which would severely limit the usefulness of the **printAnimalNames** method.

```
object CovarianceTest extends App {
  def printAnimalNames(animals: List[Animal]): Unit = {
    animals.foreach { animal =>
      println(animal.name)
    }
  }

  val cats: List[Cat] = List(Cat("Whiskers"), Cat("Tom"))
  val dogs: List[Dog] = List(Dog("Fido"), Dog("Rex"))

  printAnimalNames(cats)
  //Whiskers
  //Tom

  printAnimalNames(dogs)
  //Fido
  //Rex
}
```

The **App** trait can be used to quickly turn objects into executable programs. Here, object **CovarianceTest** inherits the **main** method of App. **args** returns the current command line arguments as an array.

11.17.2 Contravariance

A type parameter **A** of a generic class can be made contravariant by using the annotation **-A**. This creates a subtyping relationship between the class and its type parameter that is similar, but opposite to what we get with covariance. That is, for some class **Writer[-A]**, making **A** contravariant implies that for two types **A** and **B** where **A** is a subtype of **B**, **Writer[B]** is a subtype of **Writer[A]**.

Consider the **Cat**, **Dog**, and **Animal** classes defined above for the following example:

```
abstract class Printer[-A] {
  def print(value: A): Unit
}
```

A **Printer[-A]** is a simple class that knows how to print out some type **A**. Let's define some subclasses for specific types:

```
class AnimalPrinter extends Printer[Animal] {
  def print(animal: Animal): Unit =
    println("The animal's name is: " + animal.name)
```

```
}

class CatPrinter extends Printer[Cat] {
  def print(cat: Cat): Unit =
    println("The cat's name is: " + cat.name)
}
```

If a **Printer[Cat]** knows how to print any **Cat** to the console, and a **Printer[Animal]** knows how to print any **Animal** to the console, it makes sense that a **Printer[Animal]** would also know how to print any **Cat**. The inverse relationship does not apply, because a **Printer[Cat]** does not know how to print any **Animal** to the console. Therefore, we should be able to substitute a **Printer[Animal]** for a **Printer[Cat]**, if we wish, and making **Printer[A]** contravariant allows us to do exactly that.

```
object ContravarianceTest extends App {
  val myCat: Cat = Cat("Boots")

  def printMyCat(printer: Printer[Cat]): Unit = {
    printer.print(myCat)
  }

  val catPrinter: Printer[Cat] = new CatPrinter
  val animalPrinter: Printer[Animal] = new AnimalPrinter

  printMyCat(catPrinter)
  printMyCat(animalPrinter)
}
```

The output of this program will be:

```
The cat's name is: Boots
The animal's name is: Boots
```

11.17.3 Intravariance

Generic classes in Scala are invariant by default. This means that they are neither covariant nor contravariant. In the context of the following example, **Container** class is invariant. A **Container[Cat]** is not a **Container[Animal]**, nor is the reverse true.

```
class Container[A](value: A) {
  private var _value: A = value
  def getValue: A = _value
  def setValue(value: A): Unit = {
    _value = value
  }
}
```

It may seem like a **Container[Cat]** should naturally also be a **Container[Animal]**, but allowing a mutable generic class to be covariant would not be safe. In this example, it is very important that **Container** is invariant. Supposing **Container** was actually covariant, something like this could happen:

```
val catContainer: Container[Cat] = new Container(Cat("Felix"))
val animalContainer: Container[Animal] = catContainer
animalContainer.setValue(Dog("Spot"))
val cat: Cat = catContainer.getValue   // Oops, we'd end up with a Dog assigned to a
```

Fortunately, the compiler stops us long before we could get this far.

11.17.4 Other Examples

Another example that can help one understand variance is **trait Function1[-T, +R]** from the Scala standard library. **Function1** represents a function with one argument, where the first type parameter **T** represents the argument type, and the second type parameter **R** represents the return type. A **Function1** is contravariant over its argument type, and covariant over its return type. For this example we'll use the literal notation **A => B** to represent a **Function1[A, B]**.

Assume the similar **Cat**, **Dog**, **Animal** inheritance tree used earlier, plus the following:

```
abstract class SmallAnimal extends Animal
case class Mouse(name: String) extends SmallAnimal
```

Suppose we're working with functions that accept types of animals, and return the types of food they eat. If we would like a **Cat => SmallAnimal** (because cats eat small animals), but are given a **Animal => Mouse** instead, our program will still work. Intuitively an **Animal => Mouse** will still accept a **Cat** as an argument, because a **Cat** is an **Animal**, and it returns a **Mouse**, which is also a **SmallAnimal**. Since we can safely and invisibly substitute the former for the latter, we can say **Animal => Mouse** is a subtype of **Cat => SmallAnimal**.

