Statistics I Lectures

1 Numerical Linear Algebra

1.1 Matrix basics

Matrices and arrays are represented as “structures”: vectors (can therefore also be character or list) with a `dim` and optionally a `dimnames` attribute.

Creation via `matrix()`, `rbind()` and `cbind()`; `diag()` for creating diagonal matrices.

- `dim()` getter/setter for the `dim` attribute
- `nrow()` and `ncol()` for getting the number of rows or columns
- `dimnames()` getter/setter for the `dimnames` attribute
- `rownames()` and `colnames()` getters and setters for the row and column names

Subscripting of matrices:

- 2-argument using vectors of integers or logicals or chracters (if the matrix has the appropriate dimnames)
- 1-argument using a 2-column index matrix:

```
R> x <- diag(nrow = 3)
R> x

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1

R> i <- which(x > 0, arr.ind = TRUE)
R> i

     row col
[1,]   1   1
[2,]   2   2
[3,]   3   3

R> x[i]

[1] 1 1 1
```

- `diag()` can also be used for extracting the diagonal of a matrix. `lower.tri()` and `upper.tri()` can be employed for extracting the lower and upper diagonal parts of a matrix:

```
R> A <- matrix(1 : 9, 3, 3)
R> A

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
R> ## Extract diagonal elements.
R> diag(A)

[1] 1 5 9

R> ## Extract elements below the main diagonal.
R> A[lower.tri(A)]

[1] 2 3 6

R> ## Extract elements not above the main diagonal.
R> A[lower.tri(A, diag = TRUE)]

[1] 1 2 3 5 6 9

R> ## Extract elements above the main diagonal.
R> A[upper.tri(A)]

[1] 4 7 8
```

Basic "matrix" and "matrix"/"vector" operations work rather seamlessly, partially because recycling is involved.

Note: arithmetic and comparison operators work element-wise. In particular, `A * B` is the element-wise product of `A` and `B`, i.e., the *Hadamard* product.

Discuss the subtleties: there are no scalars, (row and column) vectors and matrices—instead there are sequences (vectors) which can have a dim attribute.

`t()` computes the transpose; `%*%` does the matrix product.

`crossprod()` and `tcrossprod()` compute the matrix products $A'B$ and AB', respectively, more efficiently than via the direct `t(A) %*% B` and `A %*% t(B)`.

`kronecker()` (actually, is more general: `%x%` only does the basic) computes the Kronecker product $A \otimes B$, defined as follows: if $A = [\alpha_{ij}]$, then

$$A \otimes B = \begin{bmatrix} \alpha_{11}B & \cdots & \alpha_{1n}B \\ \vdots & \ddots & \vdots \\ \alpha_{m1}B & \cdots & \alpha_{mn}B \end{bmatrix}$$

Review fundamental properties of the Kronecker product, such as

$$(A \otimes B)' = A' \otimes B', \qquad (A \otimes B)(C \otimes D) = AC \otimes BD, \qquad (A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$$

and, introducing the vec operator,

$$\mathrm{vec}(ABC) = (C' \otimes A)\mathrm{vec}(B).$$

Example. Multiplying the rows of a matrix A by a vector v. Formally,

$$\mathrm{diag}(v)A$$

(manipulating rows corresponds to pre-multiplying by a suitable matrix), but implement as

```
R> rmult <- function(A, v) v * A
```

Example. Multiplying the columns of a matrix A by a vector v. Formally,

$$A\mathrm{diag}(v)$$

(manipulating the columns corresponds to post-multiplying by a suitable matrix), but implement as

```
R> cmult <- function(A, v) A * rep(v, each = nrow(A))
```

Example. Trace of the crossprod, formally `sum(diag(crossprod(A)))`. As $[A'A]_{ij} = \sum_k \alpha_{ki}\alpha_{kj}$,

$$\mathrm{trace}(A'A) = \sum_i [A'A]_{ij} = \sum_i \sum_k \alpha_{ki}\alpha_{ki} = \sum_{i,k} \alpha_{ik}^2,$$

we can simply do

```
R> trace_of_crossprod <- function(A) sum(A ^ 2)
```

`outer()` performs the (generalized) outer product of vectors or matrices (and arrays), generalizing the basic outer product xy' of vectors. For vectors,

$$[\mathrm{outer}(f, x, y)]_{ij} = f(x_i, y_j)$$

where by default, f does multiplication and needs to be vectorized.

Example. The Vandermonde matrix of a sequence $x_1, \dots, x_n$ is

$$V(x_1, \dots, x_n) = [x_i^{j-1}]_{i,j=1,\dots,n}.$$

This can straightforwardly be implemented as

```
R> Vandermonde <- function(x) outer(x, seq(0, length(x) - 1), "^")
R> x <- 1 : 5
R> Vandermonde(x)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    1    2    4    8   16
[3,]    1    3    9   27   81
[4,]    1    4   16   64  256
[5,]    1    5   25  125  625
```

A famous result is that the determinant of the Vandermonde matrix is given by

$$\det(V(x_1, \dots, x_n)) = \prod_{i<j} (x_i - x_j)$$

How can we implement this?

```
R> Vandermonde_det <- function(x) {
+     diffs <- outer(x, x, `-`)
+     prod(diffs[upper.tri(diffs)])
+ }
R> ## check:
R> det(Vandermonde(x))
```

```
[1] 288
```

```
R> Vandermonde_det(x)
```

```
[1] 288
```

`apply()` applies functions over array margins: in the simplest case, to the rows or columns of a matrix.

```
R> A <- matrix(1 : 9, 3, 3)
R> ## Row sums
R> apply(A, 1, sum)
```

```
[1] 12 15 18
```

```
R> ## Col sums
R> apply(A, 2, sum)
```

```
[1]  6 15 24
```

For computing row/col sums and means, functions `rowSums()`/`colSums()` and `rowMeans()`/`colMeans()` are much faster than the straightforward apply-based versions.

`sweep()` is a higher-order function for sweeping out summaries from the margins of arrays. E.g., to center the rows of a matrix:

```
R> A <- matrix(1 : 9, 3, 3)
R> sweep(A, 1, rowMeans(A))
```

```
     [,1] [,2] [,3]
[1,]   -3    0    3
[2,]   -3    0    3
[3,]   -3    0    3
```

1.2 Matrix decompositions and linear systems

Solving linear system: basic case $Ax = b$, A an invertible square matrix. Then formally $x = A^{-1}b$, which can literally be translated to `x <- solve(A) %*% b`, but *do not do this.* Instead, use `solve(A, b)` or `qr.solve(A, b)`.

To illustrate matters, use

```
R> b <- 1 : 6
```

and consider the linear system

$$H_6 x = b$$

where H_6 is the 6×6 Hilbert matrix whose (i, j) entry is given by $1/(i + j - 1)$.

```
R> H <- 1 / outer(b - 1, b, `+`)
R> H
```

```
          [,1]      [,2]      [,3]      [,4]      [,5]       [,6]
[1,] 1.0000000 0.5000000 0.3333333 0.2500000 0.2000000 0.16666667
[2,] 0.5000000 0.3333333 0.2500000 0.2000000 0.1666667 0.14285714
[3,] 0.3333333 0.2500000 0.2000000 0.1666667 0.1428571 0.12500000
[4,] 0.2500000 0.2000000 0.1666667 0.1428571 0.1250000 0.11111111
[5,] 0.2000000 0.1666667 0.1428571 0.1250000 0.1111111 0.10000000
[6,] 0.1666667 0.1428571 0.1250000 0.1111111 0.1000000 0.09090909
```

Compute the inverse and check whether it does a reasonable job:

```
R> H_inv <- solve(H)
R> H_inv %*% H
```

```
           [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
[1,]  1.000e+00 -1.062e-10 -9.004e-11 -7.805e-11 -6.889e-11 -6.179e-11
[2,]  2.765e-10  1.000e+00  1.910e-10  1.655e-10  1.419e-10  1.246e-10
[3,] -1.455e-10 -5.821e-11  1.000e+00 -8.004e-11 -4.366e-11 -4.366e-11
[4,]  1.746e-10  1.164e-10  5.821e-11  1.000e+00  2.910e-11  5.821e-11
[5,]  2.328e-10  2.910e-11  8.731e-11  8.731e-11  1.000e+00  8.731e-11
[6,] -5.821e-11  0.000e+00  0.000e+00 -4.366e-11 -4.366e-11  1.000e+00
```

```
R> max(abs(H_inv %*% H) - diag(6))
```

```
[1] 2.765e-10
```

Hmm. Only up to 10 digits for a 6×6 matrix? This is not really impressive. Anyways: compute "solutions" of $H_6 x = b$ using the 3 indicated methods:

```
R> x1 <- c(H_inv %*% b)
R> ## (Use c() to obtain a dim-less vector.)
R> x2 <- solve(H, b)
R> x3 <- qr.solve(H, b)
```

How close are these?

```
R> x1 - x2
```

```
[1] -2.787e-09  5.567e-09 -9.095e-10  1.804e-09  4.366e-10 -2.474e-10
```

```
R> max(abs(x1 - x2))
```

```
[1] 5.567e-09
```

and compactly:

```
R> dist(rbind(x1, x2, x3), "maximum")

          x1        x2
x2 5.567e-09
x3 1.415e-05 1.415e-05
```

But how “good” are the solutions?

```
R> b1 <- H %*% x1
R> b2 <- H %*% x2
R> b3 <- H %*% x3
R> ## Inspect the difference to b:
R> cbind(b1, b2, b3) - b

          [,1]      [,2]       [,3]
[1,] 1.910e-10 0.000e+00 -3.638e-12
[2,] 6.312e-10 3.638e-12 -5.457e-12
[3,] 6.112e-10 1.819e-12  1.819e-12
[4,] 5.512e-10 0.000e+00 -5.457e-12
[5,] 4.893e-10 1.819e-12 -1.819e-12
[6,] 4.384e-10 2.728e-12 -3.638e-12

R> ## And the maximal differences:
R> apply(abs(cbind(b1, b2, b3) - b), 2, max)

[1] 6.312e-10 3.638e-12 5.457e-12
```

LU **decomposition.** Very nice in theory (cf. reduction to row echelon form), but not so nice from a numerical perspective—hence, not in base R (recommended package Matrix provides `lu()`).

QR **decomposition.** Via `qr()`, which returns something “strange”:

```
R> H_qr <- qr(H)
R> H_qr

$qr
        [,1]    [,2]      [,3]       [,4]       [,5]       [,6]
[1,] -1.2212 -0.7019 -0.504470 -0.3969691 -3.284e-01 -2.806e-01
[2,]  0.4094 -0.1385 -0.151130 -0.1443644 -1.340e-01 -1.237e-01
[3,]  0.2730  0.5029 -0.009562 -0.0151932 -1.813e-02 -1.953e-02
[4,]  0.2047  0.4675  0.419826  0.0004803  9.942e-04  1.419e-03
[5,]  0.1638  0.4221  0.595589 -0.3630074  1.734e-05  4.403e-05
[6,]  0.1365  0.3804  0.680569 -0.8985478  3.663e-01  3.986e-07

$rank
[1] 6

$qraux
[1] 1.819e+00 1.453e+00 1.076e+00 1.247e+00 1.930e+00 3.986e-07
```

```
$pivot
[1] 1 2 3 4 5 6

attr(,"class")
[1] "qr"
```

(The upper triangle contains the R of the decomposition and the lower triangle contains information on the Q of the decomposition, stored in compact form.) The Q and R can be retrieved using `qr.Q()` and `qr.R()`, respectively.

How well can H_6 be recovered from its QR decomposition?

```
R> max(abs(qr.Q(H_qr) %*% qr.R(H_qr) - H))
```

```
[1] 2.22e-16
```

(not bad).

