Visualisation

Introduction

Julia has no built-in graphics commands. This means that it is not possible to create some datasets and issue a plot command without first installing and loading a package.

One reason for this is that Julia needs to build from source a variety of different operating systems and any libraries that are shipped, such as OpenBLAS and LibUV, must as be in source form and not interfere with the building process. Graphics engines have a variety of different backends such as Gtk, Qt, and, whereas specialist packages may be restricted in their OS support, the overall Julia system may not.

At first, the inclusion of built-in graphics was seen as a long term goal and one that would be added in future releases. However it seems that this is not now so. With the introduction of the Plots API (which we saw in the previous chapter) it is possible to use a uniform syntax for a variety of backends (PyPlot, GR, PlotlyJS), all of which we will look at later in this chapter.

There are a few packages which do not conform to the Plots API and we will look at as well, in particular the popular packages Winston and Gadfly.

An additional point to notice is that the Julia method of importing symbols into its main namespace via the using command means that most graphics packages, which tend to have functions such as plot() and display(), do not produce a name clash.

Of course, it is possible to use import and fully qualify any function call which is an approach I will adopt in this chapter. Normally when you are working with a single package the simple using statement can be employed.

Finally in this chapter will look at some graphic frameworks such as StatPlots and Makie, which build on the Plots API to provide an high-level interface to simplify the creation of complex visualisations. Visualisation

Basic Graphics in Julia

We have seen in earlier chapters some of the important modules used for creating graphics and in this section we will meet some of these again and in addition introduce a few more of the most widely used. All of the packages discussed below are now very sophisticated, and so the examples provided only skim the surface of their capabilities, so the reader is encourage to review all the accompanying documentation for any individual package.

Inline Text Graphics

In the overview of Julia, in chapter 1, we briefly saw that it is possible to create some quite sophisticated text graphics (i.e. using printable characters) by means of the UnicodePlots package. This is the successor to early packages such as ASCIIPlots and TextPlots; it provides a wider variety of available graphic types of display.

UnicodePlots, as its name suggests is capable of using unicode characters, in addition to simple ASCII ones, and hence can create a wider variety of graphs such as:

Scatterplot Lineplot Staircase Plot Barplot (horizontal) Histogram (horizontal) Boxplot (horizontal) Sparsity Pattern Density Plot

As an example of a simple line plot consider the following damped sinusoid in the 0:3π range; which we can pass as an anonymous function to the lineplot routine:

julia> import UnicodePlots

julia> const ui = UnicodePlots

julia> ui.lineplot(x->x\*sin(3x)\*exp(-0.3x),0,3pi)

As we saw in the list above UnicodePlots is not limited line and scatterplots plots alone and here are a couple of examples of a frequency histogram and a box plot

#=

Create 1000 Gaussian numbers and display the frequency histogram for 15 bins on a log scale

=#

julia> ui.histogram(randn(1000) .\* 0.1,

nbins = 15, closed = :right, xscale=log10)

# Display a simple box plot of two variables

julia> A = [1,3,5, 2];

julia> B = [2,4,8,10,7];

julia> ui.boxplot(["A", "B"], [A, B], title="Boxplot of A & B")

The few examples we have provided here utilise the high-level interface to UnicodePlots.

These are layered on top of a more flexible low-level interface, which is also available to the programmer and provides a variety of "canvases", such as ASCII, Block, Braille etc., each with differing capabilities. The reader should consult the documentation for more information

Luxor

Most graphic packages are raster-style, where pixels are set (coloured) within a canvas. However Luxor is an alternative for drawing static vector graphics. The advantage of vector based graphics is that they are easily rescaled by just expanding (or reducing) the dimensions the viewpoint.

It is a high-level interface to the Cairo.jl module (hence the name) and provides more simple basic drawing functions for working with shapes, polygons, clipping masks, PNG images, turtle graphics, animations, and shapefiles.

In addition once a drawing is completed it can easily br saved into PDF, PNG, SVG, or EPS files.

The package contains a few macros to test the installation such as : @svg and @png. Try the following which should display the Julia circles logo

julia> using Luxor

julia> @png juliacircles()

Sierpinski Triangles

As an example lets use Luxor to display a drawingg based on Sierpinski triangles.

This is a fractal described in 1915 by Waclaw Sierpinski and is a self similar structure that occurs at different levels of iterations, or magnifications.

It is constructed from an equilateral triangle by repeated removal of triangular subsets:

1. Start with an equilateral triangle.

2. Subdivide it into four smaller congruent equilateral triangles and remove the central triangle.

3. Repeat step 2 with each of the remaining smaller triangles forever.

To develop the code in Julia we first use/import the packages we need

julia> using Random, Printf, Colors

julia> import Luxor

julia> const lx = Luxor

Next define a function to create a triangle and use this to buildup the Sierpinski triangle

julia> function triangle(points, degree)

lx.sethue(cols[degree])

lx.poly(points, :fill)

end

julia> function sierpinski(points, degree)

triangle(points, degree)

if degree > 1

p1, p2, p3 = points

sierpinski([p1, lx.midpoint(p1, p2), lx.midpoint(p1, p3)], degree-1)

sierpinski([p2, lx.midpoint(p1, p2), lx.midpoint(p2, p3)], degree-1)

sierpinski([p3, lx.midpoint(p3, p2), lx.midpoint(p1, p3)], degree-1)

end

end

Next draw it and preview the result.

julia> function draw(n)

lx.circle(lx.O, 100, :clip)

points = lx.ngon(lx.O, 150, 3, -pi/2, vertices=true)

sierpinski(points, n)

end

julia> lx.Drawing(400, 250)

julia> lx.background("white")

julia> lx.origin()

julia> depth = 8

julia> cols = distinguishable\_colors(depth)

# from Colors.jl

julia> draw(depth) julia> lx.finish()

julia> lx.preview()

Turtle Graphics

Luxor incorporates some simple "turtle graphics" functions.

