Part 2 Section 1: Numerical Analysis – Basic Tools

Danail Obreschkow

2025-08-04

Prerequisites

Load libraries for this chapter:

```
library(magicaxis) # routines for nice plots
library(Rmpfr) # arbitrarily accurate computation
library(pracma) # excellent package with 220 numerical analysis functions
library(akima) # interpolation functions
library(cooltools) # additional interpolation functions and colour palette
library(MASS) # 2D density estimation
library(mvtnorm) # multivariate normal distribution
library(ellipse) # compute covariance ellipses
```

Foreword

Numerical analysis is the study of algorithms to find numerical approximations of mathematical equations, including integrals and differential equations. In *computational statistics*, a solid understanding of such numerical methods is often necessary. In turn, several powerful numerical algorithms rely, at their heart, on statistical approaches.

The purpose of this chapter is to provide an overview of basic numerical tools that are frequently needed in scientific research. The chapter does not present an exhaustive overview, nor does it dwell in the realm of the algorithms behind the tools. It is nonetheless worth to appreciate that most algorithms used in modern numerical analysis are highly sophisticated and interesting from a mathematical point of view.

This chapter will make strong use of the package **pracma** (short for "Practical Numerical Math Functions"). This package includes 220 functions that emulate equally named Matlab functions, making this package particularly useful for those familiar with Matlab. If you ever need to evaluate a function that you might not have heard before, it is worth first checking the **pracma** package and the **gsl** package (short for "GNU Scientific Library").

Generation of random numbers

In statistical computations and throughout this course we often use computer-generated 'random' numbers. We shall therefore briefly elaborate on what we mean by such random numbers and how they are actually generated. This section will also introduce a few advanced routines for drawing random numbers. Other examples will be provided later in the course in the context of probability density functions (part 3).

Computers can generate 'random' numbers using algorithms known as random number generators (RNGs). In the case of classical computers such routines necessarily rely on strictly deterministic operations, thus the 'random' numbers cannot possibly be truly random. However, since the early days of scientific computation, very sophisticated algorithms have been developed that can generate *sequences* of numbers which appear, by most statistical measures, indistinguishable from truly random. Numbers generated by such algorithms are called pseudo-random and their generators are sometimes called pseudo-RNGs (PRNGs).

Most PRNGs start from a sequence of integers that nearly uniformly samples a finite interval (normally the interval of all 32-bit integers). In **R**, the default PRNG is a so-called modified Mersenne-Twister, which alleviates some of the failures of the original Mersenne-Twister (dating back to 1998). The randomness properties of this PRNG are remarkably good. In fact, it is extremely rare to encounter a practical situation, where the PRNGs reveal a significantly different behaviour from the expectation of true randomness.

The sequence of pseudo-random numbers produced by a PRNG normally needs to be initialised in some way. This is typically done by setting a so-called *seed*, which is most commonly a positive integer. This seed can be set manually or automatically, e.g. using the internal clock reading of the computer. The former is important if a code using pseudo-random numbers should use the *same* random numbers every time it is executed. In **R**, such a custom seed can be set using the routine **set.seed**, e.g.:

```
set.seed(1)
```

Given a base PRNG for integers, it is then possible to apply mathematical transformations to produce random numbers from different distributions. As seen earlier, **R** provides the built-in function **runif** to draw floating-point real random numbers with uniform probability between 0 and 1 or any other interval specified in the arguments. E.g.

```
runif(3,min=-1,max=1)
```

```
## [1] -0.4689827 -0.2557522  0.1457067
```

When running the same code again, we get different numbers:

```
runif(3,min=-1,max=1)
```

```
## [1]  0.8164156 -0.5966361  0.7967794
```

However, if we reset the seed to the same initial value, we can regenerate the same sequence of pseudo-random numbers again:

```
set.seed(1)
runif(6,min=-1,max=1)
```

```
## [1] -0.4689827 -0.2557522  0.1457067  0.8164156 -0.5966361  0.7967794
```

This demonstrates that pseudo-random numbers follow a *deterministic* sequence.

Similarly, the function **rnorm** draws pseudo-random numbers from a normal distribution of mean 0 and standard deviation 1 or any other normal distribution specified in the arguments. For instance,

```
x = rnorm(1e5,mean=1) # draw 1e5 random number of expectation 1
print(mean(x))
```

```
## [1] 0.9977758
```

The mean is very close to unity, i.e. to the expectation of the population distribution.

Despite their similarity to true random numbers, pseudo-random numbers are deterministic. The only way to generate (nearly) true random numbers is to replace the deterministic computations of a CPU by

some physical source of randomness. These sources most frequently rely on classical thermal noise or on a measurement of a superposed quantum state, which leads to a non-deterministic state reduction (sometimes called the 'collapse of the wave function'). RNGs relying on such processes are known as true random number generators (TRNGs) or hardware random number generators (HRNGs). Nowadays, TRNGs can be purchased off-the-shelf as plug-and-play devices. Alternatively, there are several online services that generate true random numbers in real-time on demand.

Yet another type of random numbers are so-called quasi-random numbers, which we will discuss later in the context of QMC integrators. These are not be be confused with pseudo-random numbers. The main difference is that pseudo-random numbers are meant to be as close to truly random as possible, whereas the quasi-random numbers are deliberately less 'clustered'. In most cases of scientific computations, 'random' refers to 'pseudo-random', as opposed to truly random or quasi-random. It is therefore quite common to use RNG as synonym for PRNG.

Drawing random numbers from arbitrary distributions

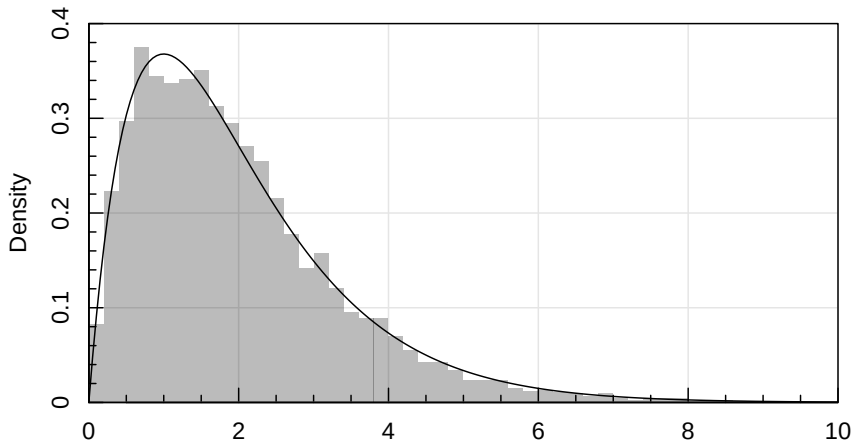
The **cooltools** package provides some useful routines to generate pseudo-random numbers. In particular, the routine **rng** of this package samples an arbitrary function f of one or several variables. Let us discuss some examples in the following. Other useful RNG variants will be discussed later in this course along with probability density functions (part 3).

Example in 1D. We wish to draw 10^4 random numbers $x \in [0, \infty)$ from the distribution function $f(x) = xe^{-x}$ (solid line in the plot below). The code to do this is:

```
f = function(x) exp(-x)*x
x = rng(f,1e4,0,10)$x # limit x<=10, because larger values are vanishingly rare
```

Let us plot the function $f(x)$ and the histogram of pseudo-random numbers:

```
maghist(x,50,freq=F,col='#00000044',border=NA,ylim=c(0,0.4),
        xaxs='i',yaxs='i',ylab='Density',verbose=F)
curve(f,0,10,200,add=T)
```



Note that this function happens to be normalised, $\int_0^\infty f(x)dx = 1$, but **rng** can also handle non-normalised functions.

Example in 2D. We would like to draw a random set of 2D vectors, which uniformly sample the dark squares of a chess board. To do so it suffices to note that the function

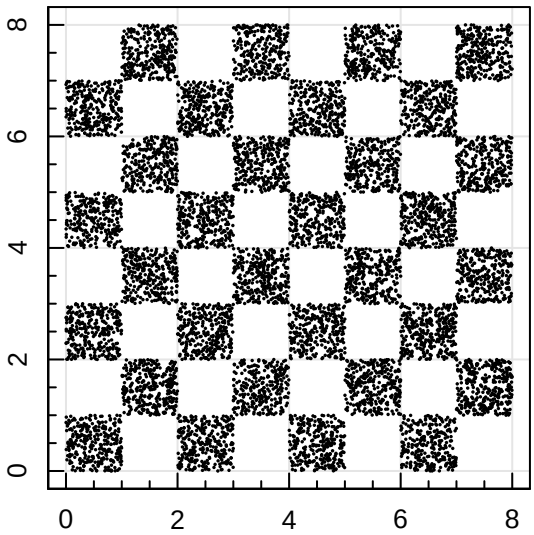
```
f = function(x) as.numeric(xor(x[1]%%2>1,x[2]%%2<1))
```

takes the value 0 if the point `(x[1],x[2])` lies inside a white (unit) square and 1 if it lies in a black square. We can then feed this function into **rng**:

```
x = rng(f,1e4,c(0,0),c(8,8))$x
```

```
## WARNING: use rng with vectorized function for faster performance.
```

```
par(pty='s',cex=0.8)
magplot(x,pch=16,cex=0.3)
```



Note that the above call of **rng** produced a warning that recommends vectorizing the function `f` for faster performance. In other words, we should write this function in such a way that it can take an N-by-2 matrix as input and return and N-element vector. In the present example, this is achieved by simply inserting commas inside the square brackets:

```
fvect = function(x) as.numeric(xor(x[,1]%%2>1,x[,2]%%2<1))
```

This small modification accelerates the RNG by more than an order of magnitude. Let's see:

```
system.time(rng(f,1e5,c(0,0),c(8,8),warn=FALSE))
```

```
##    user  system elapsed
##   0.252   0.012   0.263
```

```
system.time(rng(fvect,1e5,c(0,0),c(8,8)))
```

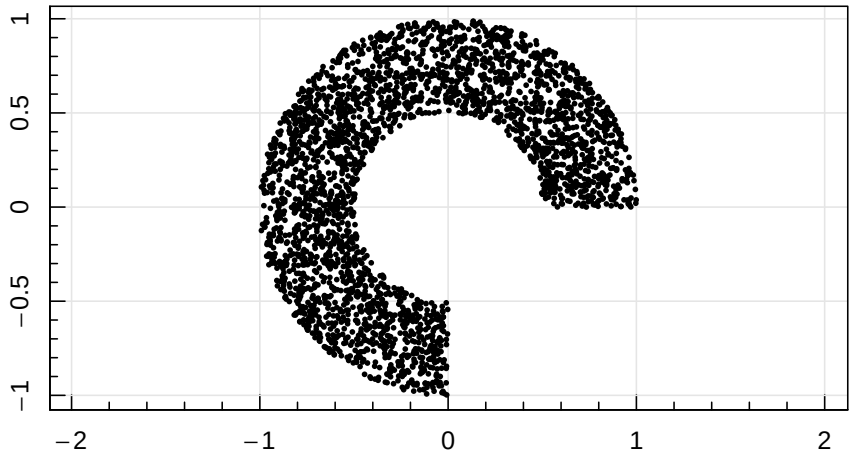
```
##    user  system elapsed
##   0.013   0.002   0.014
```

Uniformly sampling 2D and 3D vectors

The particular case of sampling 2D or 3D vectors in an isotropic manner is quite common in science. Importantly, such samplings cannot be achieved by uniformly sampling each coordinate individually – at least not without truncating the random sample. Instead care must be applied to sampling the azimuth and polar angles correctly. The functions **runif2** and **runif3** of the **cooltools** package take care of this.