As

$$H_6 x = QRx = b \leftrightarrow Rx = Q'b,$$

we can solve $H_6 x = b$ "by hand" using

```
R> x3a <- c(backsolve(qr.R(H_qr), crossprod(qr.Q(H_qr), b)))
R> x3a - x3
```

```
[1]  2.905e-11 -1.182e-10  6.839e-10 -1.746e-09  1.834e-09 -6.839e-10
```

but in fact more correctly

```
R> x3a <- c(backsolve(qr.R(H_qr), qr.qty(H_qr, b)))
R> x3a - x3
```

```
[1] 0 0 0 0 0 0
```

Note that the determinant of an upper or lower diagonal matrix is the product of its diagonal elements, so we can obtain the absolute value of the determinant (Q can have det ± 1) via

```
R> abs(prod(diag(qr.R(H_qr))))
```

```
[1] 5.367e-18
```

and see (again?) that H_6 is nearly singular.

Singular value decomposition (SVD). Via `svd()`:

```
R> H_svd <- svd(H)
R> H_svd
```

```
$d
[1] 1.619e+00 2.424e-01 1.632e-02 6.157e-04 1.257e-05 1.083e-07

$u
        [,1]    [,2]    [,3]     [,4]     [,5]      [,6]
[1,] -0.7487  0.6145 -0.2403 -0.06223  0.01114 -0.001248
[2,] -0.4407 -0.2111  0.6977  0.49084 -0.17973  0.035607
[3,] -0.3207 -0.3659  0.2314 -0.53548  0.60421 -0.240679
[4,] -0.2543 -0.3947 -0.1329 -0.41704 -0.44357  0.625460
[5,] -0.2115 -0.3882 -0.3627  0.04703 -0.44154 -0.689807
[6,] -0.1814 -0.3707 -0.5028  0.54068  0.45911  0.271605

$v
        [,1]    [,2]    [,3]     [,4]     [,5]      [,6]
[1,] -0.7487  0.6145 -0.2403 -0.06223  0.01114 -0.001248
[2,] -0.4407 -0.2111  0.6977  0.49084 -0.17973  0.035607
[3,] -0.3207 -0.3659  0.2314 -0.53548  0.60421 -0.240679
[4,] -0.2543 -0.3947 -0.1329 -0.41704 -0.44357  0.625460
[5,] -0.2115 -0.3882 -0.3627  0.04703 -0.44154 -0.689807
[6,] -0.1814 -0.3707 -0.5028  0.54068  0.45911  0.271605
```

Verify that U and V are orthonormal:

```
R> crossprod(H_svd$u)
```

```
           [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
[1,]  1.000e+00 -8.327e-17  8.327e-17  0.000e+00 -1.388e-17  2.082e-17
[2,] -8.327e-17  1.000e+00 -2.776e-17  0.000e+00  8.327e-17  4.163e-17
[3,]  8.327e-17 -2.776e-17  1.000e+00 -1.665e-16  2.498e-16 -5.551e-17
[4,]  0.000e+00  0.000e+00 -1.665e-16  1.000e+00  1.665e-16  5.551e-17
[5,] -1.388e-17  8.327e-17  2.498e-16  1.665e-16  1.000e+00 -4.163e-17
[6,]  2.082e-17  4.163e-17 -5.551e-17  5.551e-17 -4.163e-17  1.000e+00
```

```
R> crossprod(H_svd$v)
```

```
           [,1]       [,2]       [,3]       [,4]       [,5]       [,6]
[1,]  1.000e+00  2.914e-16 -2.914e-16  1.388e-17  1.388e-17  6.939e-18
[2,]  2.914e-16  1.000e+00 -2.776e-16 -1.110e-16 -2.776e-17  1.388e-17
[3,] -2.914e-16 -2.776e-16  1.000e+00 -1.110e-16 -1.665e-16  2.776e-17
[4,]  1.388e-17 -1.110e-16 -1.110e-16  1.000e+00  8.327e-17 -2.220e-16
[5,]  1.388e-17 -2.776e-17 -1.665e-16  8.327e-17  1.000e+00  1.388e-17
[6,]  6.939e-18  1.388e-17  2.776e-17 -2.220e-16  1.388e-17  1.000e+00
```

or more compactly

```
R> max(abs(crossprod(H_svd$u) - diag(6)))
```

```
[1] 5.551e-16
```

```
R> max(abs(crossprod(H_svd$v) - diag(6)))
```

```
[1] 1.11e-15
```

How well can H_6 be recovered from its SVD?

```
R> max(abs(tcrossprod(cmult(H_svd$u, H_svd$d), H_svd$v) - H))
```

```
[1] 2.22e-16
```

which is quite impressive!

The determinant can be obtained as

```
R> prod(H_svd$d)
```

```
[1] 5.367e-18
```

What if we tried to use

$$b = H_6 x = UDV'x \leftrightarrow x = VD^{-1}U'b$$

to solve $H_6 x = b$?

```
R> x4 <- cmult(H_svd$v, 1 / H_svd$d)  %*% crossprod(H_svd$u, b)
R> b4 <- H %*% x4
R> b4 - b
```

```
           [,1]
[1,]  1.091e-11
[2,]  5.457e-12
[3,]  5.457e-12
[4,]  5.457e-12
[5,]  3.638e-12
[6,] -1.819e-12
```

Again, very impressive.

Eigendecomposition. Aka spectral decomposition, via `eigen()`:

```
R> H_eigen <- eigen(H)
```

Compare the eigenvalues to the singular values (which should really be the same):

```
R> H_eigen$values - H_svd$d
```

```
[1] -4.441e-16 -6.384e-16 -5.551e-17 -1.258e-17 -1.696e-17 -1.787e-17
```

Could repeat the above program: how well can H_6 be recovered from the eigendecomposition? What if we solved $H_6 x = UDU'x = b$ via $x = U'D^{-1}Ux$? Discuss: SVD and eigendecomposition should really be "the same", but are not.

```
R> U <- H_eigen$vectors
R> x5 <- cmult(U, 1 / H_eigen$values) %*% crossprod(U, b)
R> b5 <- H %*% x5
R> b5 - b
```

```
             [,1]
[1,] -3.638e-12
[2,]  3.638e-12
[3,] -3.638e-12
[4,]  1.819e-12
[5,] -1.819e-12
[6,]  0.000e+00
```

Choleski decomposition. Use `chol()` to obtain the Choleski decomposition $A = R'R$ of a symmetric positive-definite matrix A:

```
R> H_chol <- chol(H)
```

And look:

```
R> crossprod(H_chol) - H

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    0    0    0    0    0    0
[3,]    0    0    0    0    0    0
[4,]    0    0    0    0    0    0
[5,]    0    0    0    0    0    0
[6,]    0    0    0    0    0    0
```

Does this help for solving linear systems? Try solving $H_6 x = R'Rx = b$ via $R'y = b$, $Rx = y$:

```
R> x6 <- backsolve(H_chol, forwardsolve(t(H_chol), b))
R> b6 <- H %*% x6
R> b6 - b

             [,1]
[1,]  1.819e-12
[2,]  1.091e-11
[3,]  5.457e-12
[4,]  1.819e-12
[5,] -1.819e-12
[6,]  2.728e-12
```

And to summarize matters:

```
R> ## Compare solutions:
R> dist(t(cbind(x1, x2, x3, x4, x5, x6)), "max")

          x1        x2        x3
x2 5.567e-09
x3 1.415e-05 1.415e-05
   1.147e-05 1.147e-05 2.679e-06
   2.000e-05 2.000e-05 3.415e-05 3.147e-05
x6 8.380e-06 8.379e-06 2.252e-05 1.985e-05 1.162e-05

R> ## and their quality:
R> apply(abs(cbind(b1, b2, b3, b4, b5, b6) - b), 2, max)

[1] 6.312e-10 3.638e-12 5.457e-12 1.091e-11 3.638e-12 1.091e-11
```

2 Simulation

2.1 Basics

Uniform (on the unit interval) pseudorandom numbers can be simulated using multiplicative congruential random number generators which use recursions

$$x_n = bx_{n-1} \pmod{m}, \qquad u_n = x_n/m$$

for suitable initial seed x_0.

To illustrate:

```
R> myrng <- function(n, m, b, x) {
+     u <- numeric(n)
+     for(i in 1 : n) {
+         x <- (b * x) %% m
+         u[i] <- x / m
+     }
+     u
+ }
```

BnM Example 5.1:

```
R> myrng(7, 7, 3, 2)
```

```
[1] 0.8571429 0.5714286 0.7142857 0.1428571 0.4285714 0.2857143 0.8571429
```

BnM example for bad:

```
R> myrng(5, 29241, 171, 3)
```

```
[1] 0.01754386 0.00000000 0.00000000 0.00000000 0.00000000
```

We note that the maximal period length is $m - 1$, as 0 “absorbs”.

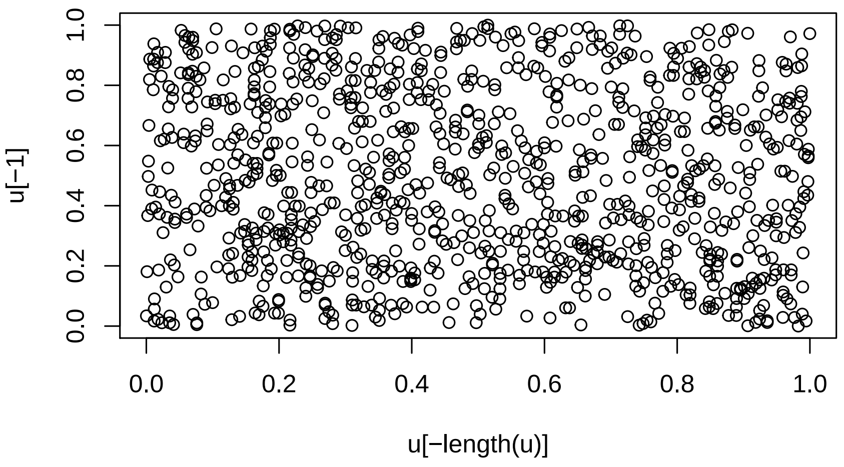
BnM Example 5.2:

```
R> myrng(50, 30269, 171, 27218)
```

```
 [1] 0.76385080 0.61848756 0.76137302 0.19478675 0.30853348 0.75922561
 [7] 0.82757937 0.51607255 0.24840596 0.47741914 0.63867323 0.21312234
[13] 0.44391952 0.91023820 0.65073177 0.27513297 0.04773861 0.16330239
[19] 0.92470845 0.12514454 0.39971588 0.35141564 0.09207440 0.74472232
[25] 0.34751726 0.42545178 0.75225478 0.63556774 0.68208398 0.63636063
[31] 0.81766824 0.82126929 0.43704780 0.73517460 0.71485678 0.24051009
[37] 0.12722587 0.75562457 0.21180085 0.21794575 0.26872378 0.95176583
[43] 0.75195745 0.58472364 0.98774324 0.90409330 0.59995375 0.59209092
[49] 0.24754700 0.33053619
```

Illustrating the “randomness”:

```
R> u <- myrng(1000, 30269, 171, 27218)
R> plot(u[-length(u)], u[-1])
```



In R: built-in uniform (pseudo-)random number generator `runif()`. In fact, for many distributions, there are density ("d"), cumulative probability (distribution) function ("p"), quantile ("q") functions, and a function for generating (pseudo-) random variates ("r"). E.g.,

```
  dunif(x, min=0, max=1, log = FALSE)
  punif(q, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
  qunif(p, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
  runif(n, min=0, max=1)
```

Note that the normal distribution is parametrized as a location-scale family, i.e., with its mean and its *standard deviation*:

```
  dnorm(x, mean = 0, sd = 1, log = FALSE)
  pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
  qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
  rnorm(n, mean = 0, sd = 1)
```

Note that the exponential distribution is parametrized by its *rate*, so that its expectation is the inverse of its parameter:

```
  dexp(x, rate = 1, log = FALSE)
  pexp(q, rate = 1, lower.tail = TRUE, log.p = FALSE)
  qexp(p, rate = 1, lower.tail = TRUE, log.p = FALSE)
  rexp(n, rate = 1)
```