11.17.5 Comparison With Other Languages

Variance is supported in different ways by some languages that are similar to Scala. For example, variance annotations in Scala closely resemble those in C#, where the annotations are added when a class abstraction is defined (*declaration-site variance*). In Java, however, variance annotations are given by clients when a class abstraction is used (*use-site variance*).

11.18 Upper Type Bounds

In Scala, type parameters and abstract types may be constrained by a type bound. Such type bounds limit the concrete values of the type variables and possibly reveal more information about the members of such types. An *upper type bound* **T <: A** declares that type variable **T** refers to a subtype of type **A**. Here is an example that demonstrates upper type bound for a type parameter of class **PetContainer**:

```
abstract class Animal {
 def name: String
}

abstract class Pet extends Animal {}

class Cat extends Pet {
  override def name: String = "Cat"
```

```
}

class Dog extends Pet {
  override def name: String = "Dog"
}

class Lion extends Animal {
  override def name: String = "Lion"
}

class PetContainer[P <: Pet](p: P) {
  def pet: P = p
}

val dogContainer = new PetContainer[Dog](new Dog)
val catContainer = new PetContainer[Cat](new Cat)
val lionContainer = new PetContainer[Lion](new Lion)
//<console>:15: error: type arguments [Lion] do not conform to
//       class PetContainer's type parameter bounds [P <: Pet]
//       val lionContainer = new PetContainer[Lion](new Lion)
//                                  ^
```

The **class PetContainer** takes a type parameter **P** which must be a subtype of **Pet**. **Dog** and **Cat** are subtypes of **Pet** so we can create a new **PetContainer[Dog]** and **PetContainer[Cat]**. However, if we tried to create a **PetContainer[Lion]**, we would get the following Error:

> type arguments [Lion] do not conform to class PetContainer's type parameter bounds [P <: Pet]

This is because Lion is not a subtype of **Pet**.

11.19 Lower Type Bounds

While upper type bounds limit a type to a subtype of another type, *lower type bounds* declare a type to be a supertype of another type. The term **B >: A** expresses that the type parameter **B** or the abstract type **B** refer to a supertype of type **A**. In most cases, **A** will be the type parameter of the class and **B** will be the type parameter of a method.

Here is an example where this is useful:

```
trait Node[+B] {
  def prepend(elem: B): Node[B]
}
//<console>:12: error: covariant type B occurs in contravariant position in type B o
//         def prepend(elem: B): Node[B]


case class ListNode[+B](h: B, t: Node[B]) extends Node[B] {
  def prepend(elem: B): ListNode[B] = ListNode(elem, this)
```

```
  def head: B = h
  def tail: Node[B] = t
}

case class Nil[+B]() extends Node[B] {
  def prepend(elem: B): ListNode[B] = ListNode(elem, this)
}
```

This program implements a singly-linked list. **Nil** represents an empty element (i.e. an empty list). **class ListNode** is a node which contains an element of type **B** (**head**) and a reference to the rest of the list (**tail**). The **class Node** and its subtypes are covariant because we have **+B**.

However, this program does not compile because the parameter **elem** in prepend is of type **B**, which we declared covariant. This doesn't work because functions are contravariant in their parameter types and covariant in their result types.

To fix this, we need to flip the variance of the type of the parameter **elem** in prepend. We do this by introducing a new type parameter **U** that has **B** as a lower type bound.

```
trait Node[+B] {
  def prepend[U >: B](elem: U): Node[U]
}

case class ListNode[+B](h: B, t: Node[B]) extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
  def head: B = h
  def tail: Node[B] = t
}

case class Nil[+B]() extends Node[B] {
  def prepend[U >: B](elem: U): ListNode[U] = ListNode(elem, this)
}
```

Now we can do the following:

```
trait Bird
case class AfricanSwallow() extends Bird
case class EuropeanSwallow() extends Bird


val africanSwallowList= ListNode[AfricanSwallow](AfricanSwallow(), Nil())
//africanSwallowList: ListNode[AfricanSwallow] = ListNode(AfricanSwallow(),Nil())
val birdList: Node[Bird] = africanSwallowList
                        //birdList: Node[Bird] = ListNode(AfricanSwallow(),Nil())
val birdList2 = birdList.prepend(new EuropeanSwallow)
//birdList2: Node[Bird] = ListNode(EuropeanSwallow(),ListNode(AfricanSwallow(),Nil()
```

The **Node[Bird]** can be assigned the **africanSwallowList** but then accept **EuropeanSwallow**s.

Exercise Write a method that has an argument, **list: Node[B]**, and print each element from a head to a tail-end.

11.20 Inner Classes

In Scala it is possible to let classes have other classes as members. As opposed to Java-like languages where such inner classes are members of the enclosing class, in Scala such inner classes are bound to the outer object. Suppose we want the compiler to prevent us, at compile time, from mixing up which nodes belong to what graph. Path-dependent types provide a solution.