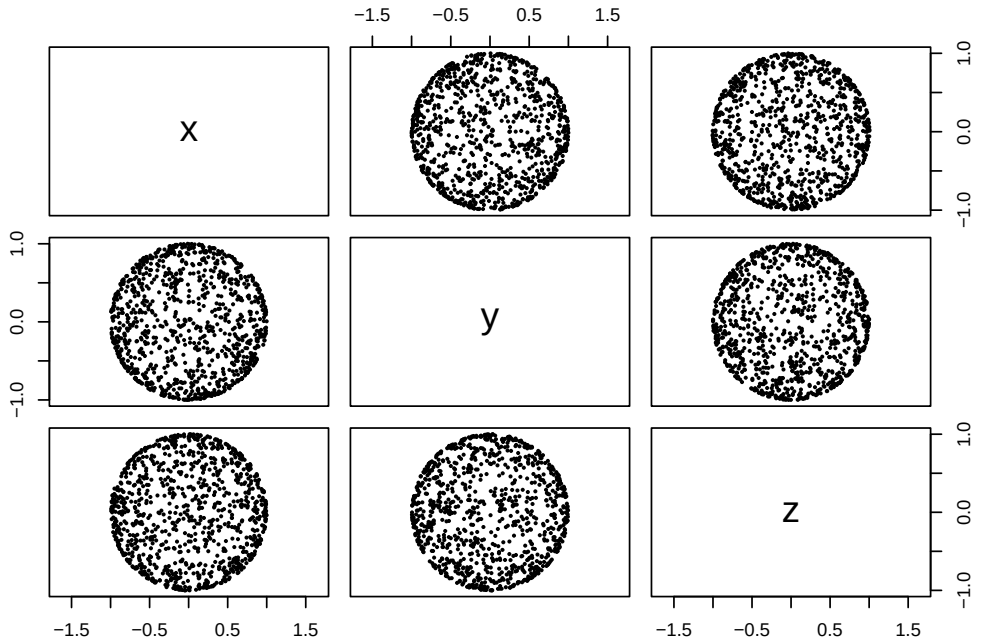
Uniform random sampling of a circle/disk/disk segment. This can be done using the **runif2** function in the **cooltools** package:

```
x = runif2(3e3,r=c(0.5,1),azimuth=c(0,3/2*pi))
magplot(x,pch=16,cex=0.5,asp=1,xlab='',ylab='')
```



Uniform random sampling of a sphere or sphere segment. This can be done using the **runif3** function in the **cooltools** package. The following example produces 10^3 isotropic unit vectors, stored in a 1000-by-3 matrix `x`. In other words, we are sampling the *surface* of the unit-sphere. We also plot the projections of the random points onto the three orthogonal planes.

```
x = runif3(1e3,r=1) # produces 1e3 unit vectors
colnames(x) = c('x','y','z')
magplot(x,pch=16,cex=0.5,asp=1,xlab='',ylab='') # show all projections
```



Non-elementary functions

By convention, *elementary functions* are functions of one variable made of arithmetic operations ($+$, $-$, $\times$, $\div$) and the basic mathematical functions normally encountered in high-school: powers, exponentials, trigonometric functions, hyperbolic functions, as well as the inverses and various (finite) combinations of these functions. There is nothing particularly unique about this set of functions other than that they are quite frequently used. Many other functions are similarly differentiable, e.g. the gamma functions, error functions, etc.

This section discusses some useful non-elementary functions available in **R**, which are often used in science. The list is by no means exhaustive.

Statistical functions

Statistical computations often require functions from the "gamma-function family", which are available as base functions in **R**:

- The gamma function, $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$, is called as **gamma(x)**. This function is commonly used in various fields of physics and statistics. For instance, it is used to model the time separation between Earth quakes or to integrate the so-called Schechter function in astrophysics. It is an analytical extension of the factorial, since $n! = \Gamma(n+1)$ for $n \in \mathbb{N}_0$.
- The beta function, $B(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}dt = \Gamma(x)\Gamma(y)/\Gamma(x+y)$, is called as **beta(x,y)**. This function is not to be confused with the "beta-distribution", a one-dimensional probability distribution defined on the unit interval $x \in [0,1]$ as $\beta_{a,b}(x) = x^{a-1}(1-x)^{b-1}/B(a,b)$.
- The factorial function, $n! = 1 \times 2 \times ... \times n$, is called as **factorial(n)**, where `n` is a non-negative integer. (In fact, **R** allows for non-integer arguments, in which case the value of $\Gamma(n+1)$ is returned.) Factorials are common place in statistical computations, mainly because there are exactly $n!$ ways to rearrange (permute) n distinct objects.
- The binomial coefficient, $\binom{n}{k} = n!/(k!(n-k)!)$, is called as **choose(n,k)**, where `n` and `k` are non-negative integers (though **R** allows for real arguments as in the factorial function). As the code-name suggests, there are exactly $\binom{n}{k}$ ways to draw an (unordered) subset of $k \leq n$ objects from a master set of n objects.

Example: Given 5 red roses, 4 yellow ones and 3 white ones, how many distinct ways are there to arrange them in an ordered sequence? To answer this question, we first compute the total number of permutations of all the $5+4+3=12$ roses, ignoring that many permutations will look the same. This number is 12!. However, permuting the red roses amongst each other does not change the sequence, thus we should divide 12! by the number of red permutations (5!). The same argument applies to the yellow and white roses. Thus, the answer is that there are $12!/(5! \cdot 4! \cdot 3!) = 27720$ distinct ways.

Natural logarithms of statistical functions

In statistical applications it is often necessary to evaluate the logarithms of the statistical functions above for large arguments. For instance, we might need to evaluate $\ln\Gamma(1000)$. However, the numerical value of $\Gamma(1000)$ is so large that it lies well beyond the standard range of 64-bit floating-point numbers; hence

```
try(log(gamma(1000)))
```

```
## [1] Inf
```

fails, although this logarithm could easily be represented as floating-point value. To bypass this issue **R** offers the functions **lgamma(x)**, **lbeta(a,b)**, **lfactorial(x)** and **lchoose(n,k)**, which return the *natural* logarithms of the corresponding functions. For example, to evaluate $\ln\Gamma(1000)$, use

```
lgamma(1000)
```

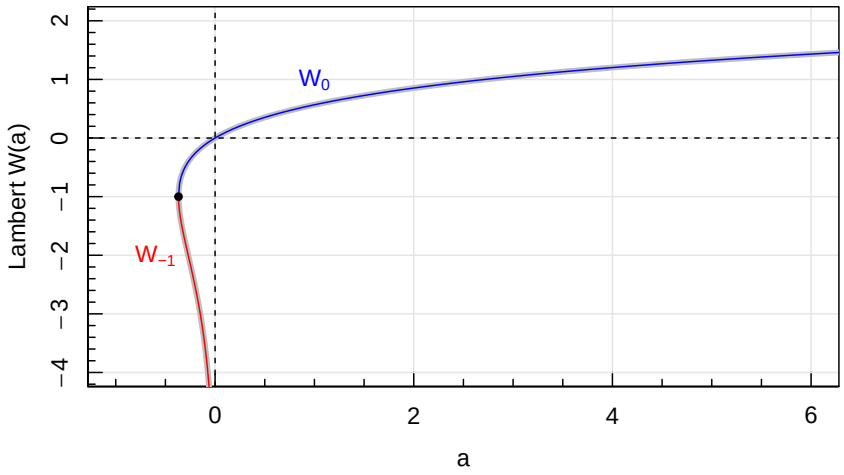
```
## [1] 5905.22
```

Lambert W functions

Lambert W functions are the solutions x of the equation $a = xe^x$. If restricted to real values x, this function only admits a solution if $a \geq -1/e$. For $a < 0$, it has two branches, the upper branch $W_0 \geq -1$ (also called W_p) and the lower branch $W_{-1} \leq -1$ (also called W_m). The two branches join at $a = -1/e$, where $W_0(a) = W_{-1}(a) = -1$. For $a \geq 0$, only the W_0 branch is exists. In **R**, the Lambert W functions can be computed via **lambertWp** and **lambertWn** from the **pracma** package. If many ($>10^5$) evaluations of W are required, much ($\times 100$) faster, numerically equivalent implementations can be found in the **lamW** package.

Let us visualize the two branches of the Lambert W function:

```
W = seq(-4.5,2,by=0.01)
a = W*exp(W)
magplot(a,W,type='l',xlim=c(-1,6),ylim=c(-4,2),lwd=4,col='grey',
        xlab='a',ylab='Lambert W(a)')
curve(lambertWp,-1/exp(1),10,n=1e3,col='blue',add=T)
curve(lambertWn,-1/exp(1),0,n=1e3,col='red',add=T)
points(-1/exp(1),-1,pch=20)
abline(h=0,v=0,lty=2)
text(1,1,expression('W' ['0']),col='blue')
text(-0.6,-2,expression('W' ['-1']),col='red')
```



In this figure, the direct evaluation of the Lambert W function $W(a)$ is shown as thin lines (blue/red for upper/lower branch). As a verification, we also plotted the inverted function $a(W) = We^W$ as thick grey line with swapped coordinates.

The Lambert W function has many applications in physics and statistics. Specifically in astronomy, Lambert W functions appear, for instance, in the context of exponential disks (e.g. Obreschkow et al., ApJ, 2016) and when sampling particles from so-called NFW density profiles of dark matter haloes (e.g. Robotham & Howlett, RNASS, 2018).

Example: half-mass radius of an exponential disk

One of the simplest models of galactic disks is a thin, axially symmetric disk of exponentially declining surface density,

$$\Sigma(r) = \frac{M}{2\pi R^2} e^{-r/R}, \tag{1}$$

where R is the scale radius and M is the total mass ($M = \int_0^\infty \Sigma(r) 2\pi r dr$). This disk model is also known as a *Freeman disk* in honour of the Australian astrophysicist Ken Freeman.

We would now like to compute the half-mass radius of this disk. To do so it is convenient to introduce the normalised radius $x = r/R$. The mass fraction $f(x) = M(< xR)/M$ contained inside x is

$$f(x) = M^{-1} \int_0^{r=xR} \Sigma(r) 2\pi r dr = \int_0^{r=xR} e^{-r/R} R^{-2} r dr = \int_0^x e^{-y} y \, dy = 1 - (x+1)e^{-x}. \tag{2}$$

The normalised half-mass radius is then given as the solution of $f(x) = 1/2$. Substituting $-s = x + 1$,

$$1 + se^{s+1} = 1/2 \leftrightarrow se^{s+1} = -1/2 \leftrightarrow se^s = -1/(2e) \leftrightarrow s = W(-1/(2e)). \tag{3}$$

Since $x \geq 0$, only the negative branch of W is physically valid and

$$x = -W_{-1}(-1/(2e)) - 1. \tag{4}$$

Numerically:

```
x = -lambertWn(-exp(-1)/2)-1
cat(sprintf('The half mass radius of an exponential disk is %.5f*R.\n',x))
```

```
## The half mass radius of an exponential disk is 1.67835*R.
```

Bessel functions

The Bessel functions are a family of standard solutions to the differential equation

$$x^2 \frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \alpha^2)y = 0, \tag{5}$$

where $\alpha \in \mathbb{C}$ is a real or complex parameter. In physics, Bessel functions are quite common. For instance, they are the radial part of the vibrational modes of a circular drum.