Finally, notice that function `sample()` samples given values, with or without replacement:

```
  sample(x, size, replace = FALSE, prob = NULL)
```

Illustration of central limit theorems. If X is binomial with parameters m and p and

$$Z = \frac{X - mp}{\sqrt{mp(1-p)}},$$

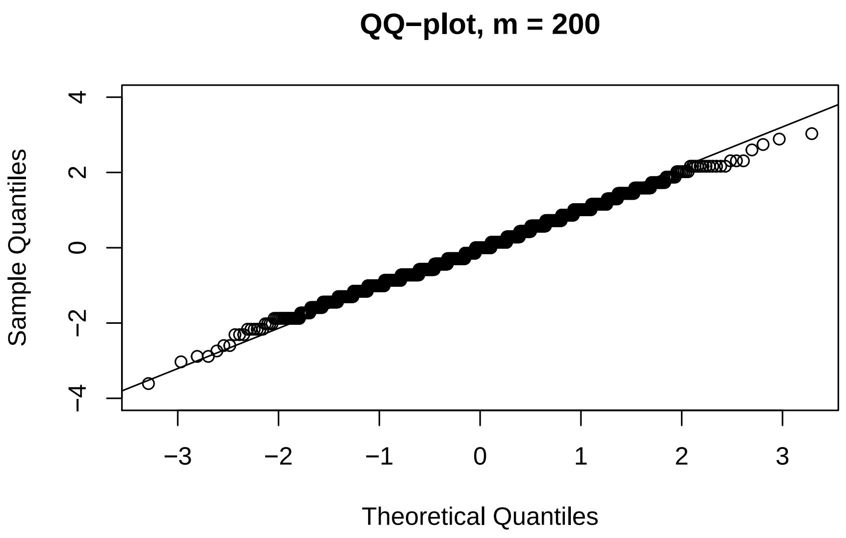
then Z is approximately standard normal when m gets large.

A simple simulation:

```
R> simbin <- function(n, m, p) {
+     z <- ((rbinom(n, size = m, prob = p) - m * p) /
+           sqrt(m * p * (1 - p)))
+     qqnorm(z, ylim = c(-4, 4), main = paste("QQ-plot, m =", m))
+     qqline(z)
+ }
```

E.g., for $m = 200$:

```
R> simbin(1000, 200, 0.4)
```



A simple movie:

```
R> for(m in seq(1, 100, 3)) {
+     simbin(1000, m, 0.4)
+     Sys.sleep(1)
+ }
```

If X is poisson with parameter λ and

$$Z = \frac{X - \lambda}{\sqrt{\lambda}},$$

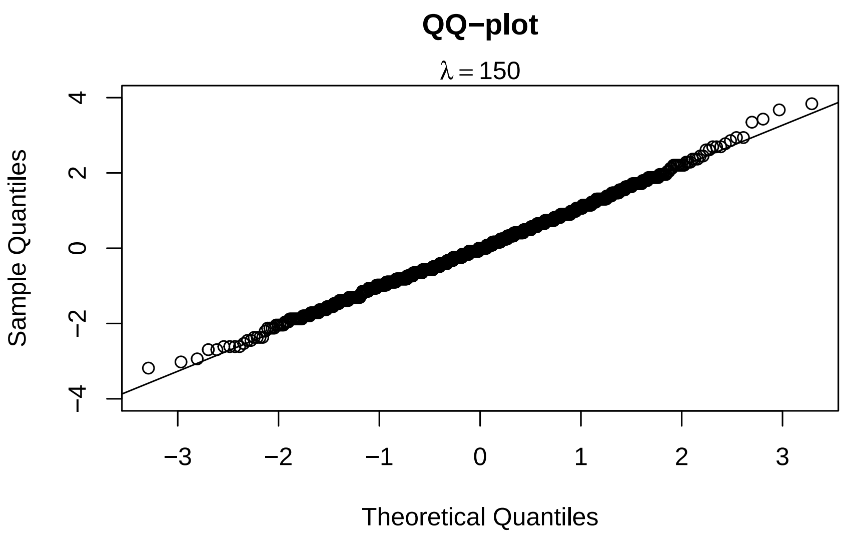
then Z is approximately standard normal as λ gets large.

A simple simulation:

```
R> simpois <- function(n, m) {
+     z <- (rpois(n, lambda = m) - m) / sqrt(m)
+     qqnorm(z, ylim = c(-4, 4), main = "QQ-plot")
+     qqline(z)
+     mtext(bquote(lambda == .(m),), 3)
+ }
```

E.g., for $\lambda = 150$,

```
R> simpois(1000, 150)
```



A simple movie:

```
R> for(m in seq(1, 100, 3)) {
+     simpois(1000, m)
+     Sys.sleep(1)
+ }
```

2.2 Inverse transform method

Let $F^{-1}(u) = \inf\{x : F(x) \geq u\}$ be the quantile function. Then (e.g., `http://en.wikipedia.org/wiki/Cumulative_distribution_function#Inverse`) $F^{-1}(u) \leq x$ if and only if $F(x) \geq u$. Hence, if U is uniformly distributed on the unit interval,

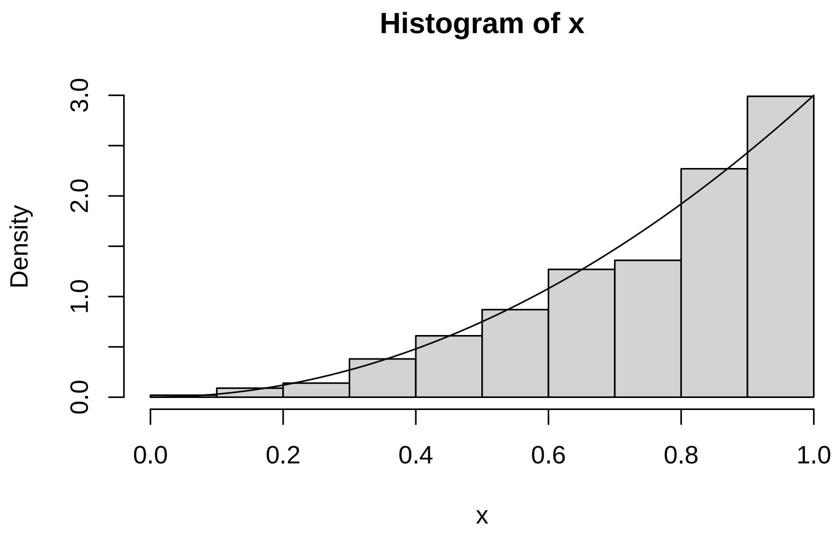
$$\mathbb{P}(F^{-1}(U) \leq x) = \mathbb{P}(U \leq F(x)) = F(x),$$

i.e., $F^{-1}(U) \sim F$.

Continuous/increasing case. Simple, as we can uniquely solve $F(x) = u$ (analytically or numerically).

Suppose e.g $f(x) = 3x^2$, $0 < x < 1$. Then $F(x) = x^3$ and solving $x^3 = u$ gives $F^{-1}(u) = u^{1/3}$. Thus, if U is standard uniform, $U^{1/3} \sim F$. To illustrate, generate a sample, draw its (density) histogram, and add the true density:

```
R> n <- 1000
R> x <- runif(n) ^ (1/3)
R> hist(x, probability = TRUE)
R> y <- seq(0, 1, .01)
R> lines(y, 3 * y ^ 2)
```



Consider the exponential distribution. This has

$$f(t) = \lambda e^{-\lambda t}, \qquad F(t) = 1 - e^{-\lambda t}, \qquad t \geq 0.$$

Solving $1 - e^{-\lambda t} = u$ gives

$$F^{-1}(u) = -\log(1-u)/\lambda$$

Thus, we could use

```
R> myrexp <- function(n, rate)
+     log(1 - runif(n)) / rate
```

Discrete case. Harder. Let x_i be the values assumed with positive probability. Then (draw a picture) the quantile function is

$$F^{-1}(u) = x_i, \qquad F(x_{i-1}) < u \leq F(x_i).$$

Bernoulli distribution: concentrated on 0 and 1 with probabilities $1-p$ and p, respectively. Thus, $F^{-1}(u) = 1$ for $u > 1-p$ and 0 otherwise. We can use

```
R> myrbernoulli <- function(n, prob)
+     as.numeric(runif(n) > (1 - prob))
```

(or equivalently, as $1-U$ has the same distribution as U, we could use the indicator of $U \leq p$.)

The geometric distribution has pmf $f(x) = pq^x$, $x = 0, 1, 2, \ldots$. The cdf at a discontinuity point is thus

$$F(x) = \sum_{i=0}^{x} pq^i = p\frac{1 - q^{x+1}}{1 - q} = 1 - q^{x+1}.$$

For sampling we need to solve

$$1 - q^x < u \leq 1 - q^{x+1}$$

or equivalently,

$$x < \log(1 - u)/\log(q) \leq x + 1.$$

We can use

```
R> myrgeom <- function(n, prob)
+     ceiling(log(1 - runif(n)) / log(1 - prob)) - 1
```

2.3 Acceptance-rejection method

Want to simulate X from f, can simulate Y from g where $f(t) \leq cg(t)$ for all t where $f(t) > 0$. Can use the following approach: for each random variate required,

1. Draw y from g
2. Draw u from $U_{0,1}$
3. If $u < f(y)/(cg(y))$ accept and deliver $x = y$; otherwise, reject y and restart.

This gives

$$\mathbb{P}(\text{accept}|y) = \mathbb{P}\left(U < \frac{f(y)}{cg(y)}\right) = \frac{f(y)}{cg(y)}$$

and hence

$$\mathbb{P}(\text{accept}) = \int \mathbb{P}(\text{accept}|y)dG(y) = \int \frac{f(y)}{cg(y)}g(y)dy = \frac{1}{c}$$

indicating that ideally c should be "small" (as close to 1 as possible). To see that X has the right distribution:

$$\mathbb{P}(X = x|\text{accept}) = \frac{\mathbb{P}(\text{accept}|x)\mathbb{P}(Y = x)}{\mathbb{P}(\text{accept})} = \frac{(f(x)/(cg(x)))g(x)}{1/c} = f(x)$$

Illustration for beta$(2, 2)$ which has $f(x) = 6x(1 - x)$, $0 < x < 1$. Take g as the density of the uniform distribution on $[0, 1$, then $f(x)/g(x) = 6x(1 - x) < 6$, so we can take $c = 6$, and accept if

$$\frac{f(y)}{cg(y)} = \frac{6y(1 - y)}{6} = y(1 - y) > u.$$

We can use e.g.

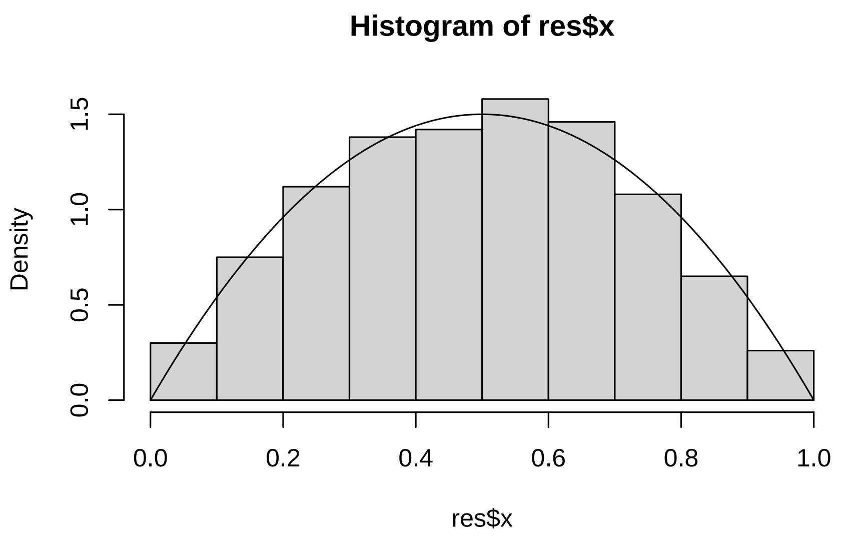
```
R> myrbeta22 <- function(n) {
+     k <- 0
+     j <- 0
+     x <- numeric(n)
+     while(k < n) {
+         u <- runif(1)
+         j <- j + 1
+         y <- runif(1)    # random variate from g
+         if(y * (1 - y) > u) {
+             k <- k + 1
+             x[k] <- y
+         }
+     }
+     list(x = x, num_of_iterations = j)
+ }
R> res <- myrbeta22(1000)
R> res$num_of_iterations
```

```
[1] 5916
```

Note that we could use a much smaller c!