The routines to control the turtle begin with a capital letter: Forward, Turn, Circle, Orientation, Rectangle, Pendown, Penup, Pencolor, Penwidth, and Reposition etc. , and angles are specified in degrees.

Below is an example using these to create a simple drawing

import Luxor lx = Luxor

lx.Drawing(600, 400, "turtles.png")

lx.origin()

lx.background("midnightblue")

tur = lx.Turtle()

lx.Pencolor(tur, "cyan")

lx.Penwidth(tur, 1.5)

n = 5

for i in 1:400

global n

Forward(tur, n)

Turn(tur, 89.5)

HueShift(tur)

n += 0.75

end

lx.fontsize(20)

lx.finish()

Winston

Winston is a 2-D plotting packages which has been available since the earliest days of Julia. It has fallen out of favour in recent days and does not conform toe the Plots API, which we will meet later in the chapter. One of the reasons is the difficulty of installing backend support on certain platforms, nevertheless is compliant with version 1.0 and is a particular favourite of mine, so I make no apology for discussing it here.

The typical usage we have already seen is via the plot() function :-

julia> import Winston;

julia> const wn = Winston

julia> t = collect(range(0, stop=4pi, length=1000));

# Define 3 functions and create arrays based on the t

# ... variate and display them

julia> f(x::Array) = 10x .\* exp.(-0.3x) .\* sin.(3x);

julia> g(x::Array) = 0.1x.\*(2pi .- x).\*(4pi .- x);

julia> h(x::Array) = 10.0 ./ (1 .+ x.\*x);

julia> y1 = f(t); y2 = g(t); y3 = h(t)

julia> wn.plot(t,y1,"b",t,y2,"r--",t,y3,"k;")

Alternatively, usefplot()and define the function f() directly that produces the same plot as shown in the following:

julia> wn.fplot(x -> 10\*x.\*exp(-0.3\*x).\*sin.(3\*x), [0,4pi], "b—")

In addition, there is a plothist() function that can take the result of the histo() function.

The following code generates a set of normally distributed numbers and displays the frequency histogram for 100 intervals:

julia> x = randn(10000);

julia> wn.plothist(x, nbins=50)

Also we can use Winston to create log-log and semi-log plots.

# Plot y1 against log(t)

julia> wn.semilogx(t,y1)

julia> wn.title("log(t) vs 10x \* exp(-0.3x) \* sin(3x)")

# ... and a phase plane plot of y2 vs log(y3)

julia>wn.semilogy(y2,y3)

julia> wn.title( "0.1\*(2\\pi - x)\*(4\\pi - x) vs log(1 /( 1 + x\*x))")

# Note the Text style use of \\pi to plot π

To conclude this brief introduction to Winston let's look at a more complex example using a Framed Plot. This allows parameters of the graphic to be set and then curves, titles etc., set prior to display.

#=

Create a Framed Plot and a linear relationship between two variables dithering via a random Gaussian variate

=#

julia> p = wn.FramedPlot(aspect\_ratio=1, xrange=(-10,110), yrange=(-10,110));

julia> n = 21;

julia> x = collect(range(0.0, length=n, stop=100.0));

julia> yA = 10.0\*randn(n) .+ 40.0;

julia> yB = x .+ 5.0\*randn(n);

# Set labels and symbol styles julia> a = wn.Points(x, yA, kind="circle"); julia> wn.setattr(a,label="'a' points"); julia> b = wn.Points(x, yB); julia> wn.setattr(b,label="'b' points"); julia> wn.style(b, kind="filled circle");

# Plot a line which 'fits' through the yB points # ... and add a legend in the top LHS part of the graph julia> s = wn.Slope(1, (0,0), kind="dotted"); julia> wn.setattr(s, label="slope");

julia> lg = wn.Legend(.1, .9, Any[a,b,s] ); julia> wn.add(p, s, a, b, lg);

# Now display the completed graphic julia> wn.display(p)

Finally the plot can be save to a disk file using the julia> wn.savefig(p, "MyWPlot.png")

statement:

PyPlot

PyPlot is a part of the work of Steven Johnson of MIT, which arose from the previous development of the PyCall module. We have used it quite extensively in the previous chapters and will take little time to discuss it further here.

Note that PyPlot is one of the graphics packages which can used as a backend for the Plots API, others being GR and PlotlyJS, which we will meet later in this chapter

PyPloy provides an interface to the matplotlib plotting library from Python; therefore in order to use it, the installation of Python and matplotlib is necessary. If this is successful, it will work either by creating an independent window (via Python) or embedding in an IJulia workbook.

I found that the easiest way to install both Python and matplotlib is using the Anaconda distribution from continuum.io.

This works on all three common platforms Windows, OS X, and Linux.

For a full discussion, and any problems relating to the installation, the reader is referred to Github online documentation of PyPlot.jl

As a first example, I have picked one from the early PyPlot documentation that of a sinusoidally modulated sinusoid.

The following code creates the code, displays it, via a native Python window (from the REPL) or inline graphics in a Jupyter notebook, and also writes the disk as an SVG file.

julia> import PyPlot julia> const py = PyPlot julia> x = collect(range(0.0,stop=2pi,length=1000)) julia> y = sin.(3\*x + 4\*cos.(2\*x)); julia> py.title("A sinusoidally modulated sinusoid"); julia> py.plot(x,y,color="red",linewidth=2.0,linestyle="--"); julia> savefig("sinusoid.svg");

The PyPlot package also imports functions from Matplotlib's mplot3d toolkit.

Unlike matplotlib, however, you can create 3D plots directly without first creating an Axes3d object, simply by calling bar3D, contour3D, contourf3D, plot3D, scatter3D etc.