To illustrate the difference, we quickly sketch the implementation of a graph datatype:

```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node) {
      if (connectedNodes.find(node.equals).isEmpty) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode: Node = {
    val res = new Node
    nodes = res :: nodes
    res
  }
}
```

This program represents a graph as a list of nodes (**List[Node]**). Each node has a list of other nodes it's connected to (**connectedNodes**). The **class Node** is a *path-dependent type* because it is nested in the **class Graph**. Therefore, all nodes in the **connectedNodes** must be created using the **newNode** from the same instance of **Graph**.

```
val graph1: Graph = new Graph
val node1: graph1.Node = graph1.newNode
val node2: graph1.Node = graph1.newNode
val node3: graph1.Node = graph1.newNode
node1.connectTo(node2)
node3.connectTo(node1)
```

We have explicitly declared the type of **node1**, **node2**, and **node3** as **graph1.Node** for clarity but the compiler could have inferred it. This is because when we call **graph1.newNode** which calls **new Node**, the method is using the instance of **Node** specific to the instance **graph1**.

If we now have two graphs, the type system of Scala does not allow us to mix nodes defined within one graph with the nodes of another graph, since the nodes of the other graph have a different type. Here is an illegal program:

```
val graph1: Graph = new Graph
val node1: graph1.Node = graph1.newNode
val node2: graph1.Node = graph1.newNode
node1.connectTo(node2)      // legal
val graph2: Graph = new Graph
```

```
val node3: graph2.Node = graph2.newNode
node1.connectTo(node3)
//<console>:17: error: type mismatch;
// found   : graph2.Node
// required: graph1.Node
//       node1.connectTo(node3)
//                       ^
```

The type **graph1.Node** is distinct from the type **graph2.Node**. In Java, the last line in the previous example program would have been correct. For nodes of both graphs, Java would assign the same type **Graph.Node**; i.e. **Node** is prefixed with class **Graph**. In Scala such a type can be expressed as well, it is written **Graph#Node**. If we want to be able to connect nodes of different graphs, we have to change the definition of our initial graph implementation in the following way:

```
class Graph (val graphID: Int = 0) {
  val thisGraph = this
  class Node(val nodeID: Int = 0) {
    val graphID: Int = thisGraph.graphID
    var connectedNodes: List[Graph#Node] = Nil
    def connectTo(node: Graph#Node) {
      if (connectedNodes.find(node.equals).isEmpty) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  var nodes: List[Node] = Nil
  def newNode(nodeID: Int = 0): Node = {
    val res = new Node(nodeID)
    nodes = res :: nodes
    res
  }
}

val graph1: Graph = new Graph(1)
val node1: graph1.Node = graph1.newNode(1)
val node2: graph1.Node = graph1.newNode(2)
node1.connectTo(node2)      // legal
val graph2: Graph = new Graph(2)
val node3: graph2.Node = graph2.newNode(3)
node1.connectTo(node3)
```

Note that this program doesn't allow us to attach a node to two different graphs. If we want to remove this restriction as well, we have to change the type of variable nodes to **Graph#Node**.

Exercise

11.21 Abstract Types

Traits and abstract classes can have an abstract type member. This means that the concrete implementations define the actual type. Here's an example:

```
trait Buffer {
  type T
  val element: T
}
```

Here we have defined an abstract type **T**. It is used to describe the type of element. We can extend this trait in an abstract class, adding an upper-type-bound to **T** to make it more specific.

```
abstract class SeqBuffer extends Buffer {
  type U
  type T <: Seq[U]
  def length = element.length
}
```

Notice how we can use yet another **abstract type U** as an upper-type-bound. This **class SeqBuffer** allows us to store only sequences in the buffer by stating that type **T** has to be a subtype of **Seq[U]** for a new abstract type **U**.

Traits or classes with abstract type members are often used in combination with anonymous class instantiations. To illustrate this, we now look at a program which deals with a sequence buffer that refers to a list of integers:

```
abstract class IntSeqBuffer extends SeqBuffer {
  type U = Int
}


def newIntSeqBuf(elem1: Int, elem2: Int): IntSeqBuffer =
  new IntSeqBuffer {
       type T = List[U]
       val element = List(elem1, elem2)
     }
val buf = newIntSeqBuf(7, 8)
println("length = " + buf.length)         //length = 2
println("content = " + buf.element)       //content = List(7, 8)
```

Here the factory **newIntSeqBuf** uses an anonymous class implementation of **IntSeqBuf** (i.e. new **IntSeqBuffer**), setting type **T** to a **List[Int]**.

```
abstract class Buffer[+T] {
  val element: T
}
abstract class SeqBuffer[U, +T <: Seq[U]] extends Buffer[T] {
  def length = element.length
}
```

```
def newIntSeqBuf(e1: Int, e2: Int): SeqBuffer[Int, Seq[Int]] =
  new SeqBuffer[Int, List[Int]] {
    val element = List(e1, e2)
  }

val buf = newIntSeqBuf(7, 8)
println("length = " + buf.length)
println("content = " + buf.element)
```

Note that we have to use variance annotations here (**+T <: Seq[U]**) in order to hide the concrete sequence implementation type of the object returned from method **newIntSeqBuf**. Furthermore, there are cases where it is not possible to replace abstract types with type parameters.