There are two classes of solutions to eq. (5):

- the *Bessel functions of the first kind* $J_\alpha(x)$, which are finite at the origin ($x = 0$) for $\alpha \in \mathbb{R}_+$ and negative integer α;
- the *Bessel functions of the second kind* $Y_\alpha(x)$, which generally diverge at $x = 0$.

The Bessel functions are valid even for complex arguments x. An important special case is that of a purely imaginary argument ix with $x \in \mathbb{R}$. In this case, the standard solutions of eq. (5) are known as the

- the *modified Bessel functions of the first kind* $I_\alpha(x)$, which are finite at $x = 0$ for positive α;
- the *modified Bessel functions of the second kind* $K_\alpha(x)$, which generally diverge at $x = 0$.

It is possible to define the function $J_\alpha(x)$ by its Taylor series around $x = 0$, given by

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\,\Gamma(m + \alpha + 1)} \left(\frac{x}{2}\right)^{2m+\alpha} \tag{6}$$

The other three Bessel functions can then be expressed as elementary functions of $J_\alpha(x)$.

The basic **R** routines include the Bessel functions are called **besselJ(x,nu)**, **besselY(x,nu)**, **besselI(x,nu)**, and **besselK(x,nu)**, where `nu` is our α parameter.

Elliptic integrals

The term *elliptic integrals* originally arose in connection with the problem of computing the arc length of an ellipse. In modern mathematics, this term has been broadened to a wider class of integrals, of which the best known ones are the so-called elliptic integrals of the first (K) and second (E) kind.

By definition, the *incomplete elliptic integral of the first kind* is the function

$$K(\phi, k) = \int_0^\phi \frac{d\theta}{\sqrt{1 - k^2 \sin^2\theta}}. \tag{7}$$

The associated *complete elliptic integral of the first kind* is obtained by setting $\phi = \pi/2$, i.e.

$$K(k) = K(\pi/2, k) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - k^2 \sin^2 \theta}}. \tag{8}$$

Similarly, the *incomplete elliptic integral of the second kind* is defined as

$$E(\phi, k) = \int_0^{\phi} \sqrt{1 - k^2 \sin^2 \theta}\, d\theta, \tag{9}$$

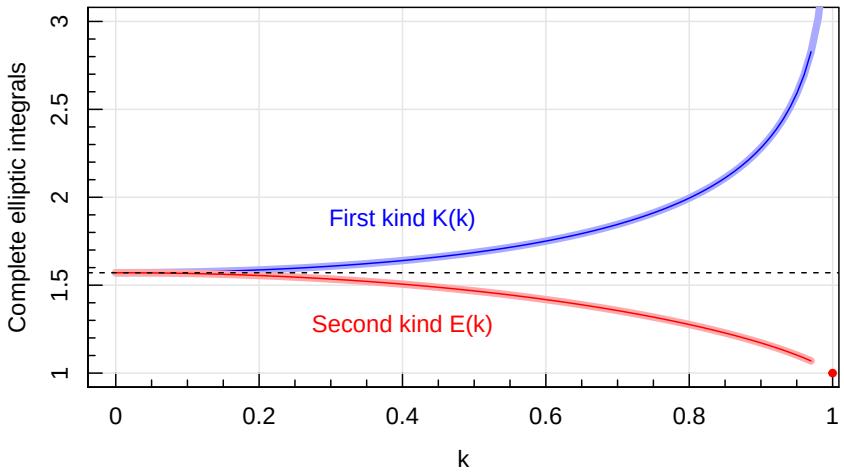
and the associated *complete elliptic integral of the first kind* is again obtained by setting $\phi = \pi/2$,

$$E(k) = E(\pi/2, k) = \int_0^{\phi} \sqrt{1 - k^2 \sin^2 \theta}\, d\theta. \tag{10}$$

All elliptic integrals are only defined for $k \in [0, 1]$. The extrema of the complete elliptic integrals occur at the domain boundaries and are given by $K(0) = E(0) = \pi/2$, $K(1) = \infty$, $E(1) = 1$.

Elliptic integrals can easily be evaluated in **R** using numerical integration. However, for the special cases of the *complete* elliptic integrals $K(k)$ and $E(k)$, the **pracma** package offers the function **ellipke(m)**, which computes both of these functions simultaneously. Note that the argument m of this function corresponds to k^2 in the definitions above. Let us plot $K(k)$ and $E(k)$, both using the functions **ellipke** (thick lines) and the solutions obtained by direct numerical integration (thin lines).

```
K.pracma = function(k) ellipke(k^2)$k
E.pracma = function(k) ellipke(k^2)$e
K.integral = Vectorize(function(k) integral(function(t) 1/sqrt(1-k^2*sin(t)^2),0,pi/2))
E.integral = Vectorize(function(k) integral(function(t) sqrt(1-k^2*sin(t)^2),0,pi/2))
magcurve(K.pracma,0,0.999999,lwd=5,col='#aaaaff',ylim=c(1,3),
         xlab='k',ylab='Complete elliptic integrals')
curve(K.integral,col='blue',add=T)
curve(E.pracma,lwd=5,col='#ffaaaa',add=T)
curve(E.integral,col='red',add=T)
points(1,1,pch=20,col='red')
abline(h=pi/2,lty=2)
text(0.4,1.87,'First kind K(k)',col='blue')
text(0.4,1.27,'Second kind E(k)',col='red')
```



Example: rotation curve of a self-gravitating exponential disk

The rotation curve of a self-gravitating exponential disk is an important astrophysical example, where both Bessel functions and elliptic integrals matter.

Gravitational potential

The gravitational field is a vector field that specifies the acceleration $\mathbf{a}(\mathbf{x})$ of a test mass at any point $\mathbf{x} \in \mathbb{R}^3$ in space. Following Newton's law of gravity, this field is conservative ($\nabla \times \mathbf{a} = 0$) and can hence be written as the (negative) gradient of a scalar field Φ, known as the *gravitational potential*,

$$\mathbf{a}(\mathbf{x}) = -\nabla\Phi(\mathbf{x}).$$

The potential produced by a point mass M located at a position $\mathbf{y}$ is (up to an arbitray constant)

$$\Phi(\mathbf{x}) = -\frac{GM}{|\mathbf{x}-\mathbf{y}|}.$$

The potential of any other mass distribution can then be derived using the principle of superposition,

$$\Phi(\mathbf{x}) = -\int_{\mathbb{R}^3} \frac{G\,dm(\mathbf{y})}{|\mathbf{x}-\mathbf{y}|}.$$

Gravitational potential of a circle

Before considering a flat galactic disk, let us first compute the gravitational potential of a circular ring of mass M and radius s. We like to evaluate the potential at any distance r from the ring centre, in the plane of the ring.

$$\Phi(r) = -\int_{\text{ring}} \frac{G\,dm}{|\mathbf{r}-\mathbf{s}|} = -\frac{GM}{2\pi}\int_0^{2\pi} \frac{d\phi}{\sqrt{r^2+s^2-2rs\cos\phi}} = -\frac{GM}{\pi}\int_0^{\pi} \frac{d\phi}{\sqrt{r^2+s^2-2rs\cos\phi}}$$

We can use the trigonometric identity $\cos(\phi) = 2\cos^2(\phi/2) - 1$ and substitute $\theta := \phi/2$:

$$\Phi(r) = -\frac{2GM}{\pi}\int_0^{\pi/2} \frac{d\theta}{\sqrt{r^2+s^2-2rs(2\cos^2\theta-1)}} = -\frac{2GM}{\pi}\int_0^{\pi/2} \frac{d\theta}{\sqrt{(r+s)^2-4rs\cos^2\theta}}$$

Factoring out $(r+s)$ and realising that $\cos^2$ can be replaced by $\sin^2$ due to the symmetry these functions on the interval $[0, \pi/2]$, we obtain

$$\Phi(r) = -\frac{2GM}{\pi(r+s)}\int_0^{\pi/2} \frac{d\theta}{\sqrt{1-\frac{4rs}{(r+s)^2}\sin^2\theta}} = -\frac{2GM}{\pi(r+s)}K\left(\frac{2\sqrt{rs}}{r+s}\right).$$

An elliptic integral has appeared!

Gravitational potential of an exponential disk

Let us now use the potential of a ring to build up the potential of a flat disk with exponential surface density,

$$\Sigma(r) = \frac{M}{2\pi R^2}e^{-r/R}.$$

This disk is made of rings of mass $dm = 2\pi r\Sigma(r)dr$. Hence the disk potential, at a distance r from the disk centre, in the plane of the disk, becomes

$$\Phi(r) = -\frac{2GM}{\pi R^2}\int_0^{\infty} \frac{s}{r+s}e^{-s/R}K\left(\frac{2\sqrt{rs}}{r+s}\right)ds.$$

Substituting for non-dimensional lengths $x := r/R$ and $y := s/R$,

$$\Phi(x) = \frac{GM}{R}f(x), \quad \text{where} \quad f(x) = -\frac{2}{\pi}\int_0^{\infty} \frac{ye^{-y}}{x+y}K\left(\frac{2\sqrt{xy}}{x+y}\right)dy$$

Rotation curve of a self-gravitating exponential disk

In order to determine the circular velocity induced by the disk potential, it suffices to remember that in circular orbits the radial component of the acceleration is $a = -r\omega^2 = -v^2/r$ (centripetal acceleration), where $a = -d\Phi/dr$. Hence,

$$v(x) = \sqrt{r\frac{d\Phi}{dr}} = \sqrt{x\frac{d\Phi}{dx}} = \sqrt{\frac{GM}{R}}g(x) \quad \text{where} \quad g(x) = \sqrt{xf'(x)}.$$

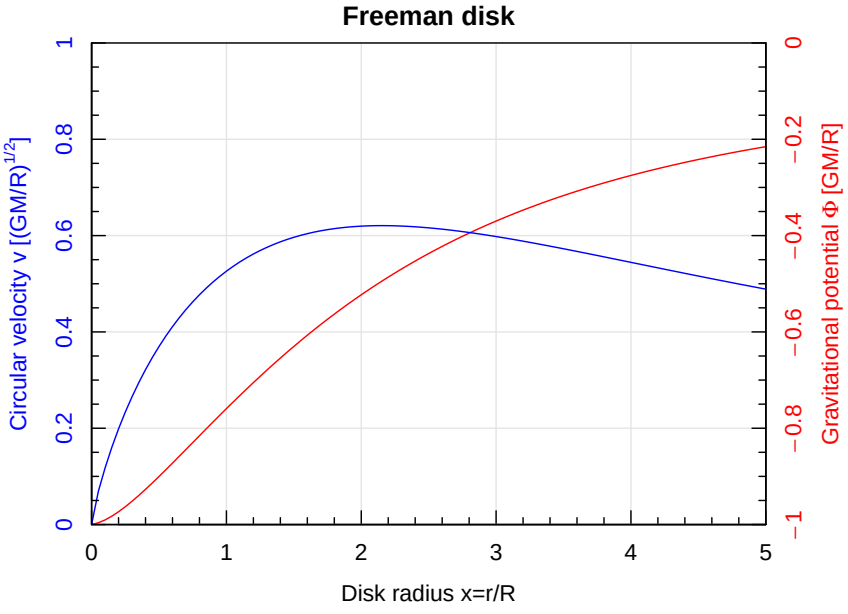
Numerical solution

In order to evaluate the non-dimensional potential $f(x)$, it is important to realise that this function involves integrals, whose integrand diverges if $y = x$, since $K(1) = \infty$. The integral is nonetheless converged. To avoid numerical difficulties at this singularity, we can add a small numerical constant ε to the denominator $x + y$ in the elliptic integral. This makes sure that the denominator $x + y$ never vanishes and that $K(k)$ is only evaluated for $k < 1$.