PyPlot also exports the MATLAB-like synonyms such as surf for plot\_surface and mesh for plot\_wireframe.

The following is a simple 3D surface using the following code:

julia> y = collect(range(0,stop=3π,length=250)) julia> py.surf(y, y, y .\* sin.(y) .\* cos.(y)' .\* exp.(-0.4y))

As a final example, let's create a more substantial display with axes, titles, and annotations using the XKCD comic mode.

The module includes an xkcd() call to switch to this mode :-

julia> py.xkcd() julia> x = collect(range(1, length=101, stop=10)); julia> y = sin.(3x + cos.(5x)) julia> p = py.plot(x,y)

Gadfly

Gadfly is a large and complex package, and provides great flexibility in the range and breadth of the visualizations possible in Julia. It is equivalent to the ggplot2 R module and similarly is based on the The Grammar of Graphics seminal work by Leland Wilkinson.

Together with Winston and PyPlot, Gadfly is one of the earliest visualisation packages which has stood the longevity test.

The package is a heavyweight, using a large set of modules, taking a long time to compile and also to create displays but this is compensated by the variety of visualisations which can be produced. Gadfly has is own website gadfly.org which demonstrates many of these.

Once created Gadfly can render the graphics to publication quality, outputting as SVG, PNG, Postscript, and PDF.

The following code creates a scatter diagram (the default) of 100 random points as an SVG file, which can be viewed in a standard web browser.

julia> using Gadfly julia> dd = plot(x = rand(100), y = rand(100)); julia> draw(SVG("random-pts.svg", 15cm, 12cm) , dd);

Note the Gadfly works with the Jupyter (via IJulia) displaying in notebook rather than a separate window; also it has a tight integration with Julia data frames.

The ability to work directly with data frames is especially useful. To illustrate this, let's look at the GCSE result set we investigated in Chapter 6, Working with Data. Recall that this is available as part of the RDatasets suite of source data.

julia> using Gadfly, RDatasets, DataFrames; julia> mlmf = dataset("mlmRev","Gcsemv") julia> df = mlmf[completecases(mlmf), :] 1,523 rows × 5 columns

School Student Gender Float64 Float64 1Categorical…Categorical…Categorical…

20920 27 F 39.0 2 20920 31 F 36.0 3 20920 42 M 16.0 4 20920 101 F 49.0 5 20920 113 M 25.0 6 22520 1 F 48.0

Written

Course

76.8

87.9

44.4

89.8

17.5

84.2

After extracting the data, we needed to operate with values that do not have any missing values and so we used the completecases() function to create a subset of the original data. To view the data values for the exam and course work results and at the same time differentiate between boys and girls, this can be displayed by :-

julia> plot(df, x="Course", y="Written", color="Gender")

Notice that Gadfly produces the legend for the gender categorization automatically.

For an example of a function type invocation, the following shows what can be produced in a single call:

# Be aware of the new broadcasting style for functions in version 1.0 julia> gd.plot((x,y) -> x .\* exp.(-(x - floor.(x))).^2 .- y.^2, -8.0, 8, -2.0, 2.0)

Looking at a different type of invocation, let's plot two arrays of data but as line graphs rather than on a scatter diagram.

Gadfly produces multiline plots using the layer() routine and uses the concept of themes to overwrite the color schemes.

Here is a plot of 100 samples of a uniform variate (in red) together with a normal variate (in blue), both centered on zero and with unit variance:

julia> x = collect(1:100); julia> y1 = ones(100) - 2\*rand(100); julia> y2 = randn(100);

julia> gd.plot( gd.layer(x=x, y=y1, gd.Geom.line, gd.Theme(default\_color=gd.colorant"red")),

gd.layer(x=x, y=y2, gd.Geom.line, gd.Theme(default\_color=gd.colorant"blue")) )

Compose

Compose is a declarative vector graphics system which is part of the Gadfly system, but which can be used in its own right. Unlike most vector graphics libraries, Compose is thoroughly declarative. Graphics are defined using a tree structure, assembling various primitives, and then letting the module decide how to draw them.

The primitives can be classified as: context, form, and property, and the assembly operation is achieved via the compose() function:

context: an internal node form: a leaf node that defines some geometry, like a line or a polygon property: a leaf node that modifies how its parent's subtree is drawn, such as fill color, font family, or line width compose(a, b): returns a new tree rooted at a and with b attached as a child

A typical invocation has a distinctly LISP-like feel.

Below is code for building a complex drawing based on the Sierpinski fractal, which we met before as an example uisng Luxor. Here the overall shape of an equilateral triangle subdivided recursively into smaller equilateral triangles.

julia> using Compose julia> function sierpinski(n) if n == 0 compose(context(), polygon([(1,1), (0,1), (1/2, 0)])); else t = sierpinski(n - 1); compose( context(),(context( 1/4, 0, 1/2, 1/2), t), (context( 0, 1/2, 1/2, 1/2), t), (context( 1/2, 1/2, 1/2, 1/2), t)); end end

The triangle is composed using the polygon() function and built up recursively.

julia> cxt = compose(sierpinski(1), linewidth(0.2mm),fill(nothing), stroke("black")); julia> draw(SVG("sierp1.svg", 10cm, 8.66cm), cxt);

julia> cxt = compose(sierpinski(3), linewidth(0.2mm),fill(nothing), stroke("black")); julia> draw(SVG("sierp3.svg", 10cm, 8.66cm), cxt);

julia> cxt = compose(sierpinski(5), linewidth(0.2mm),fill(nothing), stroke("black")); julia> draw(SVG("sierp5.svg", 10cm, 8.66cm), cxt);

The figure below shows the result of three separate invocations for cases: n = 1,3,5:

Rsvg.jl

This module is adaptation of the librsvg. It is a wrapper package which provides a subset of the full API, but the one useful feature is that it can be used to transform SVG images into other formats.