Exercise

11.22 Compound Types

Sometimes it is necessary to express that the type of an object is a subtype of several other types. In Scala this can be expressed with the help of *compound types*, which are intersections of object types.

Suppose we have two traits **Cloneable** and **Resetable**:

```
trait Cloneable extends java.lang.Cloneable {
  override def clone(): Cloneable = {
    super.clone().asInstanceOf[Cloneable]
  }
}
trait Resetable {
  def reset: Unit
}
```

Now suppose we want to write a function **cloneAndReset** which takes an object, clones it and resets the original object:

```
def cloneAndReset(obj: ?): Cloneable = {
  val cloned = obj.clone()
  obj.reset
  cloned
}
```

The question arises what the type of the parameter **obj** is. If it's **Cloneable** then the object can be **clone**d, but not **reset**; if it's **Resetable** we can reset it, but there is no clone operation. To avoid type casts in such a situation, we can specify the type of **obj** to be both **Cloneable** and **Resetable**. This compound type is written like this in Scala: **Cloneable with Resetable**.

Here's the updated function:

```
def cloneAndReset(obj: Cloneable with Resetable): Cloneable = {
  val cloned = obj.clone()
  obj.reset
  cloned
}
```

Compound types can consist of several object types and they may have a single refinement which can be used to narrow the signature of existing object members. The general form is: **A with B with C ... refinement**

An example for the use of refinements is given on the page about abstract types.

Exercise

11.23 Self-type

Self-types are a way to declare that a trait must be mixed into another trait, even though it doesn't directly extend it. That makes the members of the dependency available without imports.

A self-type is a way to narrow the type of **this** or another identifier that aliases **this**. The syntax looks like normal function syntax but means something entirely different.

To use a self-type in a trait, write an identifier, the type of another trait to mix in, and a **=>** (e.g. **someIdentifier: SomeOtherTrait =>**).

```
trait User {
  def username: String
}

trait Tweeter {
  this: User =>  // reassign this
  def tweet(tweetText: String) = println(s"$username: $tweetText")
}

class VerifiedTweeter(val username_ : String) extends Tweeter with User {  // We mix
def username = s"real $username_"
}

val realBeyonce = new VerifiedTweeter("Beyonce")
realBeyonce.tweet("Just spilled my glass of lemonade")
//real Beyonce: Just spilled my glass of lemonade
```

Because we said **this: User =>** in trait **Tweeter**, now the variable **username** is in scope for the **tweet** method. This also means that since **VerifiedTweeter** extends **Tweeter**, it must also mix-in **User** (using **with User**).

Exercise

11.24 Implicit Parameters

A method can have an *implicit* parameter list, marked by the **implicit** keyword at the start of the parameter list. If the parameters in that parameter list are not passed as usual, Scala will look if it can get an implicit value of the correct type, and if it can, pass it automatically.

The places Scala will look for these parameters fall into two categories:

- Scala will first look for implicit definitions and implicit parameters that can be accessed directly (without a prefix) at the point the method with the implicit parameter block is called.

- Then it looks for members marked implicit in all the companion objects associated with the implicit candidate type.

A more detailed guide to where scala looks for implicits can be found in the FAQ

In the following example we define a method **sum** which computes the sum of a list of elements using the **Monoid**'s **add** and **unit** operations. Please note that implicit values can not be top-level.

```
abstract class Monoid[A] {
  def add(x: A, y: A): A
  def unit: A
}

object ImplicitTest {
  implicit val stringMonoid: Monoid[String] = new Monoid[String] {
    def add(x: String, y: String): String = x concat y
    def unit: String = ""
  }

  implicit val intMonoid: Monoid[Int] = new Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }

  def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
    if (xs.isEmpty) m.unit
    else m.add(xs.head, sum(xs.tail))

  def main(args: Array[String]): Unit = {
    println(sum(List(1, 2, 3)))       // uses IntMonoid implicitly
    println(sum(List("a", "b", "c"))) // uses StringMonoid implicitly
  }
}
```

Monoid defines an operation called **add** here, that combines a pair of **A**s and returns another **A**, together with an operation called **unit** that is able to create some (specific) **A**.

To show how implicit parameters work, we first define monoids **StringMonoid** and **IntMonoid** for strings and integers, respectively. The **implicit** keyword indicates that the corresponding object can be used **implicitly**.