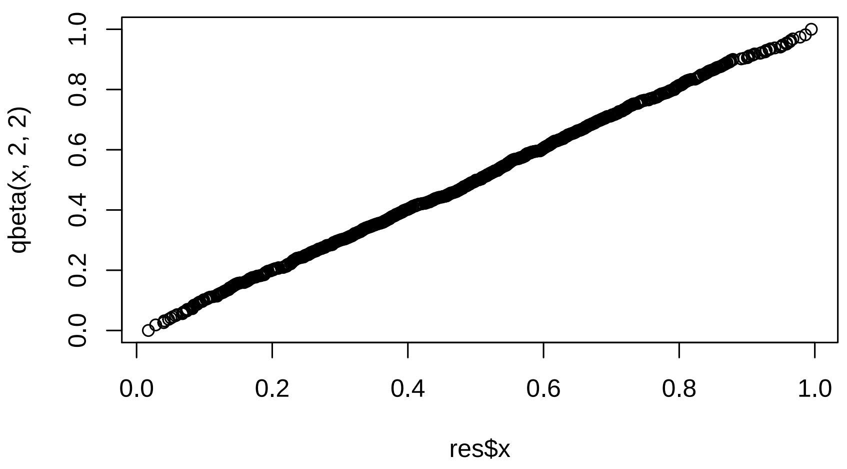
To illustrate:

```
R> hist(res$x, probability = TRUE)
R> x <- seq(0, 1, by = 0.001)
R> lines(x, dbeta(x, 2, 2))
```



Or using QQ-plots:

```
R> qqplot(res$x, qbeta(x, 2, 2))
```



2.4 Sums and Mixtures

A random variable X is a discrete mixture if its distribution function is of the form

$$F_X(x) = \sum_i \theta_i F_{X_i}(x)$$

for suitable random variables X_i and mixing probabilities (weights) θ_i which are positive and sum to one.

A random variable X is a continuous mixture if its distribution function is of the form

$$F_X(x) = \int F_{X|Y=y}(x) f_Y(y) dy$$

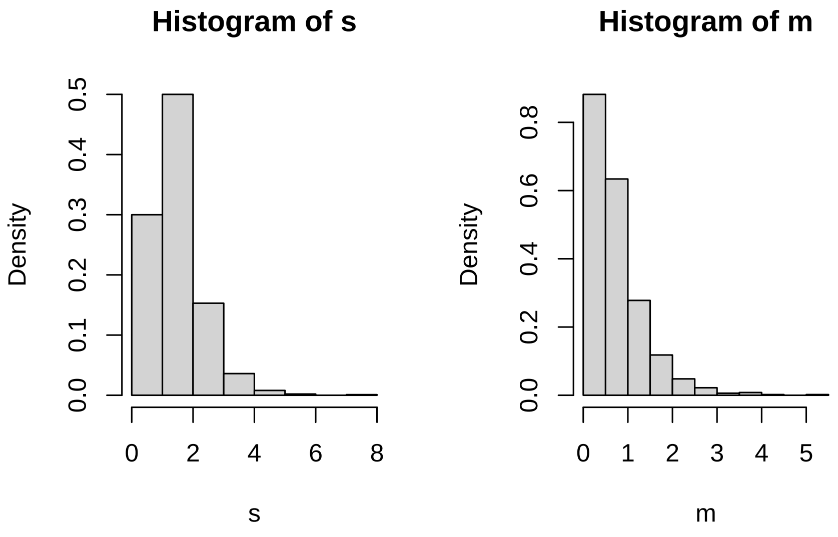
for a suitable family of conditional distribution functions $F_{X|Y=y}$ and non-negative weighting function f_Y integrating to one.

Note that mixtures are very different from convolutions, i.e., the distributions of sums of varibles drawn independently from the distributions.

Example: take $X_1 \sim \text{gamma}(2, 2)$ and $X_2 \sim \text{gamma}(2, 4)$ and compare the distribution of the sum to the discrete mixture with weights 1/2.

```
R> n <- 1000
R> x1 <- rgamma(n, 2, 2)
R> x2 <- rgamma(n, 2, 4)
R> ## Convolution:
R> s <- x1 + x2
R> ## Mixture:
R> u <- runif(n)
R> k <- u > 0.5
R> m <- k * x1 + (1 - k) * x2
R> ## Compare:
R> op <- par(mfcol = c(1, 2))
```

```
R> hist(s, probability = TRUE)
R> hist(m, probability = TRUE)
R> par(op)
```



To simulate mixtures:

1. Draw the mixing variable Y from the mixing distribution
2. Draw X from the respective conditional distribution.

Note that the density of a discrete mixture is

$$\sum_i \theta_i f_{X_i}(x).$$

2.5 Stochastic Processes

Homogeneous Poisson process: a point process where

1. The number of points in a set is Poisson with rate proportional to the size of the set.
2. The numbers of points in non-overlapping sets are independent of each other.

Simulating a homogeneous Poisson process, method A: to simulate a Poisson process with parameter λ on $[0, T]$,

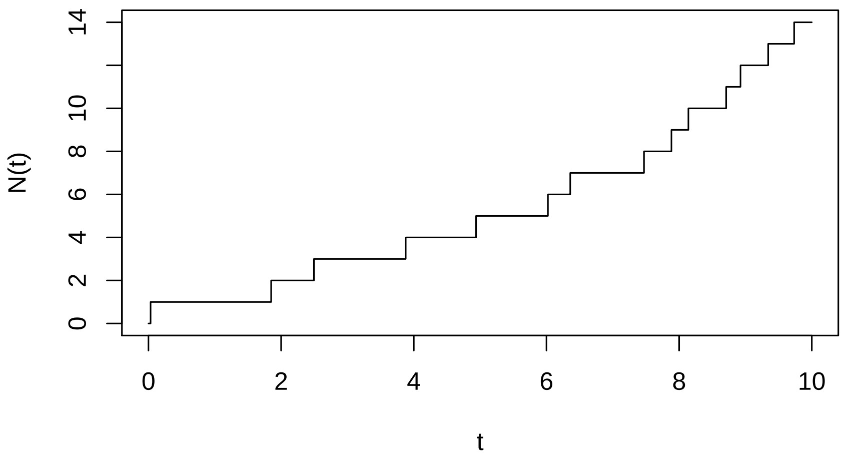
1. Generate N as a Poisson pseudorandom number with parameter λT,
2. Generate N independent uniform pseudorandom numbers on the interval $[0, T]$.

```
R> rpp1 <- function(T, lambda)
+     sort(runif(rpois(1, T * lambda), max = T))
R> x <- rpp1(10, 1.5)
R> x
```

```
 [1] 0.03206024 1.84975191 2.49488275 3.87812095 4.93893431 6.02180251
 [7] 6.35851573 7.47033746 7.88404510 8.13952005 8.70910381 8.92513295
[13] 9.34249063 9.73371583
```

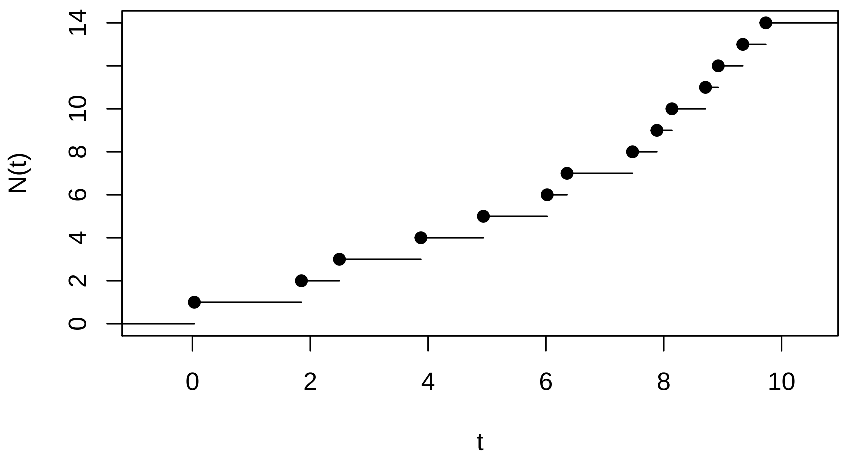
To plot, use e.g.

```
R> plot(c(0, x, 10), c(0, seq_along(x), length(x)), type = "s",
+      xlab = "t", ylab = "N(t)")
```



Perhaps better,

```
R> plot(stepfun(x, c(0, seq_along(x))), verticals = FALSE, pch = 19,
+      xlab = "t", ylab = "N(t)", main = "")
```



Simulating a homogeneous Poisson process, method B: one can show that the points of a homogeneous Poisson process with parameter λ are separated by independent exponentially distributed random variables with mean $1/\lambda$ ("interarrival times"). Thus, to simulate the first n points:

```
R> rpp2 <- function(n, lambda)
+     cumsum(rexp(n, rate = lambda))
R> x <- rpp2(20, 1.5)
R> x
```

```
 [1]  0.1762927  0.6552998  1.2091023  1.6871384  1.8608376  2.6358141
 [7]  2.8210504  3.8153332  4.8743657  5.3299389  6.0080082  6.1169626
[13]  6.4581454  6.6621908  7.6645175  8.8340803  8.8668155  9.2303129
[19]  9.8353580 11.0987723
```

2.6 MC estimation

Basic idea. To estimate

$$\theta = \mathbb{E}(g(X)) = \int g(x)f(x)\,dx$$

draw a sample $x_1, \ldots, x_n$ from f and use

$$\hat{\theta} = \frac{1}{n}\sum_{i=1}^{n} g(x_i)$$

If the X_i are drawn independently, the variance of this is $\mathrm{var}(\hat{\theta}) = \mathrm{var}(g(X))/n$. Hence, the standard error can be estimated by $\mathrm{sd}(g(x))/\sqrt{n}$. (More generally, repeatedly draw MC estimates, and use the standard deviation of these.)

Example: to compute an MC estimate of

$$\theta = \int_0^1 e^{-x} dx,$$

use

```
R> n <- 1000
R> x <- runif(n)
R> theta_hat <- mean(exp(-x))
R> c(theta_hat, 1 - exp(-1))
```

```
[1] 0.6309598 0.6321206
```

and estimate the standard error as

```
R> sd(exp(-x)) / sqrt(n)
```

```
[1] 0.005772234
```

A more refined example. MC estimation of

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt = \frac{\theta}{\sqrt{2\pi}} + \frac{1}{2}, \qquad \theta = \int_0^x e^{-t^2/2} dt$$

for $x > 0$.