The third image Sierpinski image (above) was saved as an SVG and can be changed to a PNG image using the following code:

#=

Open the SVG file and render to a Cairo Context (surface) Read svg data from a string and render to a Cairo Context are available. =# julia> fin = "images/sierp5.svg"; julia> fout = "images/sierp5.png";

julia> r = Rsvg.handle\_new\_from\_file(fin); julia> d = Rsvg.handle\_get\_dimensions(r); julia> cs = Cairo.CairoImageSurface(d.width,d.height,Cairo.FORMAT\_ARGB32); julia> c = Cairo.CairoContext(cs); julia> Rsvg.handle\_render\_cairo(c,r); julia> Cairo.write\_to\_png(cs,fout);

PGFPlots

PGFPlots is package that uses the LaTeX routines to produce visualisations. It integrates well with IJulia to output SVG images tothe notebook.

The user supplies axis labels, legend entries, and the plot coordinates for one or more plots. PGFPlots applies axis scaling, computes any logarithms, and axis ticks, and draws the plots.

The TEX library supports line, scatter, bar, area, histogram, mesh, and surface plots, but at the time of writing not all of these have been implemented in the Julia package.

As with all graphic engine type packages, certain additional executables need to be present in order for PGFPlots to work.

These are:

Pdf2svg: This is required by TikzPictures and installation varies by operations system.

Pgfplots: This is installed using a LaTeX package manager such as texlive or MiKTeX. It has a sourceforge webpage which is an excellent reference of what can be achieved and also some additional links

GNUPlot: This may be required in order to plot contours

The following code demonstrates drawing some of the curves met earlier

julia> using PGFPlots julia> p = Axis([

Plots.Linear(x -> sin.(3x) .\* exp.(-0.3x), (0,8), legendentry = L"$\sin(3x)\*exp(-0.3x)$"),

Plots.Linear(x -> sqrt.(x) ./ (1+x.^2), (0,8), legendentry = L"$\sqrt{2x}/(1+x^2)$") ])

# This requires installation of the pdf2svg utility save("linear-plots.svg", p);

It is very easy to make histograms with another type of style: Plots.Histogram

julia> fq = randn(10000); julia> p = Axis(Plots.Histogram(fq, bins=100), ymin=0) julia> save("histogram-plot.svg", p);

Note that Plots referred to here is a submodule of PGFPlots and should not be confused with the the Plots API which is the subject of the next section.

The Plots API

Plots.jl is a visualization interface and toolset. It was the brain-child of Tom Breloff and is maintained by a number of outstanding Julians from the JuliaPlots community group.

The Plots API sits above other backends, like GR or PyPlot, connecting commands with implementation. If one backend does not support the desired features or make the right trade-offs, it is possible to switch to another backend with one command.

There is no need to change the code and no need to learn a new syntax.

Another backend is Spencer Lyon's PlotlyJS ; this is essentially an "off-line" version of the older Plotly module and I will differ dealing with both of these later in of this chapter.

Some of the goals with the package are listed :

Powerful - Complex visualizations are easy to create. Intuitive - Commands "just work".

Concise - More efficient development and analysis. Flexible - Produce plots from your favourite package Consistent - No need to commit to one graphics package. Lightweight - Very few dependencies.

In addition to the github sources they provide an extensive set of online documentation with many examples of dynamic, interactive and 3-D visualisations. In this section I will just concentrate on some of the simpler main features of the API.

Simple plots

Part of the power of Plots lies is in the many combinations of allowed input data.

Writing plot(x = 1:10, y = rand(10)) will work as expected, as it will simply translate to a call of plot(1:10, rand(10)).

Instead it is possible to use plot(rand(10)) and the single input will be mapped to the :y keyword, and a missing value for :x will default a unit range1:10

Passing a (n × m) matrix of values will create m series, each with n data points; this follows a consistent rule - vectors apply to a series, matrices apply to many series as can be seen in the following example:

julia> using Plots

[ 23 ] Visualisation

Chapter 1

julia> gr() # Use GR as backend - don't load it as it will cause name clashes

# 25 data points in 3 series julia> xs = 0 : 2π/25 : 2π

# Define a sine, cosine and scaled production (by x) julia> data = [sin.(xs) cos.(xs) 0.5.\*xs.\*sin.(xs).\*cos.(xs)]

#= We put labels in a row vector: applies to each series A few bridges over the Thames in London =# julia> labels = ["Lambeth" "Westminster" "Blackfriars"]

# Marker shapes in a column vector: applies to data points julia> markershapes = [:diamond, :circle, :star5]

# Marker colors in a matrix: applies to series and data points julia> markercolors = [:orange :red :blue]

julia> p = plot(xs, data, label = labels,

shape = markershapes, color = markercolors, markersize = 5)

Notice the effect of specifying the labels, marker shapes and colours attributes :-

Layouts

Multiple plots together as subplots using layouts.

There are many methods for doing this, and the following highlights a simple method which is to define a layout which will split a series. The layout command takes in a 2-tuple which builds a grid of plot and will automatically split a series to be in each plot.

#= yy is a matrix of 3 plots shown on a grid of 3 plots the plots are labelled as y1,y2,y3 =# julia> yy = 2.0 \* randn(100,3) julia> plot(yy, layout = grid(3, 1, heights=[0.3,0.3,0.3]))

Recipes

For a full discussion of advanced features of layouts is given in the layout documentation.

Recipes are a way of defining visualizations in your own packages and code, without having to depend on Plots. They are a way of defining visualizations by utilising the @recipe macro from RecipesBase, the latter being a package which allows users to create advanced plotting logic without Plots.