The method **sum** takes a **List[A]** and returns an **A**, which takes the initial **A** from unit, and combines each next **A** in the list to that with the add method. Making the parameter **m** implicit here means we only have to provide the **xs** parameter when we call the method if Scala can find an **implicit Monoid[A]** to use for the implicit **m** parameter.

In our **main** method we call **sum** twice, and only provide the **xs** parameter. Scala will now look for an implicit in the scope mentioned above. The first call to **sum** passes a **List[Int]** for **xs**, which means that **A** is Int. The implicit parameter list with **m** is left out, so Scala will look for an implicit of type **Monoid[Int]**. The first lookup rule reads

> Scala will first look for implicit definitions and implicit parameters that can be accessed directly (without a prefix) at the point the method with the implicit parameter block is called.

intMonoid is an implicit definition that can be accessed directly in **main**. It is also of the correct type, so it's passed to the sum method automatically.

The second call to **sum** passes a **List[String]**, which means that **A** is **String**. Implicit lookup will go the same way as with **Int**, but will this time find **stringMonoid**, and passes that automatically as **m**.

The program will output

```
6
abc
```

Exercise

11.25 Implicit Conversions

An implicit conversion from type **S** to type **T** is defined by an implicit value which has function type **S => T**, or by an implicit method convertible to a value of that type.

Implicit conversions are applied in two situations:

- If an expression **e** is of type **S**, and **S** does not conform to the expression's expected type **T**.
- In a selection **e.m** with **e** of type **S**, if the selector **m** does not denote a member of **S**.

In the first case, a conversion **c** is searched for which is applicable to **e** and whose result type conforms to **T**. In the second case, a conversion **c** is searched for which is applicable to **e** and whose result contains a member named **m**.

If an implicit method **List[A] => Ordered[List[A]]** is in scope, as well as an implicit method **Int => Ordered[Int]**, the following operation on the two lists of type **List[Int]** is legal:

```
List(1, 2, 3) <= List(4, 5)
```

An implicit method **Int => Ordered[Int]** is provided automatically through scala.Predef.intWrapper. An example of an implicit method **List[A] => Ordered[List[A]]** is provided below.

```
import scala.language.implicitConversions

implicit def list2ordered[A](x: List[A])
    (implicit elem2ordered: A => Ordered[A]): Ordered[List[A]] =
  new Ordered[List[A]] {
    //replace with a more useful implementation
    def compare(that: List[A]): Int = 1
  }
//list2ordered: [A](x: List[A])(implicit elem2ordered: A => Ordered[A])Ordered[List[
```

The implicitly imported object **scala.Predef** declares several predefined types (e.g. **Pair**) and methods (e.g. **assert**) but also several implicit conversions.

For example, when calling a Java method that expects a **java.lang.Integer**, you are free to pass it a **scala.Int** instead. That's because Predef includes the following implicit conversions:

```
import scala.language.implicitConversions

implicit def int2Integer(x: Int) =
  java.lang.Integer.valueOf(x)
```

Because implicit conversions can have pitfalls if used indiscriminately the compiler warns when compiling the implicit conversion definition.

To turn off the warnings take either of these actions:

- Import **scala.language.implicitConversions** into the scope of the implicit conversion definition
- Invoke the compiler with **-language:implicitConversions**

No warning is emitted when the conversion is applied by the compiler.
Exercise

11.26 Polymorphic Methods

In methods of Scala, type can be parametrized as well as value. The syntax is similar to that of generic classes. Type parameters are enclosed in square brackets, while value parameters are enclosed in parentheses.

Here is an example:

```
def listOfDuplicates[A](x: A, length: Int): List[A] = {
  if (length < 1)
    List()                              //or Nil
  else
    x :: listOfDuplicates(x, length - 1)
}
println(listOfDuplicates[Int](3, 4))  //List(3, 3, 3, 3)
println(listOfDuplicates("La", 8))    //List(La, La, La, La, La, La, La, La)
```

The method **listOfDuplicates** takes a type parameter **A** and value parameters **x** and **length**. Value **x** is of type **A**. If **length < 1** we return an empty list. Otherwise we prepend **x** to the the list of duplicates returned by the recursive call. (Note that **::** means prepend an element on the left to a list on the right.)

In first example call, we explicitly provide the type parameter by writing **[Int]**. Therefore the first argument must be an **Int** and the return type will be **List[Int]**.

The second example call shows that you don't always need to explicitly provide the type parameter. The compiler can often infer it based on context or on the types of the value arguments. In this example, **"La"** is a **String** so the compiler knows **A** must be **String**.
Exercise

11.27 Type Inference

The Scala compiler can often infer the type of an expression so you don't have to declare it explicitly.