Suppose we prefer drawing from $U_{0,1}$ rather than $U_{0,x}$ so that we can use the same variates for different x, and substitute $y = t/x$. Then

$$\theta = \int_0^1 x e^{-(xy)^2/2} dy.$$

We can use

```
R> mypnorm <- function(x, n = 10000) {
+     u <- runif(n)
+     p <- numeric(length(x))
+     for(i in seq_along(x)) {
+         g <- x[i] * exp(-(u * x[i])^2 / 2)
+         p[i] <- mean(g) / sqrt(2 * pi) + 0.5
+     }
+     p
+ }
```

giving e.g.

```
R> x <- seq(.1, 2.5, length.out = 10)
R> p <- mypnorm(x)
R> p

 [1] 0.5398275 0.6430499 0.7366659 0.8157525 0.8779966 0.9236514 0.9548821
 [8] 0.9748367 0.9867825 0.9935247

R> p - pnorm(x)

 [1] -3.459796e-07 -1.629346e-05 -7.608520e-05 -1.873916e-04 -3.308793e-04
 [6] -4.673034e-04 -5.524033e-04 -5.524347e-04 -4.540285e-04 -2.656724e-04
```

Importance sampling. Suppose X is a random variable with density $f(x)$ and $f(x) > 0$ where $g(x) > 0$, and let $Y = g(X)/f(X)$. Then

$$\mathbb{E}(Y) = \int \frac{g(x)}{f(x)} f(x) dx = \int g(x) dx.$$

Thus, to estimate $\theta = \int g(x)dx$, use

$$\hat{\theta} = \frac{1}{n} \sum_{i=1}^{n} y_i = \frac{1}{n} \sum_{i=1}^{n} \frac{g(x_i)}{f(x_i)}.$$

Note that

$$\mathrm{var}(\hat{\theta}) = \mathrm{var}(Y))/n,$$

so the estimate will be good if the variance is small, i.e., if f is close to g.

Antithetic variates. Note that in general

$$\mathrm{var}(U+V) = \mathrm{var}(U) + \mathrm{var}(V) + 2\mathrm{cov}(U,V)$$

so maybe we could reduce the variance of MC estimates even more if we used (pairs of) negatively correlated variates?

Idea: If the X_i are simulated via the inverse transform method, $X_i = F_X^{-1}(U_i)$. But $1-U$ has the same distribution as U and is negatively correlated with U. But is $F_X^{-1}(1-U_i)$ negatively correlated with X_i?

One can show: if $g = g(x_1, \ldots, x_k)$ is monotone, then

$$Y = g(F_X^{-1}(U_1), \ldots, F_X^{-1}(U_k)) \qquad Y' = g((F_X^{-1}(1-U_1), \ldots, F_X^{-1}(1-U_k))$$

are negatively correlated.

For MC estimation: generate $n/2$ replicates of Y_j and Y_j' using the same $U_1^{(j)}, \ldots, U_k^j$, and use

$$\hat{\theta} = \frac{2}{n} \sum_{i=1}^{n/2} \frac{Y_j + Y_j'}{2}.$$

Note: this requires $nk/2$ instead of nk uniform variates, and reduces estimation variance by using antithetic variables.

To illustrate, continue MC estimation of $\Phi(x)$. We had

$$\theta = \mathbb{E}_U(xe^{-(xU)^2/2}),$$

for U standard uniform. When restricting to $x > 0$, $g(u) = xe^{-(ux)^2/2}$ is monotone. Hence, we can use

$$Y_j = xe^{-(xU_j)^2/2}, \qquad Y_j' = xe^{-(x(1-U_j))^2/2},$$

giving

$$\hat{\theta} = \frac{1}{n/2} \sum_{i=1}^{n/2} \frac{xe^{-(xu_j)^2/2} + xe^{-(x(1-u_j))^2/2}}{2}.$$

We provide a function which has a flag for toggling the use of anithetic sampling

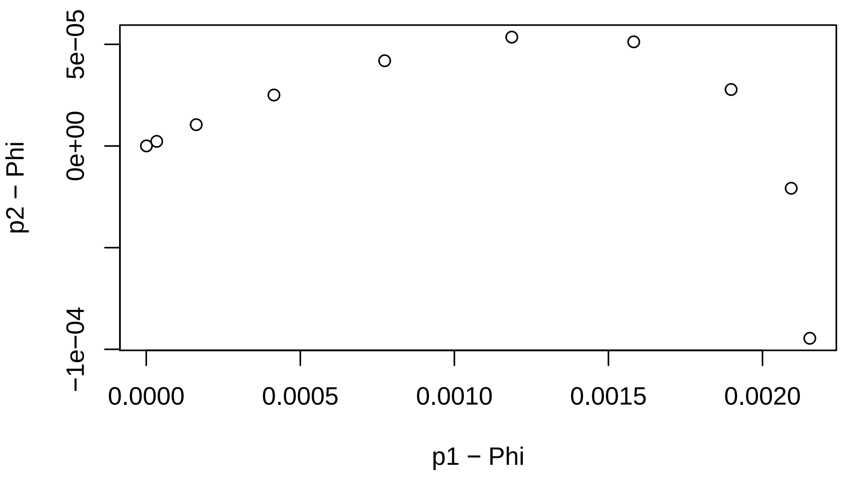
```
R> mypnorm2 <- function(x, n = 10000, antithetic = TRUE) {
+     u <- runif(n / 2)
+     v <- if(!antithetic) runif(n / 2) else 1 - u
+     u <- c(u, v)
+     p <- numeric(length(x))
+     for(i in seq_along(x)) {
+         g <- x[i] * exp(-(u * x[i])^2 / 2)
+         p[i] <- mean(g) / sqrt(2 * pi) + 0.5
+     }
+     p
+ }
```

and perform the following MC experiment:

```
R> x <- seq(.1, 2.5, length.out = 10)
R> Phi <- pnorm(x)
R> set.seed(123)
R> system.time(p1 <- mypnorm2(x, antithetic = FALSE))

   user  system elapsed
  0.013   0.000   0.013

R> set.seed(123)
R> system.time(p2 <- mypnorm2(x))

   user  system elapsed
  0.001   0.000   0.002

R> print(round(cbind(x, p1, p2, Phi), 5))

            x      p1      p2     Phi
 [1,] 0.10000 0.53983 0.53983 0.53983
 [2,] 0.36667 0.64310 0.64307 0.64307
 [3,] 0.63333 0.73690 0.73675 0.73674
 [4,] 0.90000 0.81635 0.81596 0.81594
 [5,] 1.16667 0.87910 0.87837 0.87833
 [6,] 1.43333 0.92530 0.92417 0.92412
 [7,] 1.70000 0.95702 0.95549 0.95543
 [8,] 1.96667 0.97729 0.97542 0.97539
 [9,] 2.23333 0.98933 0.98722 0.98724
[10,] 2.50000 0.99594 0.99370 0.99379
```

Graphically:

```
R> plot(p1 - Phi, p2 - Phi)
```



3 Data Management and Exploration

3.1 German Credit Data

This is a well known data set in statistical and machine learning donated to the StatLog project by Hans Hofmann (Deparment of Statistics and Econometrics, University of Hamburg) and available from the UCI (University of California at Irvine) Machine Learning Repository at `http://archive.ics.uci.edu/ml/machine-learning-databases/statlog/german/`, in fact going back to Fahrmeir, Hamerle & Tutz (1984), "Multivariate statistische Verfahren" (1st edition), see `http://www.stat.uni-muenchen.de/service/datenarchiv/kredit/kredit_e.html`. It contains, for 1000 customers/credits, measurements on 20 variables and the classification of the customer into good and bad as far as credit risk is concerned. We use the data set for illustrating data management and exploration in R.

The data (file 'german.data') and "documentation" (file "german.doc'), which is rather poor and in fact mostly contains the codes used for the categorical variables, are available from the web site.

We start by inspecting the data and documentation files.

3.1.1 Reading In and Setting Up Data Sets

We can read the data into R via

```
R> german <- read.table("german.data", header = FALSE, stringsAsFactors = TRUE)
```

This returns a *data frame*, a special list structure whose elements all have the same length which is used for the "usual" rectangular case-by-variable data frame/table layout.

```
R> class(german)

[1] "data.frame"

R> typeof(german)

[1] "list"

R> length(german)

[1] 21
```

The elements of the list correspond to the variables, which can be subscripted using `[`, `[[` and `$` in the usual way. One can also use 2-index subscripting as for matrices to select cases or cases and variables:

```
R> german[1 : 2, 3 : 5]

   V3  V4   V5
1 A34 A43 1169
2 A32 A43 5951
```

Data frames may look like matrices, but are fundamentally different, as variables may be of different "kind". In fact,

```
R> sapply(german, class)

       V1        V2        V3        V4        V5        V6        V7        V8
 "factor" "integer"  "factor"  "factor" "integer"  "factor"  "factor" "integer"
       V9       V10       V11       V12       V13       V14       V15       V16
 "factor"  "factor" "integer"  "factor" "integer"  "factor"  "factor" "integer"
      V17       V18       V19       V20       V21
 "factor" "integer"  "factor"  "factor" "integer"
```

shows many variables of class `"factor"`. Factors are another special data structure used by R for representing categorical data: class/creator `factor` for nominal data, class/creator `ordered` for ordinal data.

Discuss `read.table` details, in particular auto-conversion and headers.

Discuss other functionality for importing data into R: variants of `read.table` such as `read.csv`; functions for reading in data sets from SPSS, SAS and so on in package **foreign**. Several packages provide functions to read in data from Excel; alternatively, one can save these to CSV and use `read.csv`.

We add variable names (see the documentation):

```
R> names(german) <-
+     c("Status_of_checking_account",
+       "Duration",                        # (in months)
+       "History",                         # Credit history
+       "Purpose",
+       "Amount",                          # Credit amount
+       "Savings",                         # Savings account/bonds
+       "Employment_since",                # Present employment since
+       "Installment_rate",                # (in percentage of disposable
+                                          # income)
+       "Status_and_sex",                  # Personal status and sex
+       "Other_debtors_or_guarantors",
+       "Residence_since",                 # Present residence since
+       "Property",
+       "Age",                             # (in years)
+       "Other_installment_plans",
+       "Housing",
+       "N_of_credits",                    # Number of existing credits at
+                                          # this bank
+       "Job",                             # Employment level
+       "N_of_liables",                    # Number of people being liable
+                                          # to provide maintenance for
+       "Phone",
+       "Foreign",                         # Foreign worker?
+       "Class"                            # Credit quality
+       )
```

Having variable names makes it possible to use `subset` for simple database type queries:

```
R> subset(german, Age > 60 & Amount > 10000)

    Status_of_checking_account Duration History Purpose Amount Savings
374                        A14       60     A34     A40  13756     A65
```

```
918                        A11         6      A32     A40  14896     A61
    Employment_since Installment_rate Status_and_sex
374              A75                2           A93
918              A75                1           A93
    Other_debtors_or_guarantors Residence_since Property Age
374                        A101               4     A124  63
918                        A101               4     A124  68
    Other_installment_plans Housing N_of_credits  Job N_of_liables Phone
374                    A141    A153            1 A174            1  A192
918                    A141    A152            1 A174            1  A192
    Foreign Class
374    A201     1
918    A201     2
```

Clearly, we also need to transform (or recode) some of the variables; in particular, `Class` came out integer but should really be a factor with levels 'good' and 'bad'. We can either modify explicitly:

```
R> german$Class <- factor(german$Class, labels = c("good", "bad"))
```

or more high-level via `transform`:

```
R> german <- transform(german,
+                      Class = factor(Class, labels = c("good", "bad")))
```

What is the right scale for the measurements of variable 'Job'? Partially intended as ordinal (ordered according to skill level), but then "unemployed" does not fit in.