Recipes have given rise to a number of frameworks; one such being StatsPlots, which will be discussed later and some others in the following section

There are four main types of recipes in Plots:

User Recipes

Type Recipes Plot Recipes Series Recipes

The recipe type is determined by the dispatch signature

The following is an example of a simple recipe:

julia> mutable struct MyRecipe end julia> @recipe function f(::MyRecipe, n::Integer = 10; add\_marker = false) linecolor --> :blue seriestype := :path markershape --> (add\_marker ? :circle : :none) delete!(plotattributes, :add\_marker) rand(n) end

MyRecipe is empty mutable structure which is used for the dispatch signature.

The aim of the recipe is to create a random path of points .

1. The signature f(args...; kw...) is converted by @recipe into a definition of apply\_recipe(plotattributes::KW, args...) where plotattributes is an attribute dictionary of type alias of KW Dict{Symbol,Any}.

2. The operator --> turns linecolor --> :blue into get!(plotattributes, :linecolor, :blue), setting the attribute only when it doesn't already exist.

3. The operator := turns seriestype := :path into plotattributes[:seriestype] = :path, forcing that attribute value.

4. markershape checks the add\_marker custom keyword but only if markershape was not already set

5. the macro then returns the data to be plotted by a call to rand()

# We need to instantiate the empty structure ... # ... and define each of 4 separate plots julia> mt = MyRecipe() julia> plot( plot(mt, 20, linecolor = :black), plot(mt, 100, linecolor = :red), plot(mt, marker = (:star5,5)), plot(mt, add\_marker = true) )

The effect of using the recipes is shown below:

StatsPlots

StatsPlots replacement for the Plots API, a little ;ess weighty but one that implements many statistical methods. It is capable of dealing directly with Data Frames and naturally encompasses the concepts of the Distributions packages (maintained by the JuliaStats group)

As such it is usually possible to load a dataset, apply a statistical procedure to the dataset(s) and display the results all within a single call.

In chapter 6, we looked at the dataset for GSCE results in a series of schools in the UK (part of the RDataset package) and the differences between marks for written (exam) vs coursework and also between male (boys) and female (girls) students

The following loads the dataset:

julia> using StatsPlots, RDatasets, Query julia> mlmf = dataset("mlmRev","Gcsemv"); julia> describe mlmd

variable Symbol 1 School 2 Student 3 Gender 4 Written 5 Course

mean Union…

min Any 20920 1 F

0.6

9.25

median Union…

max Any 84772 5521 M

90.0

100.0

nunique Union… 73 649 2

nmissing Union…

46.3652

73.3874

46.0

75.9

202 180

eltype DataType CategoricalString{UInt8} CategoricalString{UInt16} CategoricalString{UInt8} Float64 Float64

We can see that the dataset contains missing values both in the written and coursework marks and to apply statistical procedures these will need to be removed (as before).

Also we can use this step to differentiate on the basis of gender.

julia> wF = collect(skipmissing( mlmf[mlmf.Gender .== "F", :Written])); julia> wM = collect(skipmissing( mlmf[mlmf.Gender .== "M", :Written])); julia> cF = collect(skipmissing( mlmf[mlmf.Gender .== "F", :Course])); julia> cM = collect(skipmissing( mlmf[mlmf.Gender .== "M", :Course]));

Now by using the @df macro we can pass columns within an array and call the density function to display the spectral density of the 4 datasets.

Recall that the spectral density is the analogue of a frequency histogram for continuous data distributions,.

julia> labs = ["Exam (Girls)", "Exam (Boys)", "Course (Girls)", "Course (Boys)"]; julia> @df mlmf density([wF, wM, cF, cM], labels=labs, legend = :topleft)

It is clear that there is little difference to be seen on the basis of gender but a marked discrepancy when looking at examination and coursework marks.

The IRIS dataset

The Iris flowerdata is a multivariate dataset introduced by the British statistician and biologist Ronald Fisher and is extensively used as an example of discriminant analysis applied to multiple measurements in taxonomic problems.

The data set consists of 50 samples from each of three species of Iris (Setosa, Virginica and Versicolor). Four features were measured from each sample: the length and the width of the sepals and petals. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.

The dataset is available as part of the RDatasets package and loaded in the usual way:

julia> iris = RDatasets.dataset("datasets", "iris") julia> iris[1:6,:]

6 rows × 5 columns

SepalLength SepalWidth PetalLength PetalWidth Float64 Float64 Float64 Float64 1 5.1 3.5 1.4 0.2 2 4.9 3.0 1.4 0.2 3 4.7 3.2 1.3 0.2 4 4.6 3.1 1.5 0.2 5 5.0 3.6 1.4 0.2 6 5.4 3.9 1.7 0.4

We can visualise the data by using a 4x4 correlation plot

Species Categorical… setosa setosa setosa setosa setosa setosa

julia> @df iris corrplot([:SepalLength :SepalWidth

:PetalLength :PetalWidth], grid = false)

An alternative way to visualize structure in high-dimensional data is as a series of Andrews curves, which are well referenced, for the IRIS dataset, in the literature using Python, R, Matlab etc., and also here with Julia:

julia> @df iris andrewsplot(:Species, cols(1:4), legend = :topleft)

Backends

Although not all graphic packages obey the Plots API, but because of the liberal reorganisation of parameters the numbers is quite extensive.

These are termed as backends and the those used most commonly are:

GR

Plotly / PlotlyJS

PyPlot.

While working with Plots, different backends do provide slight variations in functionality and some parts of the API may not be available.

GR

Although a using statement for the backend should NOT be used for a backend, the package needs to be installed prior to its use

One of the best choices is GR, which is very quick and works on all platforms; its use on OSX requires installation of GKSTerm. Like all backends GR can be used as a standalone graphics package, it will not be discussed here and the reader is encouraged to look at the extensive online documentation to see just what can be achieved.