11.27.1 Omitting the type

```
val businessName = "Montreux Jazz Ce"
```

The compiler can detect that **businessName** is a **String**. It works similarly with methods:

```
def fac(n: Int) = if (n == 0) 1 else n * fac(n - 1)
```

It is also not compulsory to specify type parameters when polymorphic methods are called or generic classes are instantiated. The Scala compiler will infer such missing type parameters from the context and from the types of the actual method/constructor parameters.

Here are two examples:

```
case class MyPair[A, B](x: A, y: B);
val p = MyPair(1, "scala")          //p: MyPair[Int,String] = MyPair(1,scala)

def id[T](x: T) = x
val q = id(1)                       //q: Int = 1
```

The compiler uses the types of the arguments of **MyPair** to figure out what type TEXTBFA and TEXTBFB are. Likewise for the type of TEXTBFx.

11.27.2 Parameters

The compiler never infers method parameter types. However, in certain cases, it can infer anonymous function parameter types when the function is passed as argument.

```
val a = Seq(1, 3, 4).map(x => x * 2)            //a: Seq[Int] = List(2, 6, 8)
```

The parameter for map is **f: A => B**. Because we put integers in the **Seq**, the compiler knows that **A** is Int (i.e. that **x** is an integer). Therefore, the compiler can infer from **x * 2** that **B** is type **Int**.

11.27.3 When not to rely on type inference

It is generally considered more readable to declare the type of members exposed in a public API. Therefore, we recommended that you make the type explicit for any APIs that will be exposed to users of your code.

Also, type inference can sometimes infer a too-specific type. Suppose we write:

```
var obj = null                      //obj: Null = null
```

We can't then go on and make this reassignment:

```
obj = new AnyRef                    //<console>:14: error: type mismatch;
                                    // found   : Object
                                    // required: Null
                                    // obj = new AnyRef
                                    //       ^
var obj1: AnyRef = null             //obj1: AnyRef = null
obj1 = new AnyRef                   //obj1: AnyRef = java.lang.Object@59d29065
```

It won't compile, because the type inferred for **obj** was **Null**. Since the only value of that type is null, it is impossible to assign a different value.

Exercise

11.28 Operators

In Scala, operators are methods. Any method with a single parameter can be used as an *infix operator*. For example, + can be called with dot-notation:

```
10.+(1)                                  //11
```

However, it's easier to read as an infix operator:

```
10 + 1                                  //11
```

11.28.1 Defining and using operators

You can use any legal identifier as an operator. This includes a name like **add** or a symbol(s) like **+**.

```
case class Vec(val x: Double, val y: Double) {
  def +(that: Vec) = new Vec(this.x + that.x, this.y + that.y)
}

val vector1 = Vec(1.0, 1.0)
val vector2 = Vec(2.0, 2.0)

val vector3 = vector1 + vector2
vector3.x                               // 3.0
vector3.y                               // 3.0
```

The class Vec has a method **+** which we used to add **vector1** and **vector2**. Using parentheses, you can build up complex expressions with readable syntax. Here is the definition of **class MyBool** which includes methods **and** and or:

```
case class MyBool(x: Boolean) {
  def and(that: MyBool): MyBool = if (x) that else this
  def or(that: MyBool): MyBool = if (x) this else that
  def negate: MyBool = MyBool(!x)
}
```

It is now possible to use **and** and **or** as infix operators:

```
def not(x: MyBool) = x.negate
def xor(x: MyBool, y: MyBool) = (x or y) and not(x and y)
```

11.28.2 Precedence

When an expression uses multiple operators, the operators are evaluated based on the priority of the first character:

```
(characters not shown below)
* / %
+ -
:
```

```
= !
< >
&
^
|
(all letters)
```

This applies to functions you define. For example, the following expression:

```
a + b ^? c ?^ d less a ==> b | c
```

is equivalent to

```
((a + b) ^? (c ?^ d)) less ((a ==> b) | c)
```

?

has the highest precedence because it starts with the character **?**. **+** has the second highest precedence, followed by **?^**, **==>**, **|**, and **less**.

Exercise

11.29 By-name Parameters

By-name parameters are only evaluated when used. They are in contrast to *by-value parameters*. To make a parameter called by-name, simply prepend **=>** to its type.

```
def calculate(input: => Int) = input * 37
```

By-name parameters have the advantage that they are not evaluated if they aren't used in the function body. On the other hand, by-value parameters have the advantage that they are evaluated only once.

Here's an example of how we could implement a while loop:

```
def whileLoop(condition: => Boolean)(body: => Unit): Unit =
  if (condition) {
    body
    whileLoop(condition)(body)
  }

var i = 2

whileLoop (i > 0) {
  println(i)
  i -= 1
}                                          //2 1
```

The method **whileLoop** uses multiple parameter lists to take a condition and a body of the loop. If the **condition** is true, the body is executed and then a recursive call to **whileLoop** is made. If the **condition** is false, the body is never evaluated because we prepended **=>** to the type of body.