What is the right scale for the measurements of the variable indicating the status of an existing checking account? Ordinal only if the credit customer (obligor) has a checking account, hence, use nomimal. To recode:

```
R> head(german$Status_of_checking_account)

[1] A11 A12 A14 A11 A11 A14
Levels: A11 A12 A13 A14

R> german <-
+     transform(german,
+               Status_of_checking_account =
+               factor(Status_of_checking_account,
+                      levels = c("A11", "A12", "A13", "A14"),
+                      labels = c("<0", "[0,200)", ">=200", "none")))
R> head(german$Status_of_checking_account)

[1] <0      [0,200) none    <0      <0      none
Levels: <0 [0,200) >=200 none
```

Or, modify the levels directly:

```
R> levels(german$Status_of_checking_account) <-
+     c("neg", "p_lo", "p_hi", "none")
R> head(german$Status_of_checking_account)
```

```
[1] neg  p_lo none neg  neg  none
Levels: neg p_lo p_hi none
```

The variable measuring the duration of the current employment is clearly ordinal (unemployed corresponds to a duration of 0), so we could recode as follows:

```
R> head(german$Employment_since)
```

```
[1] A75 A73 A74 A74 A73 A73
Levels: A71 A72 A73 A74 A75
```

```
R> german <-
+     transform(german,
+               Employment_since =
+               ordered(Employment_since,
+                       levels = sprintf("A7%d", 1 : 5),
+                       labels = c("0", "(0,1)", "[1,4)",
+                                  "[4,7)", "[7,Inf)")))
R> head(german$Employment_since)
```

```
[1] [7,Inf) [1,4)   [4,7)   [4,7)   [1,4)   [1,4)
Levels: 0 < (0,1) < [1,4) < [4,7) < [7,Inf)
```

Finally, we recode variable 'Purpose' (note that 'A47' does not occur in the data):

```
R> german <-
+     transform(german,
+               Purpose =
+               factor(Purpose,
+                      levels = sprintf("A4%d", 0 : 10),
+                      labels = c("car/new", "car/used", "furn/equip",
+                                 "radio/tv", "appliance", "repairs",
+                                 "education", "vacation", "retraining",
+                                 "business", "others")))
```

3.1.2 Summarizing Individual Variables

We can use `summary` to summarize the data set:

```
R> summary(german)
```

```
 Status_of_checking_account    Duration    History        Purpose
 neg :274                   Min.   : 4.0   A30: 40   radio/tv  :280
 p_lo:269                   1st Qu.:12.0   A31: 49   car/new   :234
 p_hi: 63                   Median :18.0   A32:530   furn/equip:181
 none:394                   Mean   :20.9   A33: 88   car/used  :103
                            3rd Qu.:24.0   A34:293   business  : 97
                            Max.   :72.0             education : 50
                                                     (Other)   : 55
     Amount      Savings   Employment_since Installment_rate Status_and_sex
 Min.   :  250   A61:603   0      : 62      Min.   :1.000    A91: 50
 1st Qu.: 1366   A62:103   (0,1)  :172      1st Qu.:2.000    A92:310
 Median : 2320   A63: 63   [1,4)  :339      Median :3.000    A93:548
 Mean   : 3271   A64: 48   [4,7)  :174      Mean   :2.973    A94: 92
 3rd Qu.: 3972   A65:183   [7,Inf):253      3rd Qu.:4.000
 Max.   :18424                              Max.   :4.000

 Other_debtors_or_guarantors Residence_since Property        Age
 A101:907                    Min.   :1.000   A121:282   Min.   :19.00
 A102: 41                    1st Qu.:2.000   A122:232   1st Qu.:27.00
 A103: 52                    Median :3.000   A123:332   Median :33.00
                             Mean   :2.845   A124:154   Mean   :35.55
                             3rd Qu.:4.000              3rd Qu.:42.00
                             Max.   :4.000              Max.   :75.00

 Other_installment_plans Housing     N_of_credits     Job       N_of_liables
 A141:139                A151:179   Min.   :1.000   A171: 22   Min.   :1.000
 A142: 47                A152:713   1st Qu.:1.000   A172:200   1st Qu.:1.000
 A143:814                A153:108   Median :1.000   A173:630   Median :1.000
                                    Mean   :1.407   A174:148   Mean   :1.155
                                    3rd Qu.:2.000              3rd Qu.:1.000
                                    Max.   :4.000              Max.   :2.000

  Phone      Foreign     Class
 A191:596   A201:963   good:700
 A192:404   A202: 37   bad :300
```

We see that for numerical summaries, five-point summaries plus means are used for the numeric, and frequency tables (of the most frequent levels) for the categorical variables.

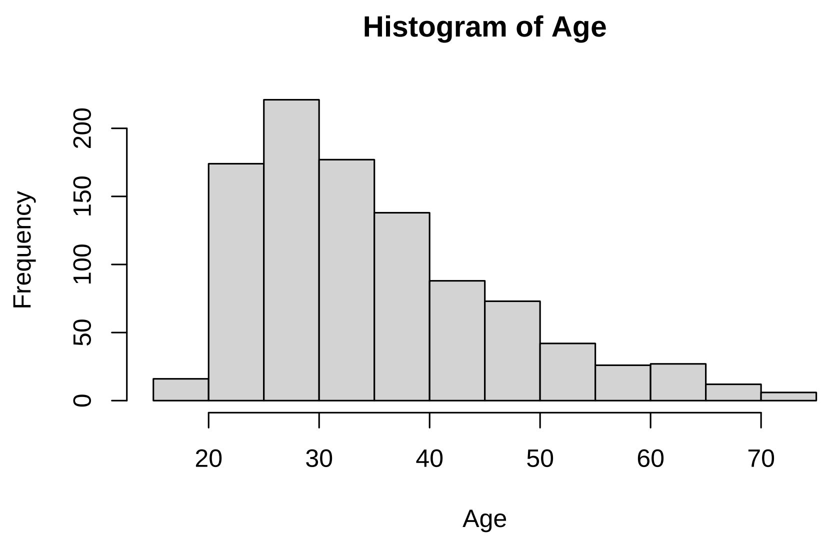
We can also use `summary` individually:

```
R> summary(german$Age)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  19.00   27.00   33.00   35.55   42.00   75.00

R> with(german, summary(Age))

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  19.00   27.00   33.00   35.55   42.00   75.00
```

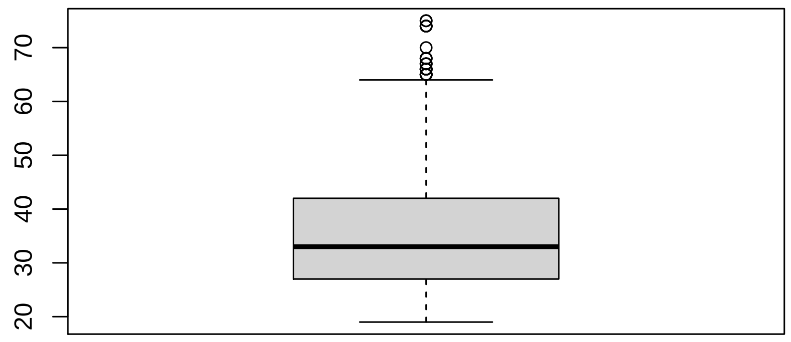
> To refer to the variables in a data frame, we can always use "direct access via `$`", or `with()`, or `attach()` (which we recommend not to use).

For graphical summaries of numeric variables we can use histograms or boxplots:

```
R> with(german, hist(Age))
```

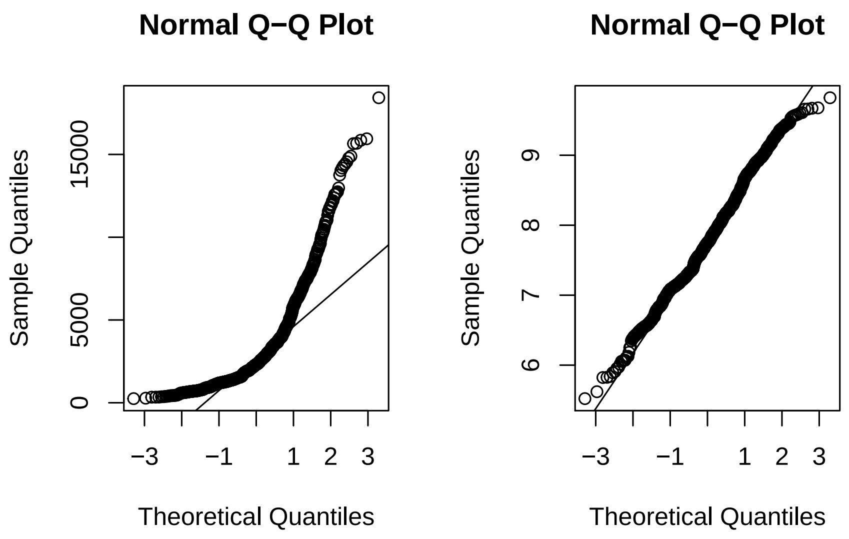


```
R> with(german, boxplot(Age))
```



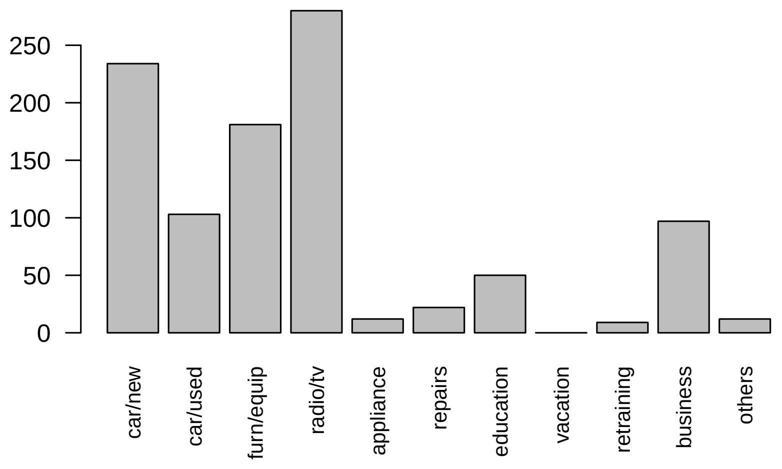
We can also use QQ-plots to inspect the distributions. Quite often, taking logs of positive variables substantially improves the fit by a normal distribution:

```
R> with(german, {
+     op <- par(mfcol = c(1, 2))
+     qqnorm(Amount)
+     qqline(Amount)
+     qqnorm(log(Amount))
+     qqline(log(Amount))
+     par(op)
+ })
```



For graphical summaries of categorical variables we can use barplots (statistical graphics scholars do not use pie charts):

```
R> with(german, plot(Purpose, las = 2, cex.names = 0.8))
```

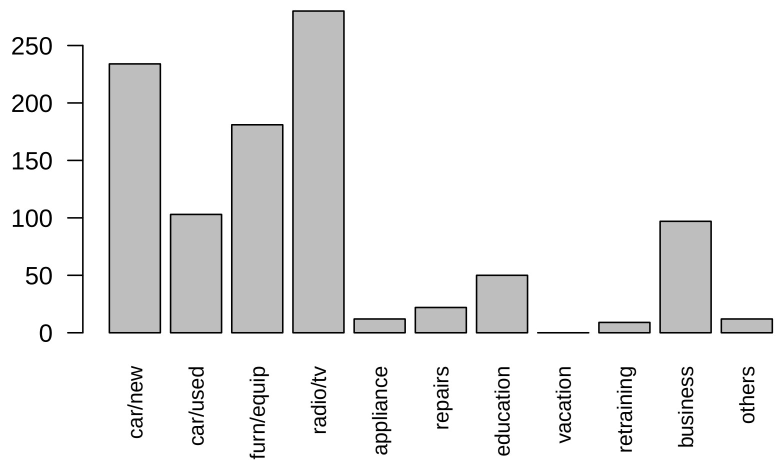


Or directly:

```
R> tab <- with(german, table(Purpose))
R> tab
```

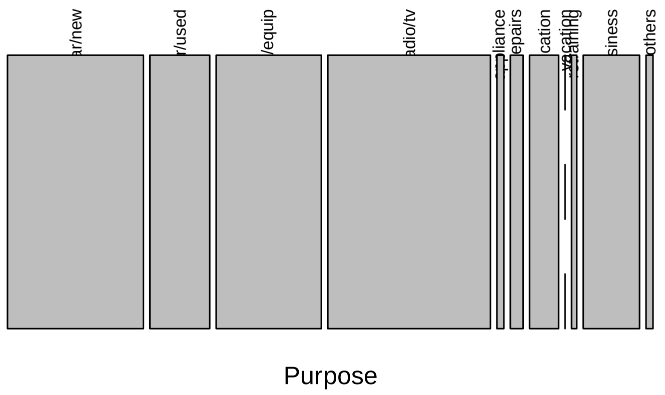
```
Purpose
   car/new   car/used furn/equip   radio/tv  appliance    repairs  education
       234        103        181        280         12         22         50
  vacation retraining   business     others
         0          9         97         12
```

```
R> barplot(tab, las = 2, cex.names = 0.8)
```