When computing the non-dimensional circular velocity $g(x)$, we use a numerical differentiation of $f(x)$. At the point $x = 0$, this will include some evaluations of $f(x)$ at slightly negative values x, where $f(x)$ is ill-defined. To avoid numerical issues, we therefore use the absolute values of x in computing $f(x)$.

```
# non-dimensional gravitational potential
K = function(k) ellipke(k^2)$k
f = Vectorize(function(x) {
  x = abs(x)
  integrand = function(y) -2/pi*y*exp(-y)/(x+y)*K(2*sqrt(x*y)/(x+y+1e-5))
  return(integral(integrand,0,100))
})
par(mar=c(3,4,2,4))
magcurve(f,0,5,col='red',xlim=c(0,5),ylim=c(-1,0),xaxs='i',yaxs='i',yaxt='n',
         xlab=expression('Disk radius x=r/R'),ylab='',main='Freeman disk')
par(col.axis='red',fg='red')
magaxis(4)
mtext(expression('Gravitational potential'~Phi~'[GM/R]'),4,2)

# non-dimensional circular velocity
g = function(x) sqrt(x*fderiv(f,x))
par(usr=c(par()$usr[1:2],0,1),col.axis='blue',fg='blue')
curve(g,0,5,col='blue',add=T)
magaxis(2,ylab=expression('Circular velocity v [(GM/R)'^'1/2'*']'))
```



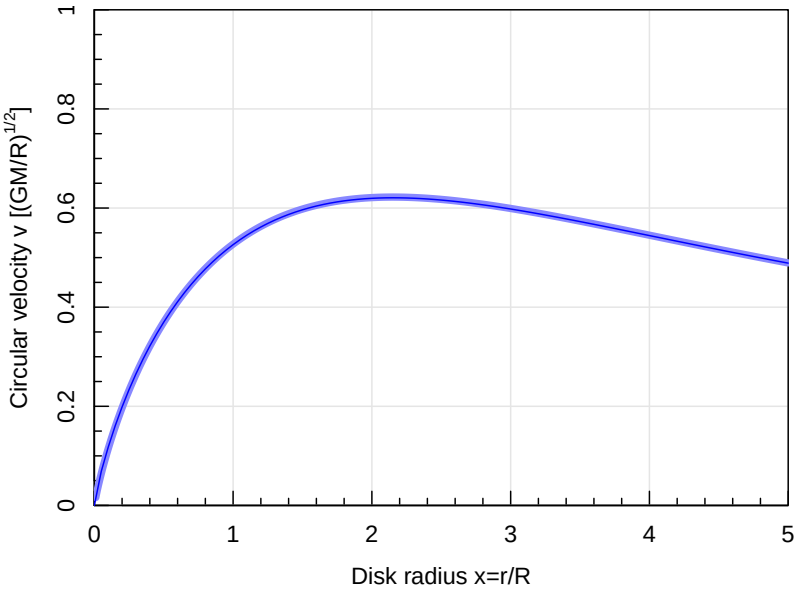
Analytical solution

As mentioned earlier, axially symmetric flat exponential galactic disks are sometimes called *Freeman* disks in honour of the Australian astrophysicist Ken Freeman, who laid down a lot of important analytical work on these systems (Freeman ApJ 160, 1970). Freeman found an analytical closed-form solution for $g(x)$ in terms of the modified Bessel functions I and K,

$$g^2(x) = \frac{x^2}{2}\left[I_0\left(\frac{x}{2}\right)K_0\left(\frac{x}{2}\right) - I_1\left(\frac{x}{2}\right)K_1\left(\frac{x}{2}\right)\right]. \tag{11}$$

Without following the challenging derivation, let us verify our numerical solution (thin line) against this Freeman solution (thick line):

```
g.freeman = function(x) {
  q = x/2
  x*sqrt((besselI(q,0)*besselK(q,0)-besselI(q,1)*besselK(q,1))/2)
}
par(mar=c(3,4,2,4)) # bottom,left,top,right
magcurve(g.freeman,0,5,500,col='#8888ff',lwd=5,xlim=c(0,5),ylim=c(0,1),xaxs='i',
         yaxs='i',xlab=expression('Disk radius x=r/R'),
         ylab=expression('Circular velocity v [(GM/R)'^'1/2'*']'))
curve(g,0,5,col='blue',add=TRUE)
```



Arbitrarily accurate computation

On most platforms, **R** represents real numbers as double-precision floating-point numbers. In this 8-byte representation, numbers are bound to approximately $\pm 2 \cdot 10^{308}$ and 15-16 significant digits. If larger values and/or more accuracy are required, the package **Rmpfr** is a remarkably powerful resource. This package is built on the GNU C library MPFR (short for "Multiple Precision Floating-Point Reliably"). It allows the user to specify an arbitrary number of bits in floating-point operations.

Let's start with an example. Native **R** can compute the factorial $n!$ up to $n = 170$, e.g. 25! is given by

```
factorial(25)
```

```
## [1] 1.551121e+25
```

However, of the 26 digits of this factorial, only the first 15-16 are expected to be correct, due to the limits of the double-precision representation. This can be seen by printing all the digits:

```
sprintf("%-.0f", factorial(25))
```

```
## [1] "15511210043330986055303168"
```

The last digits are obviously wrong, since they are known to be zeros. In order to evaluate this factorial exactly, it suffices to replace the argument by an **mprf** number, as follows:

```
n = mpfr(25,100)
factorial(n) # 100-bit floating-point computation
```

```
## 1 'mpfr' number of precision  100   bits
## [1] 15511210043330985984000000
```

The argument `100` specifies the number of bits used for the floating-point representation. One hundred is enough for 26 decimal digits. In fact, as a conservative rule of thumb, *4d bits are sufficient to exactly represent d decimals.*

As second example, let us evaluate $\cos(\pi/2)$, which, analytically, is exactly zero. If evaluated numerically, we find `cos(pi/2)=` 6.123234×10^{-17}, which defers from 0 to the level expected from double-precision accuracy. If we try the same operation using `mpfr`, via

```
cos(mpfr(pi/2,100))
```

```
## 1 'mpfr' number of precision  100   bits
## [1] 6.1232339957367658861303296613738e-17
```

the result doesn't actually improve, but the error has just more digits. This is because the value of `pi` is stored as a double-precision floating-point value. To benefit from the 100-bit floating-point accuracy, we need an equally accurate value for `pi`. To this end, we can use the function **Const**, which can evaluate standard mathematical constants. Evaluating π with 100-bit accuracy, is done via

```
my.pi = Const('pi',100)
```

Using this 100-bit representation of π, the error of $\cos(\pi/2)$ now drops to about 10^{-31}:

```
sprintf('%.5e',cos(my.pi/2))
```

```
## [1] "8.47843e-32"
```

Note that `mpfr` numbers work with all standard arithmetic operations (e.g. multiplication, addition) and base functions (e.g. `sin`, `exp`, `factorial`), but they may not work with derived functions and functions from external packages.

For more details about using the **Rmpfr** package, please refer to the built-in package documentation, as well as to the online documentation linked therein.

Interpolation & extrapolation

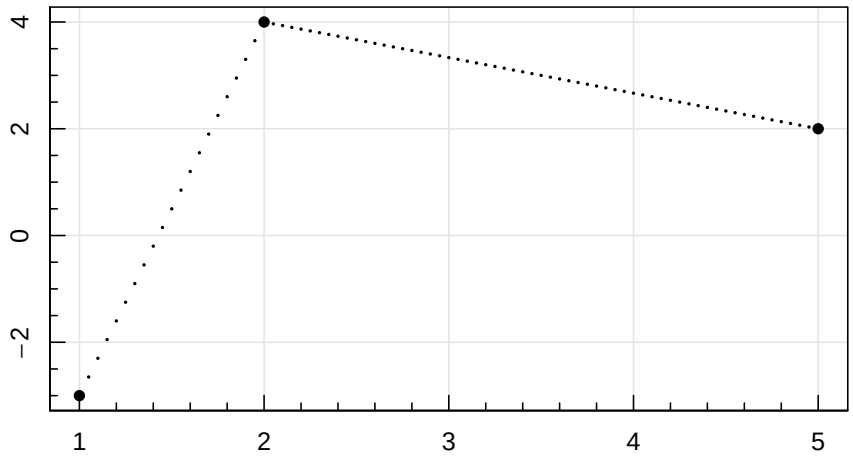
Interpolation encompasses methods for constructing new data points *in*side the range of a discrete dataset, whereas *extrapolation* refers to constructing new data points *out*side the range of the data. **R** offers many ways for interpolating and extrapolating data. Here, we overview the most common ones.

Interpolation & extrapolation of one-dimensional data

Standard methods to construct a function $f(x)$ interpolating (and sometimes extrapolating) the known points $\{x_i\}$ and $\{y_i = f(x_i)\}$ are listed in the following. In all cases `x` denotes the vector of known x-values and `y` denotes the vector of known y-values; and in all cases the values of `x` can be regularly or irregularly spaced and do not need to be ordered monotonically.

The most basic interpolation routine in **R** is the built-in **approx** routine. Calling **approx(x,y,xout)**, linearly interpolates the function $y(x)$, specified by `x` and `y` to the points `xout`. For instance,

```
x = c(1,2,5)
y = c(-3,4,2)
xout = seq(1,5,by=0.05)
yout = approx(x,y,xout)$y
magplot(x,y,pch=16)
points(xout,yout,pch=16,cex=0.3)
```



The big dots are the input points, whereas the little dots are linearly interpolated between them. Setting the optional argument `method='constant'` produces a constant interpolation, where the values of `yout` are equal to the y-value of the nearest x-value *to the left* of `xout`.

By default, **approx** does not extrapolate (i.e. it returns `NA` if `xout` lies beyond the range of `x`). However, constant extrapolation can be specified by setting the optional argument `rule=2`.

R also provides a built-in routine **spline** for spline-interpolation, called analogously to **approx**. Calling **spline(x,y,xout)** uses a natural spline fit to interpolate, except at the end points, where an exact cubic is fitted through the four points at each end of the data. In this mode, cubic extrapolation is enabled by default. Alternatively, the user can specify their preferred splining method by setting the optional argument `method`. Most importantly, **spline(x,y,xout,method='natural')** uses a natural spline for all points and linearly extrapolates the data beyond the range of `x`.