Now when we pass **i > 0** as our condition and **println(i); i-= 1** as the **body**, it behaves like the standard while loop in many languages.

This ability to delay evaluation of a parameter until it is used can help performance if the parameter is computationally intensive to evaluate or a longer-running block of code such as fetching a URL.

11.30 Annotations

Annotations associate meta-information with definitions. For example, the annotation **@deprecated** before a method causes the compiler to print a warning if the method is used.

```
object DeprecationDemo extends App {
  @deprecated("This method will be removed","Library 0.1")
  def hello = "hola"

  hello
}  //warning: there was one deprecation warning; re-run with -deprecation for detail
```

This will compile but the compiler will print a warning: "there was one deprecation warning".

An annotation clause applies to the first definition or declaration following it. More than one annotation clause may precede a definition and declaration. The order in which these clauses are given does not matter.

11.30.1 Annotations that ensure correctness of encodings

Certain annotations will actually cause compilation to fail if a condition(s) is not met. For example, the annotation **@tailrec** ensures that a method is

tail-recursive. *Tail-recursion* can keep memory requirements constant. Here's how it's used in a method which calculates the factorial:

```
import scala.annotation.tailrec

def factorial(x: Int): Int = {

  @tailrec
  def factorialHelper(x: Int, accumulator: Int): Int = {
    if (x == 1) accumulator else factorialHelper(x - 1, accumulator * x)
  }
  factorialHelper(x, 1)
}
```

The factorialHelper method has the **@tailrec** which ensures the method is indeed tail-recursive. If we were to change the implementation of **factorialHelper** to the following, it would fail:

```
import scala.annotation.tailrec

def factorial(x: Int): Int = {
  @tailrec
  def factorialHelper(x: Int): Int = {
    if (x == 1) 1 else x * factorialHelper(x - 1)
  }
  factorialHelper(x)
}
//<console>:15: error: could not optimize @tailrec annotated method factorialHelper:
//           if (x == 1) 1 else x * factorialHelper(x - 1)
//                                  ^
```

We would get the message "Recursive call not in tail position".

11.30.2 Annotations affecting code generation

Some annotations like **@inline** affect the generated code (i.e. your jar file might have different bytes than if you hadn't used the annotation). Inlining means inserting the code in a method's body at the call site. The resulting bytecode is longer, but hopefully runs faster. Using the annotation **@inline** does not ensure that a method will be inlined, but it will cause the compiler to do it if and only if some heuristics about the size of the generated code are met.

11.30.3 Java Annotations

When writing Scala code which interoperates with Java, there are a few differences in annotation syntax to note. Note: Make sure you use the **-target:jvm-1.8** option with Java annotations.

Java has user-defined metadata in the form of annotations. A key feature of annotations is that they rely on specifying name-value pairs to initialize their elements. For instance, if we need an annotation to track the source of some class we might define it as

```
@interface Source {
  public String URL();
  public String mail();
}
```

And then apply it as follows

```
@Source(URL = "http://coders.com/",
        mail = "support@coders.com")
public class MyClass extends HisClass ...
```

An annotation application in Scala looks like a constructor invocation, for instantiating a Java annotation one has to use named arguments:

```
@Source(URL = "http://coders.com/",
        mail = "support@coders.com")
class MyScalaClass ...
```

This syntax is quite tedious if the annotation contains only one element (without default value) so, by convention, if the name is specified as value it can be applied in Java using a constructor-like syntax:

```
@interface SourceURL {
    public String value();
    public String mail() default "";
}
```

And then apply it as follows

```
@SourceURL("http://coders.com/")
public class MyClass extends HisClass ...
```

In this case, Scala provides the same possibility

```
@SourceURL("http://coders.com/")
class MyScalaClass ...
```

The mail element was specified with a default value so we need not explicitly provide a value for it. However, if we need to do it we can not mix-and-match the two styles in Java:

```
@SourceURL(value = "http://coders.com/",
           mail = "support@coders.com")
public class MyClass extends HisClass ...
```

Scala provides more flexibility in this respect

```
@SourceURL("http://coders.com/",
           mail = "support@coders.com")
    class MyScalaClass ...
```

11.31 Default Parameter Values

Scala provides the ability to give parameters default values that can be used to allow a caller to omit those parameters.

```
def log(message: String, level: String = "INFO") = println(s"$level: $message")

log("System starting")              //INFO: System starting
log("User not found", "WARNING")    //WARNING: User not found
```

The parameter **level** has a default value so it is optional. On the last line, the argument **"WARNING"** overrides the default argument **"INFO"**. Where you might do overloaded methods in Java, you can use methods with optional parameters to achieve the same effect. However, if the caller omits an argument, any following arguments must be named.

```
class Point(val x: Double = 0, val y: Double = 0)

val point1 = new Point(y = 1)
```

Here we have to say **y = 1**.