One can also use bars of constant height, resulting in simple *mosaic plots*:

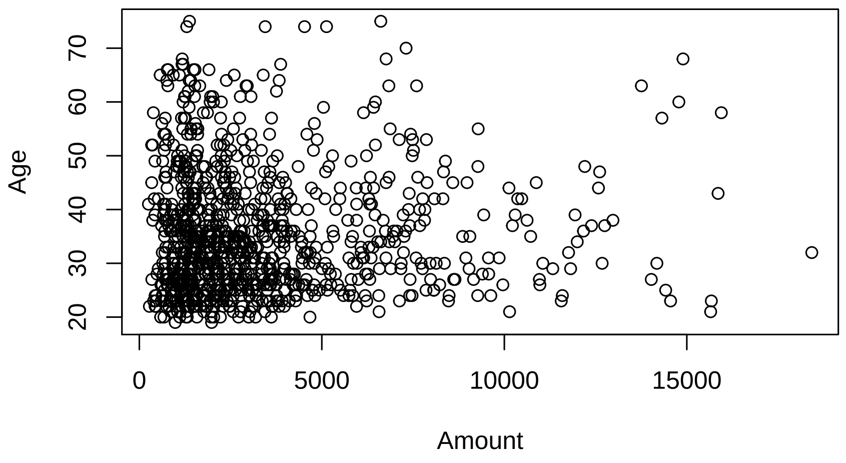
```
R> mosaicplot(tab, las = 2, main = "")
```



3.1.3 Summarizing Several Variables

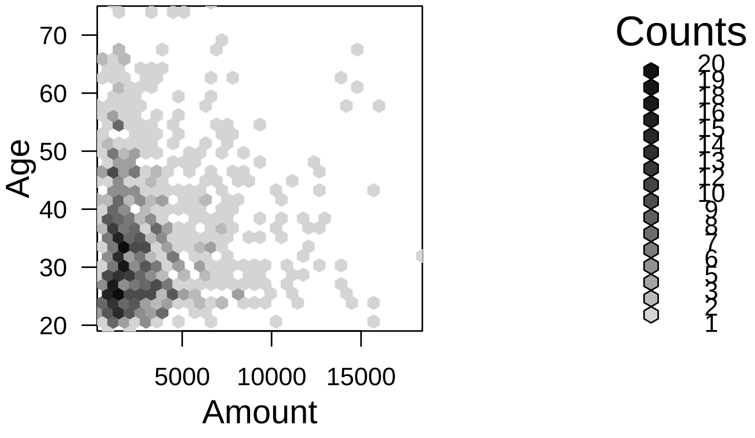
To summarize two numeric variables, we can use scatterplots:

```
R> with(german, plot(Amount, Age))
```



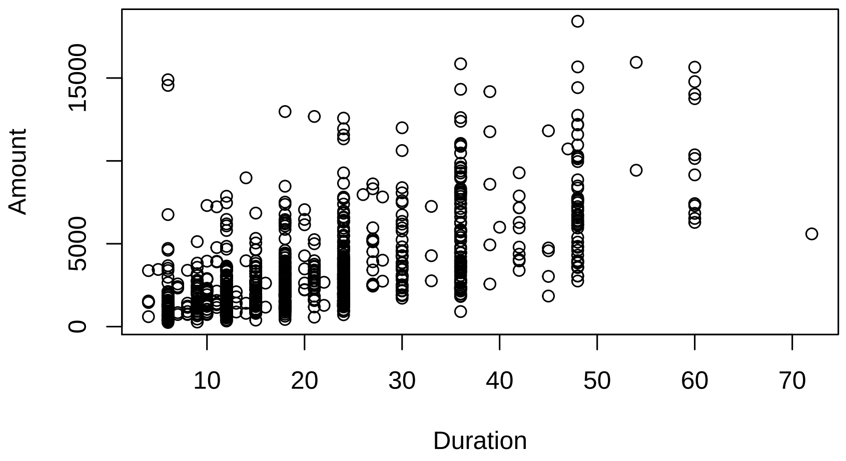
Or, perhaps better use hexagonal binning:

```
R> require("hexbin")
R> with(german, plot(hexbin(Amount, Age)))
```



Or, using the formula interface

```
R> plot(Amount ~ Duration, german)
```

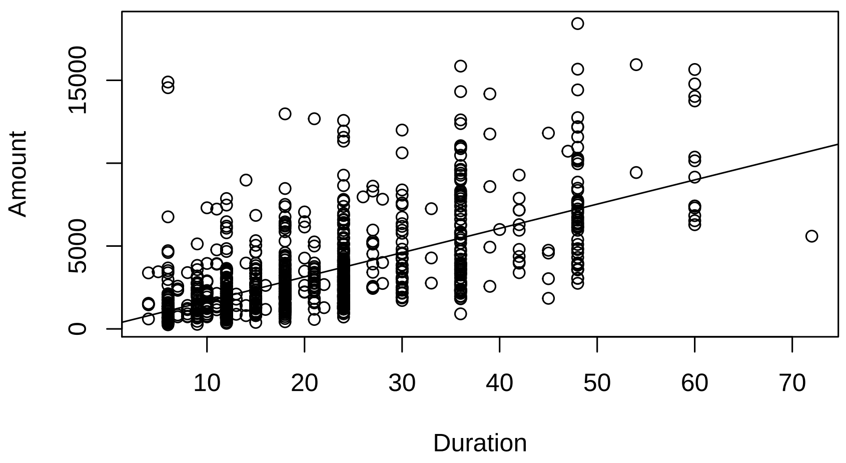


This model formula suggests treating 'Amount' as the dependent or response variable, and 'Duration' as the independent or explanatory variable.

> Exploration and inference have to appropriately take the roles of the variables (explanatory versus response) into account.

One can add a regression line to the previous scatterplot via

```
R> plot(Amount ~ Duration, german)
R> abline(lm(Amount ~ Duration, german))
```



> Modeling and plotting functions feature formula interfaces for analyzing relations between variables in a data frame.

To summarize two categorical variables, we use contingency tables and visualize these using mosaic plots.

```
R> tab1 <- with(german, table(Job, Class))
R> tab1
```

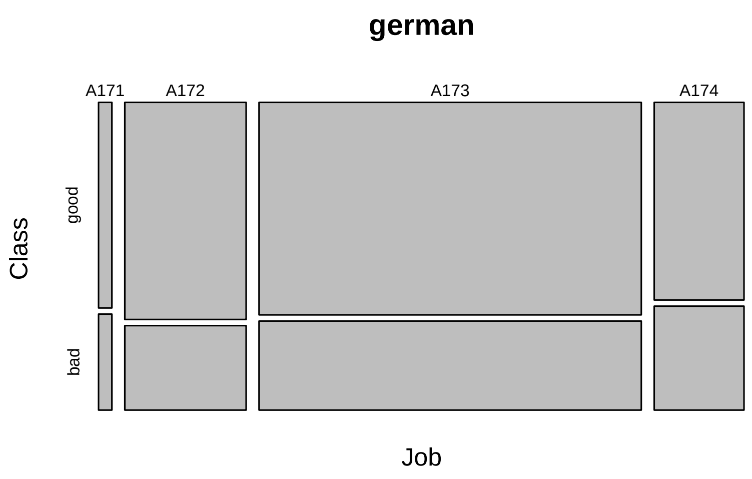
```
      Class
Job    good bad
  A171   15   7
  A172  144  56
  A173  444 186
  A174   97  51
```

```
R> tab2 <- xtabs(~ Job + Class, german)
R> tab2
```

```
      Class
Job    good bad
  A171   15   7
  A172  144  56
  A173  444 186
  A174   97  51
```

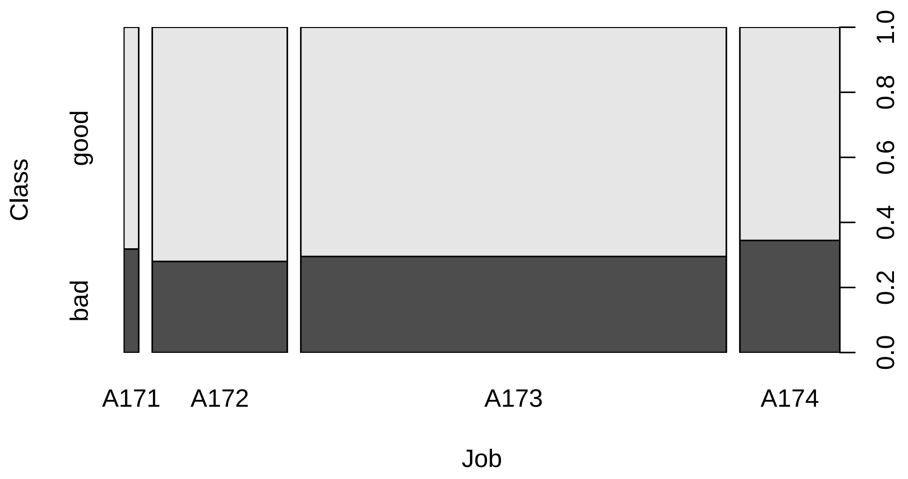
Note that even though we clearly think of 'Class' as the dependent variable, it appears on the RHS of the model formula for `xtabs` and `mosaicplot`:

```
R> mosaicplot(~ Job + Class, german)
```



To more clearly indicate 'Class' as the dependent variable, we can use

```
R> plot(Class ~ Job, german)
```



This gives *spine plots*, a special kind of mosaic plots.

When the response is numeric and the explanatory variable is categorical, we can summarize according to the levels of the factor, e.g.,

```
R> with(german, sapply(split(Amount, Purpose), median))
```

```
   car/new   car/used furn/equip   radio/tv  appliance    repairs  education 
    1980.0     4788.0     2578.0     1890.0     1249.0     1749.0     1884.5 
  vacation retraining   business     others 
        NA      932.0     3161.0     6948.0 
```

or more conveniently,

```
R> with(german, tapply(Amount, Purpose, median))
```

```
   car/new   car/used furn/equip   radio/tv  appliance    repairs  education 
    1980.0     4788.0     2578.0     1890.0     1249.0     1749.0     1884.5 
  vacation retraining   business     others 
        NA      932.0     3161.0     6948.0 
```

or, using complete summaries:

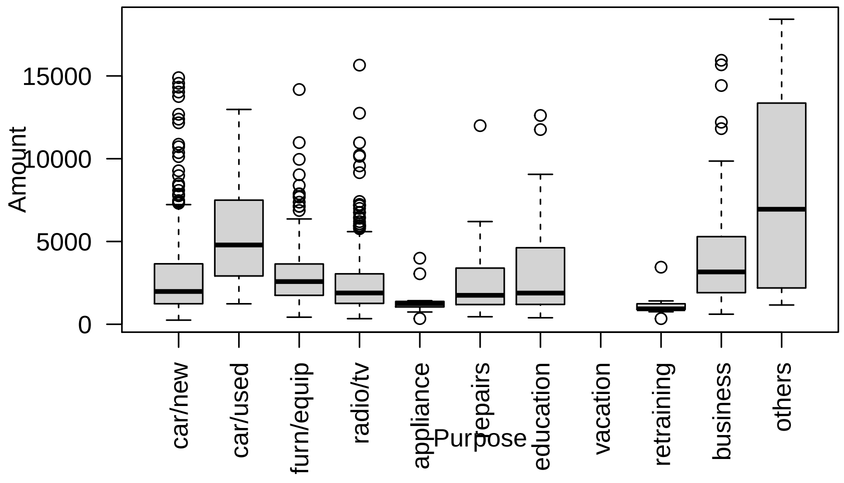
```
R> with(german, tapply(Amount, Purpose, summary))
```

```
$`car/new`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    250    1242    1980    3063    3632   14896 

$`car/used`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   1236    2917    4788    5370    7498   12976 
```

```
$`furn/equip`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    428    1747    2578    3067    3643   14179 

$`radio/tv`
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    338    1261    1890    2488    3038   15653 

$appliance
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    343    1131    1249    1498    1360    3990 

$repairs
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    454    1214    1749    2728    3203   11998 

$education
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    392    1198    1884    3180    4425   12612 

$vacation
NULL

$retraining
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    339     894     932    1206    1238    3447 

$business
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    609    1908    3161    4158    5293   15945 

$others
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   1164    2410    6948    8209   12649   18424 
```