Fast interpolation functions

Sometimes, one needs an interpolation *function* rather than just interpolated values. For example imagine that we have to repeat the iterative operation $x \mapsto f(x)$ 10^3-times, where $f(x)$ is interpolated between 10^4 points. A *bad* way of doing this is calling **approx** 10^3-times, either directly, or packed into a function. In **R**, this would look something like this:

```
x = seq(0,1,length=1e4)
y = runif(1e4)
f = function(xout) approx(x,y,xout)$y
```

```
xout = 0
system.time(for (i in 1:1e3) xout = f(xout))
```

```
##    user  system elapsed
##   0.168   0.034   0.205
```

```
print(xout)
```

```
## [1] 0.754158
```

The problem with this approach is that each call of **approx(x,y,xout)** involves a lot of overhead to interpret the arguments `x` and `y`. This overhead is the same at each iteration and therefore adds unnecessary computation time. A much faster way of handling this task is offered by the routine **approxfun(x,y)**, which produces a *function* that can later be called much more efficiently. Let us try this:

```
f = approxfun(x,y)
xout = 0
system.time(for (i in 1:1e3) xout = f(xout))
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```

```
print(xout)
```

```
## [1] 0.754158
```

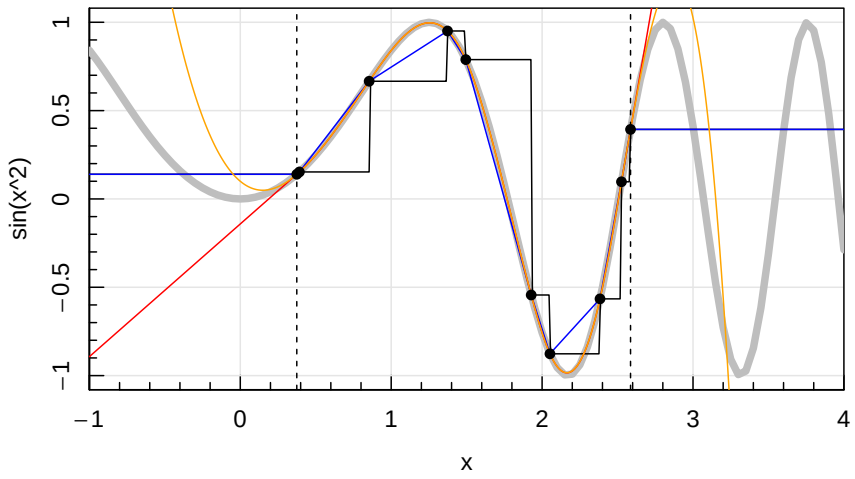
This code produces the same result as the previous one, but runs about 100-times faster!

Likewise, the routine **splinefun** returns an approximation function for the spline interpolator **spline**.

Summary

The following code illustrates some common types of interpolation/extrapolation, already described above:

```
x = runif(10,max=3)
y = sin(x^2)
f = list()
f[[1]] = approxfun(x,y,method='constant',rule=2)
f[[2]] = approxfun(x,y,rule=2)
f[[3]] = splinefun(x,y,method='natural')
f[[4]] = splinefun(x,y)
magcurve(sin(x^2),xlim=c(-1,4),ylim=c(-1,1),xaxs='i',lwd=5,col='grey')
for (i in (1:4)) {
  curve(f[[i]](x),-1,4,n=500,add=T,col=c('black','blue','red','orange')[i])
}
points(x,y,pch=16)
abline(v=c(min(x),max(x)),lty=2)
```



The thick grey line is the input function, which is provided to the interpolation routines in ten isolated points (black dots). The vertical dashed lines delimit the transition from interpolation to extrapolation. Four different interpolation/extrapolation functions are shown as thin lines:

- Constant interpolation (black): **approxfun(x,y,xout,method='constant')** produces an interpolation function that uses the y-values of the nearest x-value to the left (black line in figure below). Constant extrapolation has been enabled by the optional argument *rule=2*.
- Linear interpolation (blue): **approxfun(x,y,xout)** produces a linear interpolation between the given points. Constant extrapolation has been enabled by the optional argument `rule=2`.
- Natural spline interpolation (red): **splinefun(x,y,method='natural')** produces a natural spline interpolation function. This is a step-wise cubic function, which ensures a continuous first and second derivative. The second derivative is set to 0 at the end points (i.e. the function becomes a straight line at both ends). Linear extrapolation is enabled by default.
- FMM spline interpolation: **splinefun(x,y)** produces an alternative spline interpolation function (orange line), which doesn't assume a value for the second derivatives at the end points, but instead fits an exact cubic function through the four points at each end of the data. Cubic extrapolation is enabled by default.

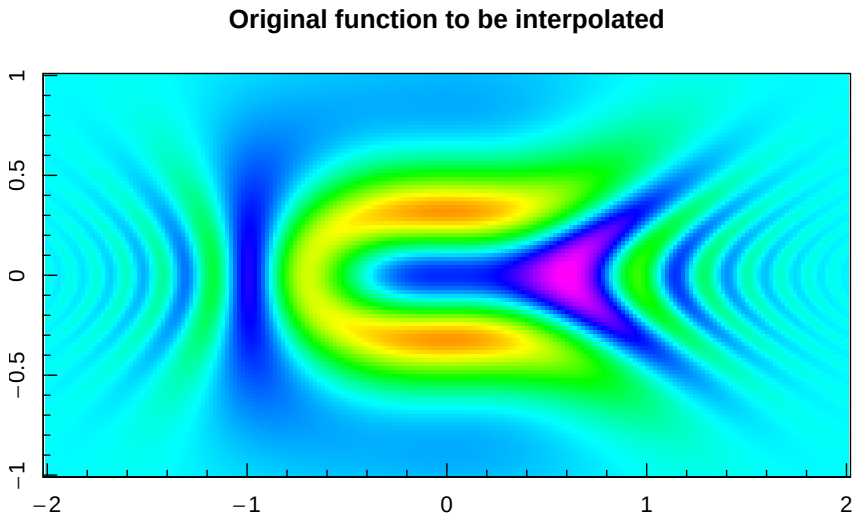
Interpolation & extrapolation of two-dimensional data

Sometimes it is necessary to interpolate/extrapolate a two-dimensional function $z = f(x, y)$ based on a set of discrete points. As an example we consider the function

$$f(x,y) = e^{-x^2-2y^2} \sin\left(\frac{1+x^3}{0.2+y^2}\right),$$

which we plot with the following code:

```
f = function(x,y) exp(-x^2-2*y^2)*cos((1+x^3)/(0.2+y^2))
x.grid = seq(-2,2,0.02)
y.grid = seq(-1,1,0.02)
m = meshgrid(x.grid,y.grid)
z.grid = t(f(m$X,m$Y))
col = rainbow(1000)
par(pin=c(6,3))
magimage(x.grid,y.grid,z.grid,zlim=c(-1,1),col=col,magmap=F,xlab='',ylab='')
title('Original function to be interpolated')
```

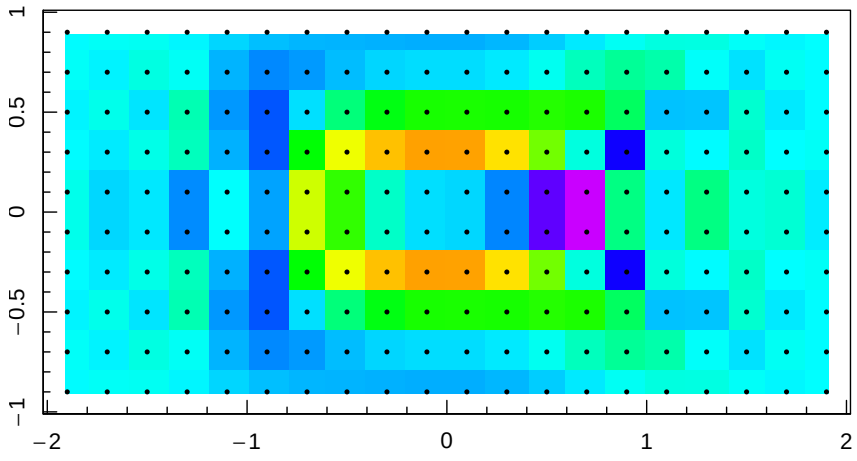


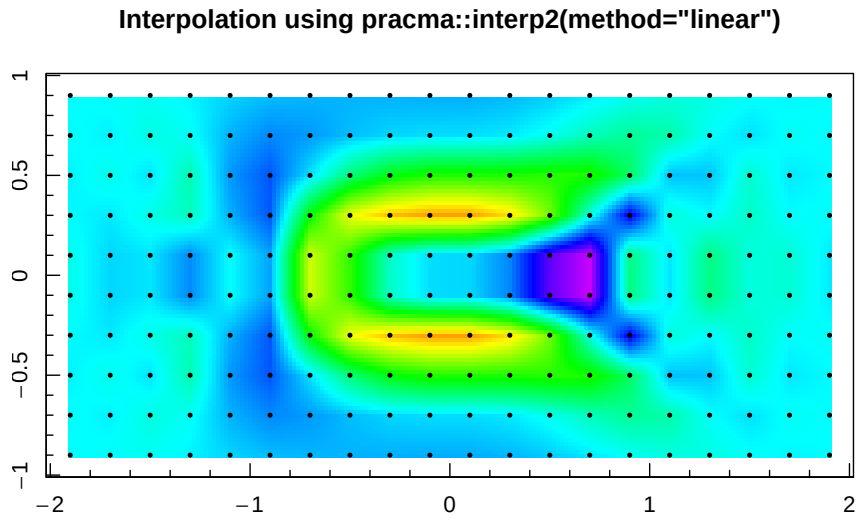
Two-dimensional interpolation methods normally depend on whether the data is provided on a grid (with equal or irregular spacing) or on a set of irregular points.

Interpolating data on a 2D grid

The routine **interp2** of the **pracma** package can be used to interpolate (but not extrapolate) between known data points on a regular 2D grid. It can handle linear (default) and constant interpolation. The following code illustrates this routine, using 20-by-10 points to interpolate the function $f(x, y)$.