Note that default parameters in Scala are not optional when called from Java code:

```
// Point.scala
class Point(val x: Double = 0, val y: Double = 0)


// Main.java
public class Main {
    public static void main(String[] args) {
        Point point = new Point(1);             // does not compile
    }
}
```

11.32 Named Arguments

When calling methods, you can label the arguments with their parameter names like so:

```
def printName(first: String, last: String): Unit = {
  println(first + " " + last)
}

printName("John", "Smith")                  //John Smith
printName(first = "John", last = "Smith")  //John Smith
printName(last = "Smith", first = "John")  //John Smith
```

Notice how the order of named arguments can be rearranged. However, if some arguments are named and others are not, the unnamed arguments must come first and in the order of their parameters in the method signature.

```
def printName(first: String, last: String): Unit = {
  println(first + " " + last)
}

printName(last = "Smith", "john")
//<console>:13: error: positional after named argument.
//       printName(last = "Smith", "john")
//                                 ^
```

Note that named arguments do not work with calls to Java methods.

11.33 Packages and Imports

Scala uses packages to create namespaces which allow you to modularize programs.

11.33.1 Creating a package

Packages are created by declaring one or more package names at the top of a Scala file.

```
package users

class User
```

One convention is to name the package the same as the directory containing the Scala file. However, Scala is agnostic to file layout. The directory structure of an **sbt** project for **package users** might look like this:

```
- ExampleProject
  - build.sbt
  - project
  - src
    - main
      - scala
```

```
        - users
          User.scala
          UserProfile.scala
          UserPreferences.scala
      - test
```

Notice how the **users** directory is within the **scala** directory and how there are multiple Scala files within the package. Each Scala file in the package could have the same package declaration. The other way to declare packages is by using braces:

```
package users {
  package administrators {
    class NormalUser
  }
  package normalusers {
    class NormalUser
  }
}
```

As you can see, this allows for package nesting and provides greater control for scope and encapsulation.

The package name should be all lower case and if the code is being developed within an organization which has a website, it should be the following format convention: **<top-level-domain>.<domain-name>.<project-name>**. For example, if Google had a project called SelfDrivingCar, the package name would look like this:

```
package com.google.selfdrivingcar.camera

class Lens
```

This could correspond to the following directory structure: **SelfDrivingCar/src/main/scala/com/google/selfdrivingc**

11.33.2 Imports

import clauses are for accessing members (**class**es, **trait**s, **function**s, etc.) in other packages. An **import** clause is not required for accessing members of the same package. Import clauses are selective:

```
import users._                          // import everything from the users pac
import users.User                       // import the class User
import users.{User, UserPreferences}    // Only imports selected members
import users.{UserPreferences => UPrefs} // import and rename for convenience
```

One way in which Scala is different from Java is that imports can be used anywhere:

```
def sqrtplus1(x: Int) = {
  import scala.math.sqrt
  sqrt(x) + 1.0
}
```

In the event there is a naming conflict and you need to import something from the root of the project, prefix the package name with **__root__**:

```
package accounts

import __root__.users._
```

Note: The **scala** and **java.lang** packages as well as **object Predef** are imported by default; in other words, the following **import**s are automatically executed.

```
import java.lang._
import scala._
import scala.Predef._
```

Bibliography

1) D. J. Wilkinson, Statistical computing with scala: A functional approach to data science (2017).
URL https://github.com/darrenjw/scala-course

2) Tour of scala (2018).
URL https://docs.scala-lang.org/tour/tour-of-scala.html

3) A. Alexander, How to execute (exec) external system commands in scala.
URL https://alvinalexander.com/scala/scala-execute-exec-external-system-commands-in

4) Sbt documentation (2018).
URL https://www.scala-sbt.org/documentation.html

5) Powerful new number types and numeric abstractions for scala. (2018).
URL https://github.com/non/spire

6) Visual scala reference (2018).
URL https://superruzafa.github.io/visual-scala-reference/

7) Linear algebra cheat sheet (2018).
URL https://github.com/scalanlp/breeze/wiki/

8) QR decomposition (2018).
URL https://en.wikipedia.org/wiki/QR_decomposition

9) D. J. Wilkinson, Scala library for regression modelling (fitting linear and generalised linear statistical models, diagnosing fit, making predictions) (2018).
URL https://github.com/darrenjw/scala-glm

10) D. B. Dahl, Integration of **R** and **Scala** using rscala (2017).
URL https://github.com/dbdahl/rscala

11) D. J. Wilkinson, Darren wilkinson's research blog (2018).
URL https://darrenjw.wordpress.com/

12) Spark documentations (2018).
URL https://spark.apache.org/documentation.html

13) Simulacrum (2018).
URL https://github.com/mpilquist/simulacrum

14) Online learning (2018).
URL https://www.coursera.org/learn/scala-spark-big-data

UTF8min