Plotly and PlotlyJS

These are treated as separate backends, though they share much of the code and use the Plotly javascript API. plotly() is the only dependency-free plotting option, as the required javascript is bundled with Plots. It can create inline plots in IJulia, or open standalone browser windows when run from the Julia REPL. However plotlyjs() the seen as preferred option, and taps into the greater functionality of Spencer Lyon's PlotlyJS.jl, e.g. inline IJulia plots can be updated from any cell..

From the Julia REPL (rather than a notebook), plotlyjs() uses Blink.jl and Electron to plot within a standalone GUI window; also it supports more output formats than Plotly, viz. such as EPS and PDF

We will be looking at Plotly in a little more detail in the new section.

PyPlot

The PyPlot packages is familiar to us by now, it has been used for many of the examples previously in this book and integrates well with Jupyter notebooks. It does also conform to the Plots API and is a good choice for a backend.

PyPlot has a great wealth of functionality, inherited from Python's matplotlib and this well supported by the API. we have seen that it can create 2-D and 3-D displays and will work with the REPL as well as Jupyter or Juno.

The downside is that Python needs to be installed (including Matplotlib) but if has been remarked that this is likely to be the case if using Jupyter and that a distribution such as Continuum's Anaconda will setup should handle all the necessary between Julia and

Python.

Others

There can be some setup problems with Python support by these are well discussed in the Julia documentation.

The three backends above are the most popular ones but there are a few other alternatives.

Two of these we have met before when describing some standalone packages, namely UnicodePlots and PGFPlots

Two more worth a mention here are InspectDR and HDF5

InspectDR

InspectDR is a relatively new, fast plotting tool with a responsive GUI designed to target efficient navigation of simulation results. It uses GTK+ to handle the graphics interface to the target display.

The InspectDR library is implemented using 3 distinct plot layers:

Image layer: Implemented with the Cairo library, the plot image layer allows the user to render (multi-) plots as simple images.

Widget layer: Library users can also integrate plots to their own GTK+ application by instantiating a single InspectDR widget.

Application layer: Most end users will likely display/interact with plots/data using the built-in Julia/GTK+ multi-plot application.

An example using InspectDR is not included here in the text but is available in the accompanying code for this chapter, both as a notebook and a REPL script.

HDF5

We saw in Chapter 6 that Hierarchical Data Format (HDF) is a set of file formats (HDF4, HDF5) designed to store and organise large amounts of data and that Julia has support for the latter which is also used as a special case for its internal JLD data format

Using HDF5 as a backend is then unusual since it does NOT create any graphics, rather it can be used to save, and later retrieve, a visualisation.

julia> using Plots; hdf5() Plots.HDF5Backend()

#= Create a plot 'p' using the simple plots example as above, this needs to be done AFTER the backend is specified.

=#

# This can be saved to disk in HDF5 format # The user is (currently) issued a warning julia> p = plot(. . .)

┌ Warning: HDF5 interface does not support `display()` function.

│ Use `Plots.hdf5plot\_write(::String)` method to write to .HDF5 "plot" file instead.

└ @ Plots ~/.julia/packages/Plots/UQI78/src/backends/hdf5.jl:193

julia> Plots.hdf5plot\_write(p, "plotsave.hdf5")

At a later stage/session it is possible to specify an additional conventional backend, retrieve the plot and display it.

julia> using Plots; julia> gr() Plots.GRBackend()

julia> p = Plots.hdf5plot\_read("plotsave.hdf5")

Display Frameworks

A graphics framework provides a high level interface to create complex visualisations as easily as possible. Ideally we would like to acquire the data, specify a minimum of parametrisation to identify the layout, labels etc. and pass these to a routine to produce the overall display with little (or no) knowledge of the underlying plotting methods.

We have already met one framework in the previous section to apply statistical procedures, namely that of StatsPlots. There are a number of others in Julia worth some attention and a couple of these will be discussed here.

Plotly

Plotly is a data analysis and graphing web application that is able to create precise and beautiful charts. It is based on D3 and as such incorporates a high degree of interaction such as hover text, panning, and zoom controls, as well as real-time data streaming options.

Originally, access to Plotly was via a REST API to the website http://plot.ly but a variety of programming languages now can access the API including Julia.

To use Plotly, you will need to sign up for an account via http://plot.ly providing a unique username and e-mail address. On registration, an API key will be generated and emailed together with a confirmation link. All plots are stored under this account and can be viewed and managed online as well as embedded in web pages.

So all coding require a call to the signin() routine:

julia> using Plotly julia> Plotly.signin("myuserid", "abc32def7g")

On successful execution, the signin routine returns a PlotlyAccount data object and an online graph is created under that account by formulating and executing a response function. The response function posts the data to Plot.ly that creates the plot and generates a URL for it as a reply.

The following is a script to display some log-log plots. The data is passed as an array of arrays (allowing for multiple curves) and a layout array is constructed to set the axis to logarithmic.

Additionally, we need to pass a name under which the plot is to be stored and indicate that, if the script is rerun the plot can be overwritten:

julia> trace1 = [ "x" => [0, 1, 2, 3, 4, 5, 6, 7, 8], "y" => [8, 7, 6, 5, 4, 3, 2, 1, 0], "type" => "scatter" ];

julia> trace2 = [ "x" => [0, 1, 2, 3, 4, 5, 6, 7, 8], "y" => [0, 1, 2, 3, 4, 5, 6, 7, 8], "type" => "scatter" ];

julia> data = [trace1, trace2];

julia> layout = [

"xaxis" => ["type" => "log", "autorange" => true], "yaxis" => ["type" => "log", "autorange" => true] ];

julia> response = Plotly.plot(data, ["layout" => layout, "filename" => "plotly-log-axes", "fileopt" => "overwrite"]);

julia> plot\_url = response["url"]

A value for plot\_url, such as http://plot.ly/~myuserid/17, indicates that it is stored with ID 17 under the myuser ID account.

When logging on the Plot.ly site the plot stored as plotly-log-axes.