```
x = seq(-1.9,1.9,length=20)
y = seq(-0.9,0.9,length=10)
m = meshgrid(x,y)
z = f(m$X,m$Y)
g = expand.grid(x.grid,y.grid)
xp = g[,1]
yp = g[,2]
for (method in c('nearest','linear')) {
  zp = interp2(x,y,z,xp,yp,method=method)
  z.grid = matrix(zp,ncol=length(y.grid))
  par(pin=c(6,3))
  magimage(x.grid,y.grid,z.grid,zlim=c(-1,1),col=col,magmap=F,xlab='',ylab='')
  title(sprintf('Interpolation using pracma::interp2(method="%s")',method))
  points(m$X,m$Y,pch=16,cex=0.5)
  cat('\n')
}
```

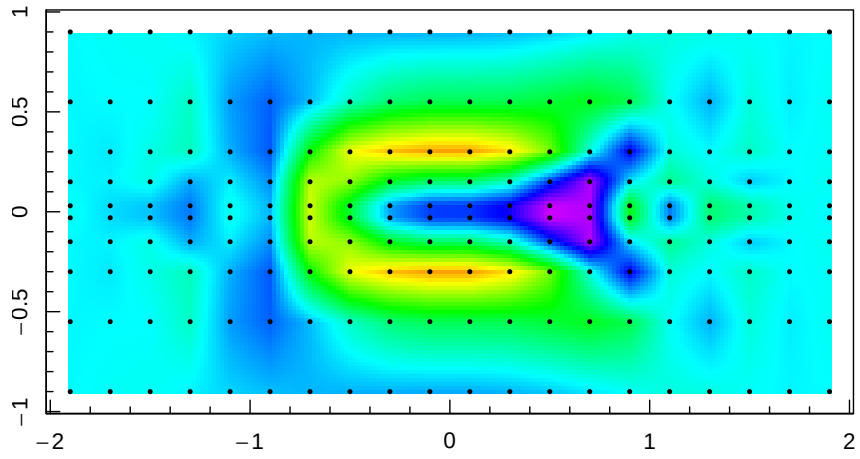




The region outside the range of the data points appears white in the figure, because **interp2** returns the value NA in this region.

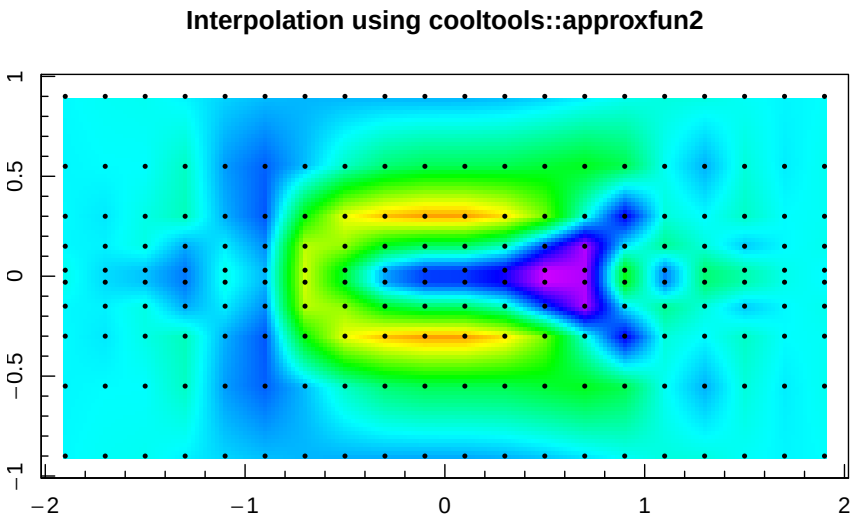
The grid lines do not need to be regularly spaced, however the elements of the vectors `x` and `y` in **interp2** have to be sorted in increasing order. For example, we can use this grid to better capture the central part along the y-coordinate:

```
x = seq(-1.9,1.9,length=20)
y = c(-0.9,-0.55,-0.3,-0.15,-0.03,0.03,0.15,0.3,0.55,0.9)
m = meshgrid(x,y)
z = f(m$X,m$Y)
g = expand.grid(x.grid,y.grid)
xp = g[,1]
yp = g[,2]
zp = interp2(x,y,z,xp,yp,method='linear')
z.grid = matrix(zp,ncol=length(y.grid))
par(pin=c(6,3))
magimage(x.grid,y.grid,z.grid,zlim=c(-1,1),col=col,magmap=F,xlab='',ylab='')
title('Interpolation using pracma::interp2(method="linear")')
points(m$X,m$Y,pch=16,cex=0.5)
```



Generating a 2D interpolation function. When discussing one-dimensional interpolation, we emphasized the important distinction between routines that perform the interpolation (such as **approx**) and routines that generate an interpolation function for later use (such as **approxfun**). A linear two-dimensional interpolation *function* can be generated by calling **approxfun2** from the **cooltools** package. Example using the same interpolation grid as before:

```
ap = approxfun2(x,y,t(z),outside=NA)
zp = ap(xp,yp)
z.grid = matrix(zp,ncol=length(y.grid))
par(pin=c(6,3))
magimage(x.grid,y.grid,z.grid,zlim=c(-1,1),col=col,magmap=F,xlab='',ylab='')
title('Interpolation using cooltools::approxfun2')
points(m$X,m$Y,pch=16,cex=0.5)
```



It is much faster to call the interpolation function *ap* than to call the interpolation routine **interp2**:

```
system.time(for(i in 1:1e3) interp2(x,y,z,runif(1),runif(1)))
```

```
##    user  system elapsed
##   0.024   0.000   0.024
```

```
system.time(for(i in 1:1e3) ap(runif(1),runif(1)))
```

```
##    user  system elapsed
##   0.007   0.000   0.007
```

Interpolating data on an irregular point set

The *interp* routine of the *akima* package is a powerful tool to interpolate two-dimensional functions. This routine uses either bilinear elements (default) or bicubic splines (by setting the argument `linear=FALSE`). If the input points lie on a grid (as in the examples above), only the bilinear method works. However, it is possible to add some jitter to the data to make the bicubic method work as well (see package documentation). In the case of bicubic splines, extrapolation is enabled by setting the optional argument `extrap=TRUE`.

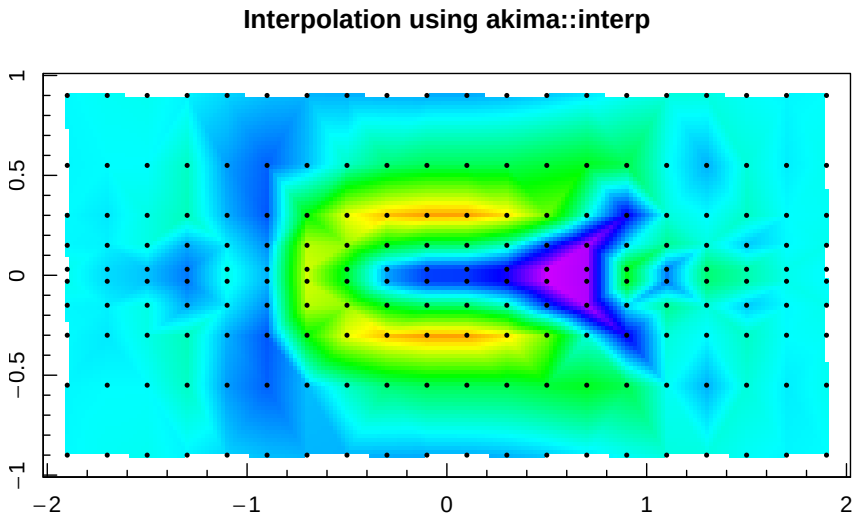
Let us first show the bilinear interpolation using the same grid as before. The linear interpolation scheme slightly differs from that used by **interp2** and **approxfun2**, leading to small visible differences:

```
z.grid = interp(as.vector(m$X),as.vector(m$Y),as.vector(z),x.grid,y.grid)$z
```

```
Warning in interp(as.vector(m$X), as.vector(m$Y), as.vector(z), x.grid, :
collinear points, trying to add some jitter to avoid colinearities!
```

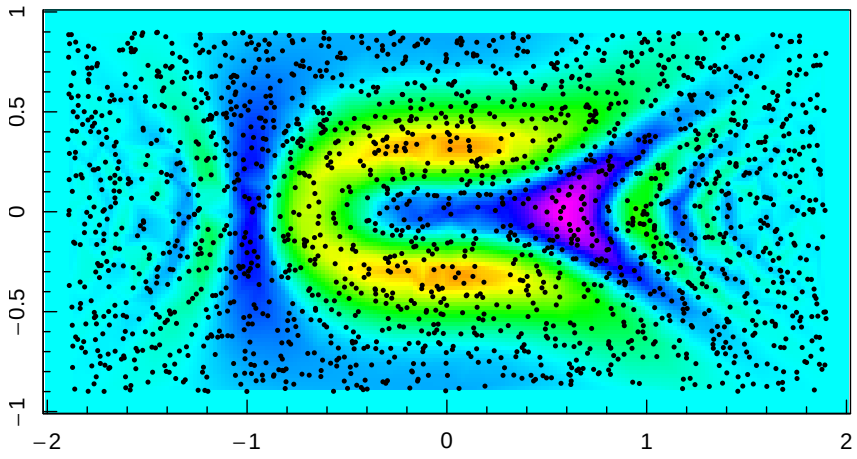
```
Warning in interp(as.vector(m$X), as.vector(m$Y), as.vector(z), x.grid, :
success: collinearities reduced through jitter
```

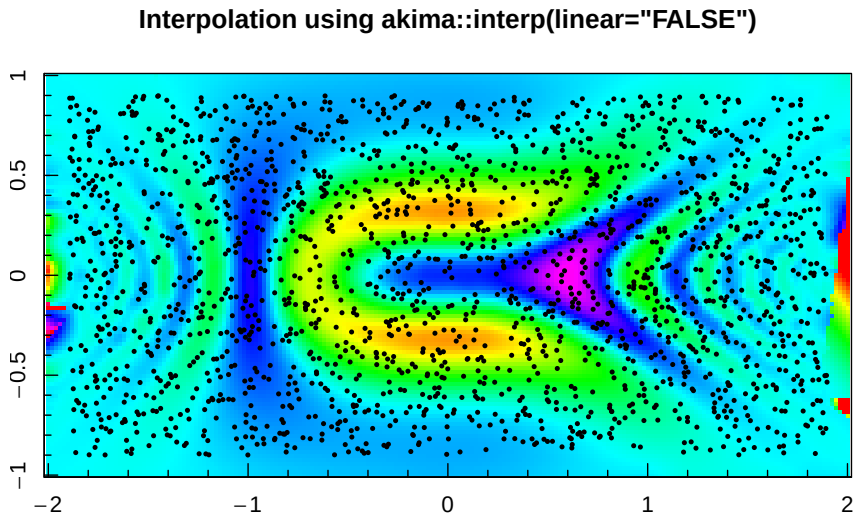
```
par(pin=c(6,3))
magimage(x.grid,y.grid,z.grid,zlim=c(-1,1),col=col,magmap=F,xlab='',ylab='')
title('Interpolation using akima::interp')
points(m$X,m$Y,pch=16,cex=0.5)
```



Finally, let us use an irregular grid and visualise the two interpolation schemes:

```
x = runif(2e3,-1.9,1.9)
y = runif(2e3,-0.9,0.9)
z = f(x,y)
for (linear in c(T,F)) {
  z.grid = lim(interp(x,y,z,x.grid,y.grid,linear=linear,extrap=T)$z,-1,1)
  par(pin=c(6,3))
  magimage(x.grid,y.grid,z.grid,zlim=c(-1,1),col=col,magmap=F,xlab='',ylab='')
  title(sprintf('Interpolation using akima::interp(linear="%s")',linear))
  points(x,y,pch=16,cex=0.5)
  cat('\n')
}
```





As we can see, the bilinear interpolation does not extrapolate the data, whereas the bicubic interpolation scheme can be used for extrapolation, too. However, bicubic interpolation and extrapolation can produce strong wiggles reaching outside the interval spanned by the data, which is why we truncated the range to the interval $[-1, 1]$ using the **lim** function of the **cooltools** package.

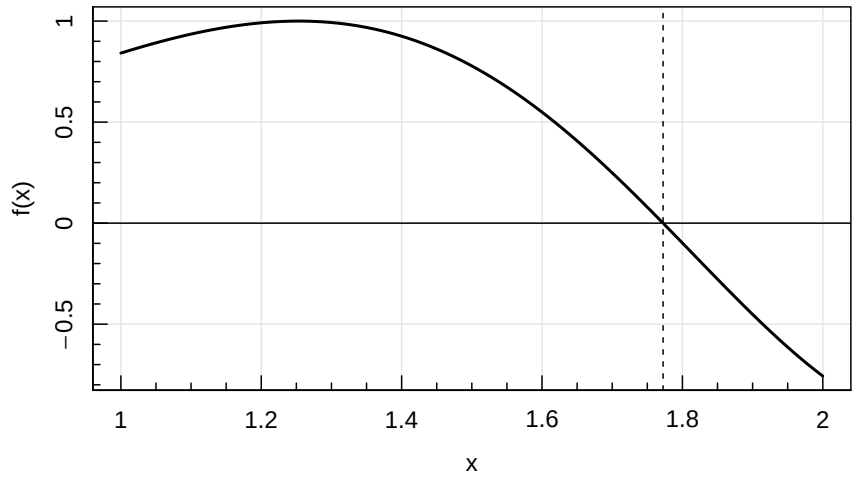
Miscellaneous useful routines

This section presents a collection of additional routines, which I regularly require in my research. I list them here for reference in no particular order and without providing much mathematical background. For each routine I give one or several examples that illustrates its main purpose. Please refer to the package documentations and links therein for background information.

Zero-point solver

Finding the zero-points (or 'roots') of a function is a very common problem in science. In **R**, this can be done using the **uniroot** function. E.g.