The site contains a wide variety of code examples that can be downloaded as templates for your graphics; moreover, they are tailored with your specific username and password.

As a second example, here is a contour plot of some sinusoids with a randomly generated component:

julia> N = 100; julia> x = collect(range(-2\*pi, stop=2\*pi, length=N)); julia> y = collect(range(-2\*pi, stop=2\*pi, length=N)); julia> z = rand(N, N);

julia> for i = 1:N, j = 1:N r2 = (x[i]^2 + y[j]^2); z[i,j] = sin(x[i]) \* cos(y[j]) \* sin(r2)/log(r2+1); end

julia> data = [["z" => z,"x" => x,"y" => y, "type" => "contour"]]; julia> response = Plotly.plot(data,

["filename" => "simple-contour",

"fileopt" => "overwrite"]);

julia> plot\_url = response["url"]; julia> Plotly.openurl(plot\_url) # Display the plot via the URL

The figure below shows the results from both examples:

PlotlyJS

We noted that the Plotly online site uses Javascript libraries to create its displays from uploaded datasets. PlotlyJS does not interact with the Plotly web API, but rather uses the underlying javascript library to construct graphics using all local resources. This means that a Plotly account (nor an internet connection) to is NOT needed to use this package.

The routines, and their syntax, reflect their Plotly heritage and documentation for Plotly itself is useful. Also visualisations created by PlotlyJS are web-based, incorporating a degree of interactivity without any additional coding.

Since the underlying display for Jupiter notebooks is web-based then this presents not difficulty. When using PlotlyJS from the REPL the Blink.jl package, from Mike Innes' excellent JunoLabs is used. Blink acts as a Julia wrapper around Electron and can serve HTML content in a local window, thus enabling communication between Julia and the web page.

It was mentioned in the previous section that PlotlyJS conforms with the Plots API and so can be used as a backend. In circumstances where a degree of interactivity is required, this makes it an excellent choice.

As an example we will read the Apple stock prices from a CSV file and display the [Open,High,Low,Close] prices against date over a 100-day trading period at the end of 2013.

# Use DelimitedFiles to read the Apple data

julia> using PlotlyJS, DelimitedFiles, Dates julia> aaplcsv="/Users/malcolm/PacktPub/data/AAPL.csv";

#= Split off the header and reverse the dataset to get it in chronological order =# julia> (dd,dh) = readdlm(aaplcsv,','; header=true); julia> ddr = reverse(dd, dims=1);

The first column is the data as a String which needs to be converted to a Date type.

The next 4 columns are the opening, high, low and closing prices.

julia> t = Date.(ddr[end-99:end,1]); julia> yy = float.(ddr[end-99:end,2:5]);

To specify titles and axes labels etc., we need to setup a layout object and pass this to the plot() routine.

julia> lyo = Layout(;title="Apple Stock",

xaxis\_title="Date", yaxis\_title="Price");

julia> plot(t,yy,lyo)

The screenshot below shows the Blink screen and the effect of moving the mouse over the display.

Makie

Makie is a high level plotting interface for GLVisualize, with a focus on interactivity and speed. It uses the GPU to provide its displays but can operate via the CPU alone albeit somewhat slowly

It is relatively new (as of February 2019) and seen by its developers as a prototype for a redesign of Plots.jl, which will implement a very similar interface.

There are already created bewildering variety of examples which can be seen on the Makie website, http://makie.juliaplots.org , here we will just look at a couple here.

First the obligatory "Hello World" simple line plot.

# Define two target functions ...

julia> using Makie julia> f1(u) = sin.(u) ./ (1 .+ u) julia> f2(u) = u.\*exp.(-0.5u) .\* cos.(u)

# ... and create some values for a specific range = 4pi, length = 80) julia> y1 = f1(x); julia> y2 = f2(x);

julia> x = range(0, stop)

Makie uses the concept of a scene and a canvas from creating the visualisation. Lines are plotted between points using the lines() routine and individual points using scatter()

julia> scene = lines(x, y1, color = :blue) julia> scatter!(scene, x, y1, color = :red, markersize = 0.1)

julia> lines!(scene, x, y2, color = :black) julia> scatter!(scene, x, y2, color = :green, marker = :utriangle, markersize = 0.1)

julia> scene

Notice the conventional use of ! style routines which that an existing scene as first parameter and overlay onto it.

Secondly we are returning to the IRIS dataset from RDatasets, which we used this to create some correlation and Andrews plots via StatsPlots.

A scatter plot of the data is a well-known example from the earliest days of Gadfly and reproduced here. To recreate the display will require a little more coding to navigate through the dataframe.

julia> using RDatasets, DataFrames julia> iris = dataset("datasets", "iris")

We can create an empty scene using the Scene() constructor and buildup the scatter diagram.

To categorise the different species we will iterate over iris[:Species] and get corresponding the SepalWidth and SepalLength and then add the datapoint using scatter!()

julia> scene = Scene() julia> colors = [:red, :green, :blue] julia> i = 1 #color incrementer julia> for sp in unique(iris[:Species]) idx = iris[:Species] .== sp

sel = iris[idx, [:SepalWidth, :SepalLength]] scatter!(scene, sel[:,1], sel[:,2], color = colors[i], limits = FRect(1.5, 4.0, 3.0, 4.0)) global i = i + 1 end

# Add the axes and label then ...

julia> axis = scene[Axis] # get axis julia> axis[:names][:axisnames] = ("Sepal width", "Sepal length")

# ... then show the scene. julia> scene

Raster Graphics

Working with images and colormaps at the pixel level is often referred to as raster graphics. Since low-level packages eventually translate their plots to rasters, these packages are capable of working with images directly as well.