```
f = function(x) sin(x^2)
magcurve(f,1,2,lwd=2)
x = uniroot(f,c(1,2))
abline(h=0,v=x,lty=c(1,2))
```



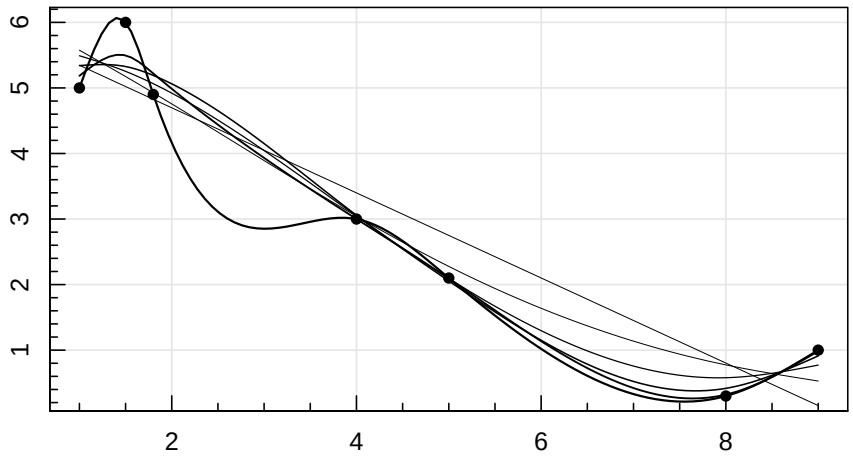
By default, **uniroot** assumes that the two limits of the interval contain a single root, implying that the value of f has opposite signs at the two end points. Other scenarios can be dealt with using the arguments (see help).

Smoothed splines

The routine **smooth.spline** can be used to smooth a function $y(x)$ with a spline function. The routine is called as **smooth.spline(x,y)**, where `x` and `y` are vectors of equal length that sample the function. The routine has many optional arguments specifying how exactly the smoothing is performed. Most importantly the parameter `df` specifies the polynomial degree (`df=2` for linear) of the smoothed function. This degree can lie anywhere between 2 (approximation by a straight line) and `length(x)`, in which case the smoothed function passes exactly through all the points. If not specified, a 'reasonable' value for `df` is determined automatically;

Calling **smooth.spline** returns an object of an identically-named class, which can then be evaluated by calling **predict**. The following example shows how this is done:

```
x = c(1,1.5,1.8,4,5,8,9)
y = c(5,6,4.9,3,2.1,0.3,1)
magplot(x,y,pch=16)
for (df in seq(2,length(x))) {
  ss = smooth.spline(x,y,df=df)
  smooth.function = function(x) predict(ss,x)$y
  curve(smooth.function,add=T,lwd=df*0.2)
}
```

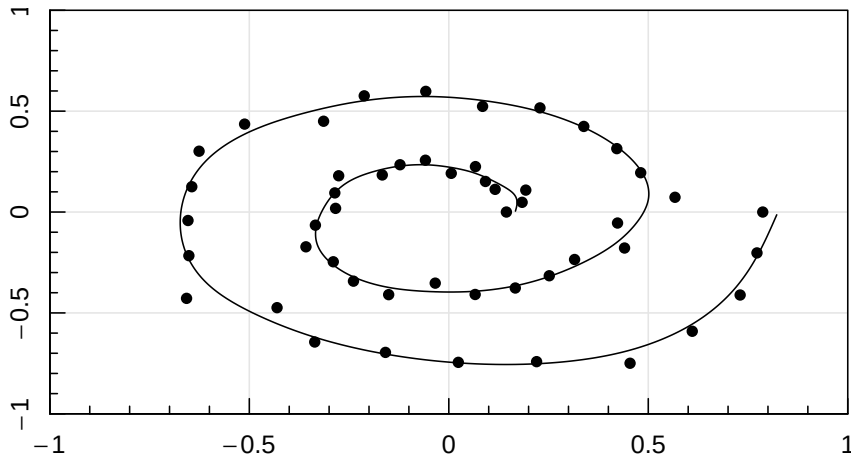


Smoothing parametric functions

One can use n splines to interpolate a parametric curve in n dimensions. Example for 2D:

```
# make points on a randomly perturbed spiral
t = seq(0,1,length=50)
radius = runif(length(t),0.1+0.65*t,0.2+0.75*t)
x = radius*cos(t*4*pi)
y = radius*sin(t*4*pi)
magplot(x,y,pch=16,xlim=c(-1,1),ylim=c(-1,1),xaxs='i',yaxs='i',xlab='',ylab='')

# smooth by a parametric spline
ssx = smooth.spline(t,x,df=length(x)/3)
ssy = smooth.spline(t,y,df=length(y)/3)
fx = function(t) predict(ssx,t)$y
fy = function(t) predict(ssy,t)$y
t = seq(0,1,length=1e3)
lines(fx(t),fy(t))
```



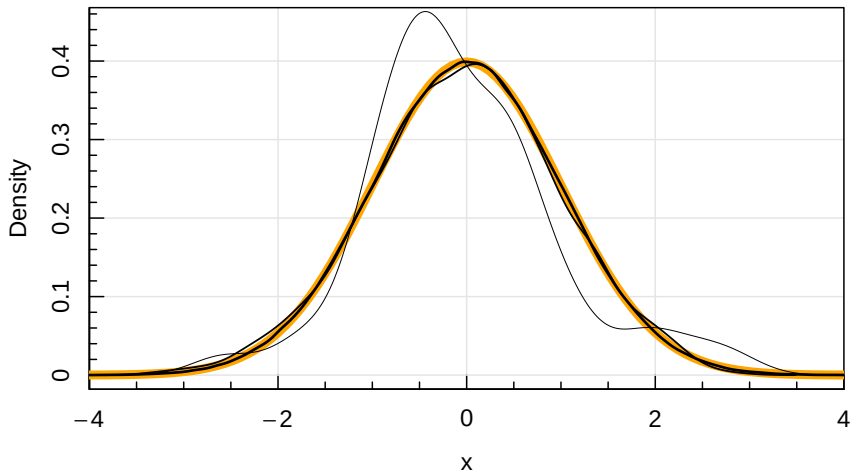
Density estimation

Imagine the following statistical problem: you are given a set of n random numbers $\mathbf{x} = x_1, ..., x_n$ that were drawn from an unknown probability density function (PDF) and you have to estimate the shape of this PDF. [Please see part 3 for an in-depth discussion of PDFs.]

This problem can be tackled using so-called kernel density estimation (KDE), a non-parametric way to estimate the probability density function of a random variable. KDE is a fundamental data smoothing problem where inferences about the population are made based on a finite data sample. Intuitively, it makes sense that a larger number of samples n will allow us to determine the population PDF in a more refined way than a small number of samples. KDE algorithms account for this by adaptively smoothing the data as a function of their local density. A detailed discussion of the mathematics is beyond the scope of this lecture. We note, however, that within some reasonable assumptions, KDE is a solvable problem.

The built-in function **density** performs a KDE and returns the estimated PDF on a grid of 512 points by default, but the number of points and range of this grid can be changed via the arguments. The following code illustrates how this works:

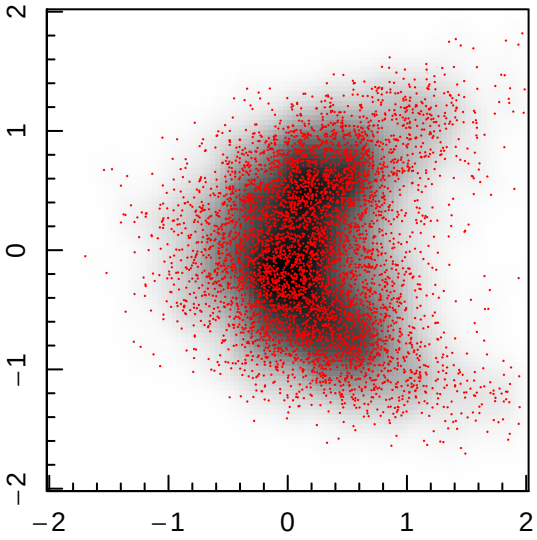
```
magcurve(dnorm,-4,4,lwd=6,col='orange',ylab='Density',ylim=c(0,0.45),xaxs='i')
for (i in seq(3)) {
  x = rnorm(c(50,2e3,1e6)[i])
  lines(density(x),lwd=i*0.6)
}
```



The true PDF of the population, a normal distribution (see part 3), is shown in orange. The black lines show the estimated PDFs based on 50, $2 \cdot 10^3$ and 10^6 random samples (thinnest to thickest line).

KDE algorithms also exist for higher dimensions. The base version of **R** does not include routines for this, however the **MASS** package provides the routine **kde2d** to estimate the two-dimensional PDF underlying a random set of 2D vectors. Example:

```
f = function(x) exp(-x[,1]^2-(x[,1]-x[,2]^2)^2)
x = rng(f,5e3,c(-2,-2),c(2,2),fmax=1)$x
d = kde2d(x[,1],x[,2],lims=c(-2,2,-2,2),n=100)
par(pty='s',cex=0.8)
magimage(d$x,d$y,-d$z,magmap=F)
points(x,pch=16,cex=0.2,col='red')
```

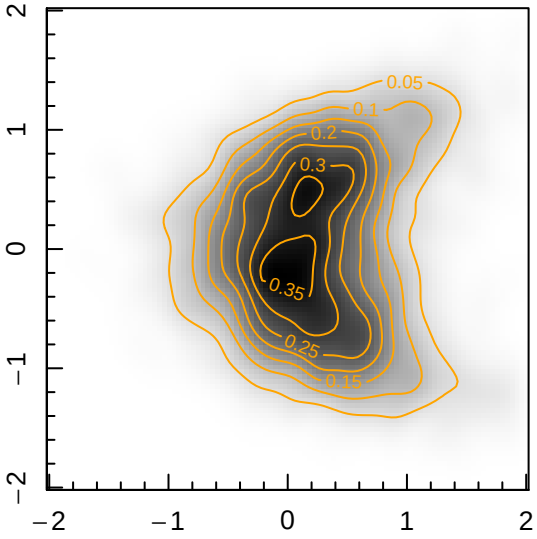


The random data is shown in red, while the estimated underlying PDF of the population is shown in shades of grey from white (0) to black (maximum).

Contours

In many statistical problems, it is useful to plot iso-contours around a two (or higher) dimensional data or parameter distributions. As an example, we might wish to plot contours of constant probability density on the previous figure. The built-in **contour** function can do this:

```
par(pty='s',cex=0.8)
magimage(d$x,d$y,-d$z,magmap=F)
contour(d$x,d$y,d$z,col='orange',add=T)
```

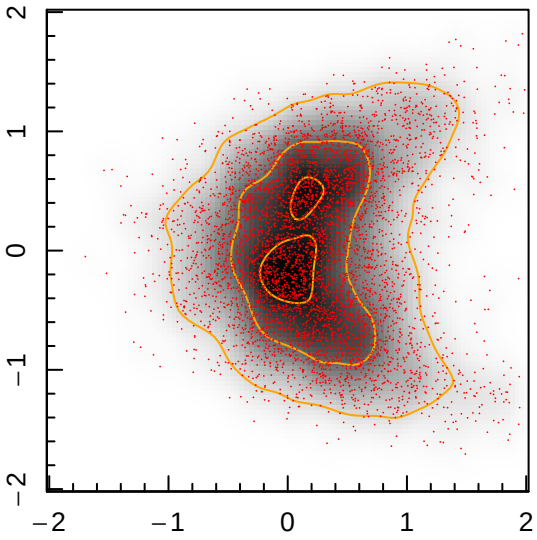


By default, **contour** automatically determines 10 contour levels, whose values are displayed on the contours. Use the arguments `nlevels` and `levels` to adjust these levels. Set `drawlabels=FALSE` to suppress the numbers on the contours.

Contours containing a specified probability mass. When dealing with PDFs one is often interested in plotting a contour that contains a specified probability mass. To do so, we need to determine the value of the PDF along the contour that contains this specified probability. Finding this level is a non-trivial task, as it involves solving an integral equation.

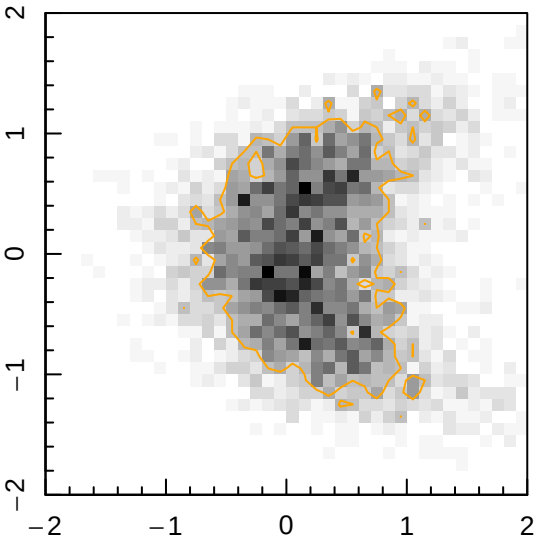
The **cooltools** package offers the routine **contourlevel** to solve this task. For instance, to draw the contours containing 10%, 50% and 90% of all samples use:

```
par(pty='s',cex=0.8)
magimage(d$x,d$y,1-d$z,magmap=F)
lev = contourlevel(d$z,c(0.1,0.5,0.9))
contour(d$x,d$y,d$z,col='orange',add=T,levels=lev,drawlabels=FALSE)
points(x,col='red',pch=16,cex=0.15)
```



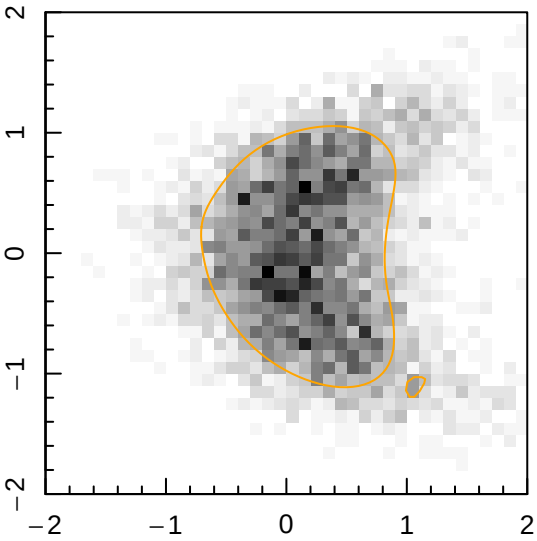
Smoothed contours. If a density field is quite noisy, the iso-contours are can be broken into many pieces. For instance, if we simply grid the samples above (using the **griddata** routine in **cooltools**), we obtain the following raster field and contour:

```
g = griddata(x,min=-2,max=2,n=40,type='counts')
par(pty='s',cex=0.8)
magimage(g$grid[[1]]$mid,g$grid[[2]]$mid,1-g$field,magmap=F)
contour(g$grid[[1]]$mid,g$grid[[2]]$mid,g$field,col='orange',add=T,levels=7,drawlabels=FALSE)
```



To smooth this contour, you can use the **smoothcontour** function included in the **cooltools** package:

```
par(pty='s',cex=0.8)
magimage(g$grid[[1]]$mid,g$grid[[2]]$mid,1-g$field,magmap=F)
smoothcontour(g$grid[[1]]$mid,g$grid[[2]]$mid,g$field,col='orange',levels=7,smoothing=0.8)
```



The level of smoothing can be adjusted via the arguments.

Matrix operations

As detailed in part 1, the base routines in **R** include very fast routines to perform standard matrix operations. As a brief reminder:

- Transpose $A^\intercal$: `t(A)`
- Matrix product AB: `A%*%B`
- Singular value decomposition, i.e. solve $A = UDV^\intercal$ for the diagonal matrix D with orthogonal matrices U and V: `svd(A)`

Particularly for square matrices:

- Diagonal vector $\mathrm{diag}(A)$: `diag(A)`
- Trace $\mathrm{Tr}(A)$: `sum(diag(A))`
- Inverse A^{-1}: `solve(A)`
- Determinant $|A|$: `det(A)`
- Eigen decomposition, i.e. solve $A\mathbf{x} = \lambda\mathbf{x}$ for λ and $\mathbf{x}$: `eigen(A)`

Covariance ellipses

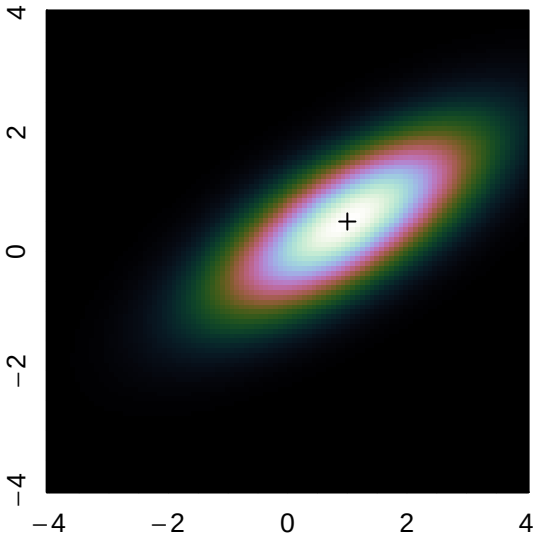
For some statistical applications encountered later in the course, it is worth pointing out that matrix inversion is frequently used in the context of covariance matrices (discussed in part 3). This is because one of the most fundamental distributions of several random variables, the *multivariate normal distribution*, requires the inverse of the covariance matrix of these variables. Explicitly, this distribution reads

$$\rho(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^D|\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\intercal\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})},$$

where $\mathbf{x} \in \mathbb{R}^D$ is the free variable, $\boldsymbol{\mu} \in \mathbb{R}^D$ is the mean of the distribution and $\boldsymbol{\Sigma}$ is the covariance matrix.

The **mvtnorm** package offers a range of routines to handle multivariate normal distributions. Let us give an example:

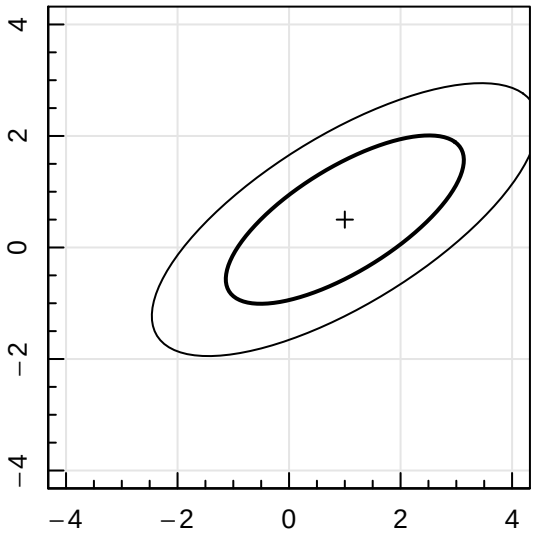
```
npixel = 100 # number of pixels a side
mu = c(1,0.5) # mean of multivariate distribution
sigma = matrix(c(2,1,1,1),nrow=2) # covariance matrix
v = seq(-4,4,length=npixel) # vector of grid lines
x = expand.grid(v,v) # array of x,y pixels
f = matrix(dmvnorm(x,mu,sigma),nrow=npixel) # evaluate mu distribution
par(pty='s',cex=0.8) # make square plot
magimage(v,v,f,col=cubehelix(200),magmap=F) # plot mu distribution
points(mu[1],mu[2],pch=3) # add cross at mean
```



Often, one is interested in plotting iso-contours, that is the locus of the points $\rho(\mathbf{x}) = k$ for some particular value k. This locus correspond to constant values of $(\mathbf{x} - \boldsymbol{\mu})^\intercal \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})$, which is an ellipse (in two dimensions), an ellipsoid (in three dimensions) or a 'hyper-ellipsoid' (in more dimensions), centred at $\boldsymbol{\mu}$. The directions of the axes of this ellipse/ellipsoid correspond to the eigenvectors of the covariance matrix $\boldsymbol{\Sigma}$; and the length of these axes are proportional to the square-root of the eigenvalues. (As an exercise, you could try to proof this and compute the proportionality factor of $\sqrt{\lambda}$.)

Conveniently, the function **ellipse** in the **ellipse** package computes the x- and y-coordinates of the elliptical contour containing a specified probability mass. For instance, to draw the 68% and 95% contours of the previous distribution use:

```
par(pty='s',cex=0.8)
magplot(ellipse(sigma,centre=mu,level=0.95),type='l',lwd=1,
        xlim=range(v),ylim=range(v),xlab='',ylab='')
lines(ellipse(sigma,centre=mu,level=0.68),lwd=2)
points(mu[1],mu[2],pch=3)
```


4
2
0
−2
−4
−4
−2
0
2
4