So in this final section, we'll turn our attention to a brief overview of some of the ways you

[ 43 ] Visualisation

can manipulate images in Julia

Cairo

Chapter 1

We met the Cairo package earlier as an enabler to high-level package, in particular when discussing the Luxor vector display module. Most of the types of displays we have been considering are defined in terms of points, curves, and shapes such as circles, rectangles, and polygons and Cairo is capable of generating these shapes quickly and efficiently.

In the following example, we will create the graphics context in Cairo from anRGBSurface method, create a rectangular region and fill the background with a light grey.

julia> using Cairo julia> c = CairoRGBSurface(256,256); julia> cr = CairoContext(c); julia> save(cr);

julia> set\_source\_rgb(cr,0.8,0.8,0.8); julia> rectangle(cr,0.0,0.0,256.0,256.0); # background julia> fill(cr); julia> restore(cr); julia> save(cr);

Now we create a circular region within this context, load an image of London's Tower Bridge and scale it to the context size.

julia> arc(cr, 128.0, 128.0, 76.8, 0, 2\*pi); julia> clip(cr); julia> new\_path(cr); julia> image = read\_from\_png ("images/towerbridge.png"); julia> w = image.width; h = image.height; julia> scale(cr, 256.0/w, 256.0/h);

Now paint the new clipped images and write it to disk.

Winston (Revisited)

Winston uses Cairo to create its displays and has some limited raster support via a couple of functions: colormap() to load RGB colormaps and imagesc() to display am image from a matrix of pixel values.

Winston defines one rainbow-style Jet colormap, but it is also capable of loading any maps defined in the Color package or indeed any maps created by the user, as can be seen from the following code snippet

julia> using Winston julia> wn = Winston

# Define a [0:4π, 0:4π] region ...

julia> x = collect(range(0., stop=4π, length=1000)); julia> y = collect(range(0., stop=4π, length=1000));

#= ... and a couple of 2-D functions and display them, after specifying the appropriate colour map =# julia> z1 = 100\*(x.^0.5) .\* exp.(-0.1y)'; julia> wn.colormap("jet", 10) julia> wn.imagesc(z1)

julia> z2 = sin.(x) .\* cos.(y)'; julia> wn.colormap("rdbu"); julia> wn.imagesc(z2)

Images package(s)

The set of packages from Tim Holy (and others) provides the most comprehensive support for manipulating images in Julia.

The main package is Images.jl; this previously used utility programs from the ImageMagick progam suite but now has been greatly expanded and implemented as a purely native set of modules.

In addition the TestImages.jl package, that functions similar to RDatasets does in the statistics world. It is a small set of common images that can be used in developing and testing Julia routines. The installation and build process of TestImages will retrieve this subset and store them locally. For other images the load routine fetch it from the online repository.

One of the interesting images is of the Earth taken from Apollo 17, this is included with the accompanying material to this book, but can be retrieved (and stored) as follows:

julia> using TestImages julia> img = testimage("earth\_apollo17"); [ Info: Could not find earth\_apollo17 in directory images Checking if it exists in the online repository.

[ Info: Found earth\_apollo17 in the online repository.

Downloading to the images directory.

% Total % Received % Xferd Average Speed Time Time Time Current Dload Upload Total Spent Left Speed 100 149 0 149 0 0 285 0 --:--:-- --:--:-- --:--:-- 285 100 160 100 160 0 0 140 0 0:00:01 0:00:01 --:--:-- 140 100 6358k 100 6358k 0 0 2942k 0 0:00:02 0:00:02 --:--:-- 11.5M

Notice that the type of image is not specified in the testimage() routine, these are normally either as TIFF or PNG , Once loaded, the internal representation in Julia is completely equivalent, regardless of its original format

The first snippet code reads the grayscale image of Lena Söderberg provided as an image to this chapter and then applies a convolution kernel to detect the edges of the image. A number of standard filters are available but we have specified a larger 5x5 kernel which produces a better resolution

Recall that in order to preserve the overall image intensity the components of the kernel must add up to zero.

julia> using Images julia> img = load("images/lena.png")

julia> kern = [

0 0 -1 0 0 0 -1 -4 -1 0 -1 -4 24 -4 -1 0 -1 -4 -1 0 0 0 -1 0 0 ];

# We need to centre the kernel (avoiding a warning) as it is 5x5

julia> imgg = imfilter(img, centered(kern))

The show() routine in the Image now defaults to displaying the image in the Jupyter notebook without the necessity now to use the ImageView package

Resizing and transformations

The next example gives an example which loads the lighthouse image from the TestImages package. This is a large colour image which a 1.5 aspect ratio and can be viewed in the accompanying notebook.

julia> using TestImages julia> img = testimage("lighthouse");

We wish to detect the centre of the image, rotate it through 180 o and convert it to greyscale. For this we need to add (and use) a two additional modules:

julia> using CoordinateTransformations, OffsetArrays julia> tfm = recenter(RotMatrix(pi), Images.center(img)) AffineMap([-1.0 -1.22465e-16; 1.22465e-16 -1.0], [513.0, 769.0])

julia> imgw = Gray.(warp(img, tfm)); julia> imresize(imgw, 256, 384)

The resultant image is shown below :-

Summary

This chapter has presented the wide variety of options for producing visualization which are now available to the Julia programmer. We looked at some of the elderly popular packages, such as Winston, Gadfly, and PyPlot, and PGFPlots.

Following on these the Plots API was introduced together with some of the newer backends such as GR and PlotlyJS. Also, we looked at how the Plotly system can be utilized in Julia to generate, manipulate, and store data visualizations online.

The API makes the provision of graphic frameworks such as StatsPlots, GraphicPlots and Makie possible and these where discussed briefly.

Finally, we looked at the means by which raster graphics and imagery can be processed and displayed.

In the next chapter, we will return to the subject of accessing data by looking at the various ways with which we can interact with SQL and NoSQL databases in Julia, together with the JuliaB analytical engine.