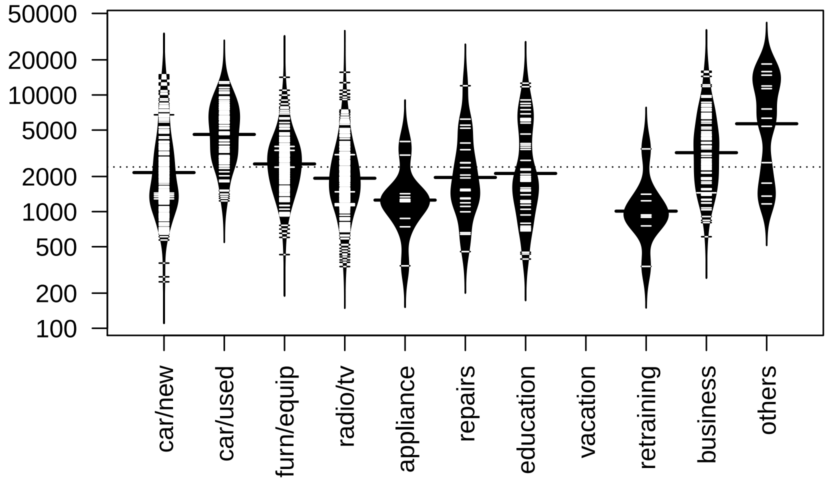
The groups can conveniently be compared using juxtaposed boxplots:

```
R> boxplot(Amount ~ Purpose, german, las = 2)
```



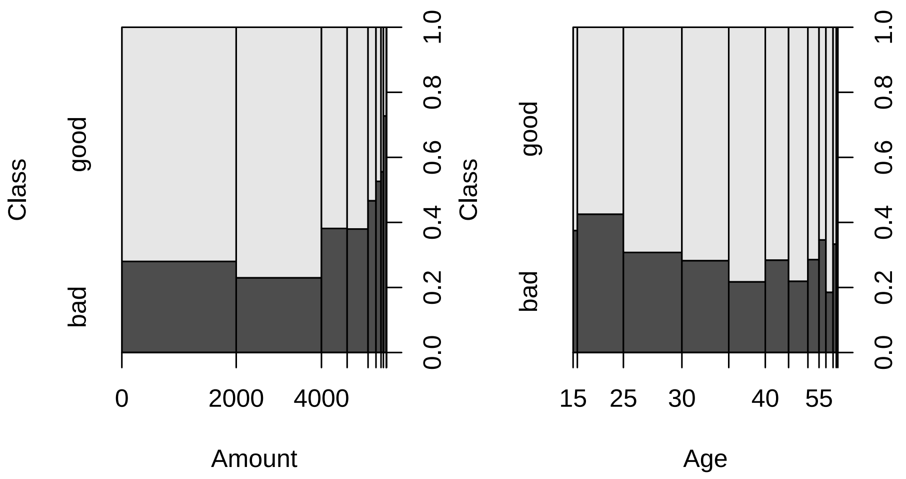
Or, perhaps better use violin plots or bean plots (which by default auto-selects log scaling, and here does so for the y axis):

```
R> require("beanplot")
R> beanplot(Amount ~ Purpose, german, las = 2)
```



Finally, when the response is categorical and the explanatory variable is numeric, we can again use spine plots for the visualization:

```
R> op <- par(mfcol = c(1, 2))
R> plot(Class ~ Amount, german)
R> plot(Class ~ Age, german)
R> par(op)
```

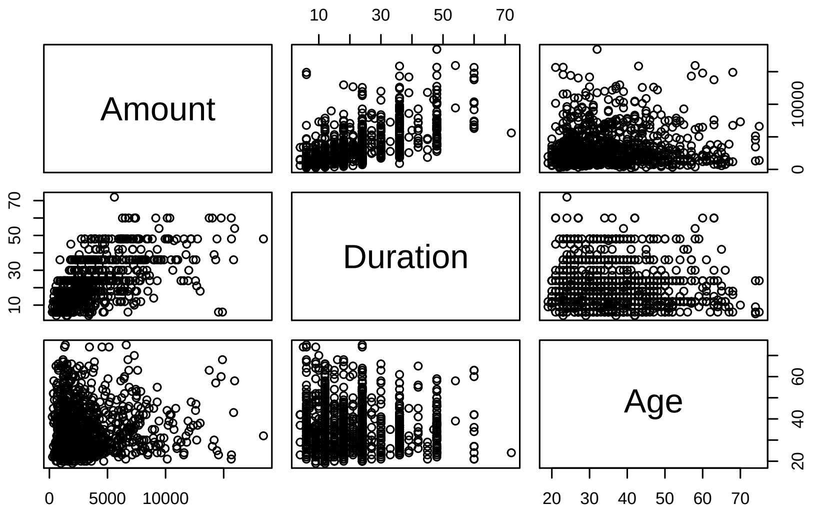


Note that the x-axis is not really scaled linearly!

Exploring the relation between more than two variables is even more challenging. We mention a few examples:

Scatterplot matrices for several numeric variables.

```
R> pairs(~ Amount + Duration + Age, german)
```



Conditioning plots for visualizing the relation between two numeric variables within the groups of one or more categorical variables:

```
R> coplot(Duration ~ Amount | Job + Purpose, german)
```

As higher dimensional contingency tables are hard to read “directly”, they are often flattened out:

```
R> ftable(Class ~ Job + Status_of_checking_account, german)

                                Class good bad
Job  Status_of_checking_account               
A171 neg                               2   4
     p_lo                              7   2
     p_hi                              3   1
     none                              3   0
A172 neg                              36  23
     p_lo                             38  19
     p_hi                              9   5
     none                             61   9
A173 neg                              76  96
     p_lo                             98  57
     p_hi                             31   6
     none                            239  27
A174 neg                              25  12
     p_lo                             21  27
     p_hi                              6   2
     none                             45  10
```

3.2 Statistical Graphics

Discuss distinction between base and grid graphics engines (and high level graphics systems such as lattice ("trellis") and ggplot2 ("Grammar of Graphics") built on top of grid). Here, we look at base graphics, an old graphics model in the pen and paper style.

In general, graphics functions draw on graphics *devices*: these can be opened via functions like `pdf()` (PDF output file), `X11()`, `windows()` and `quartz()` (screen devices on Unix, Windows, and MacOS X), and manipulated via `dev.new()`, `dev.prev()`, `dev.next()`, and `dev.copy()`.

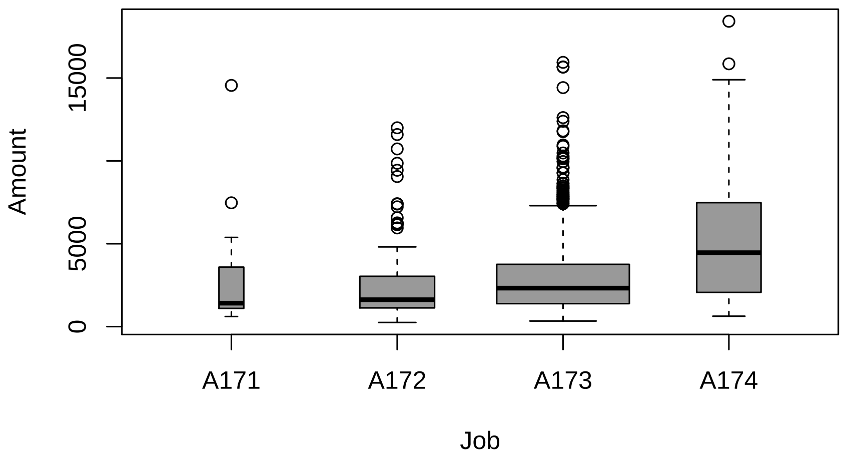
High-level graphics functionality allows indicating the kind of plot desired without worrying in detail where the ink goes. Such functions include, e.g., `hist()`, `boxplot()`, etc.

Such functions typically have additional arguments allowing to customize the plot. E.g., using a scatterplot smoother in a conditioning plot:

```
R> coplot(Duration ~ Amount | Job + Purpose, german, panel = panel.smooth)
```

Or, a boxplot where boxes are drawn with widths proportional to the square-roots of the number of observations in the groups, and colored in a medium gray:

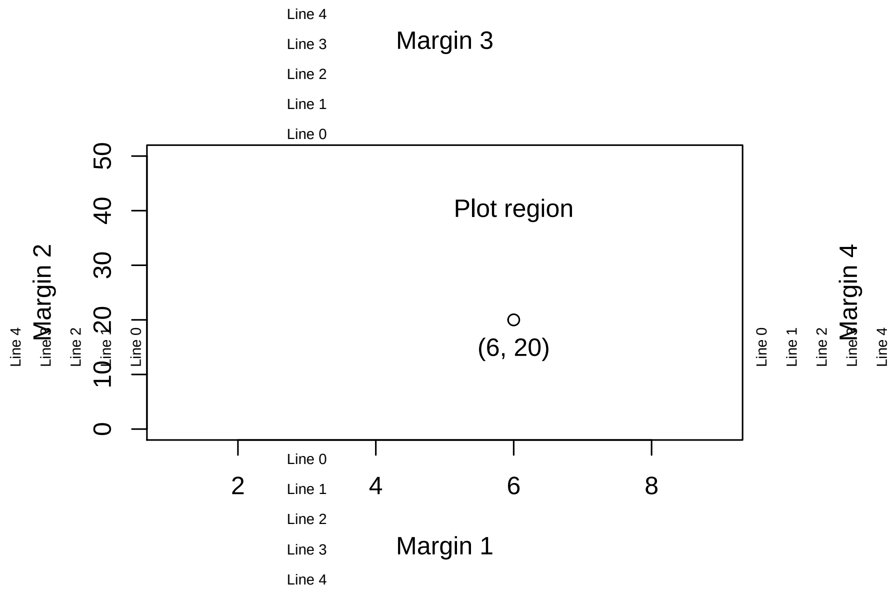
```
R> boxplot(Amount ~ Job, german, varwidth = TRUE, col = gray(0.6))
```



Low-level graphics functionality can also be used to add to existing plots.

We must first understand how R sets up the page for drawing on (using a single figure environment for simplicity):

```
R> op <- par(mar = c(5, 5, 5, 5) + 0.1)
R> plot(c(1, 9), c(0, 50), type = "n", xlab = "", ylab = "")
R> text(6, 40, "Plot region")
R> points(6, 20)
R> text(6, 20, "(6, 20)", adj = c(0.5, 2))
R> mtext(paste("Margin", 1 : 4), side = 1 : 4, line = 3)
R> mtext(paste("Line", 0 : 4), side = 1, line = 0 : 4, at =  3, cex = 0.6)
R> mtext(paste("Line", 0 : 4), side = 2, line = 0 : 4, at = 15, cex = 0.6)
R> mtext(paste("Line", 0 : 4), side = 3, line = 0 : 4, at =  3, cex = 0.6)
R> mtext(paste("Line", 0 : 4), side = 4, line = 0 : 4, at = 15, cex = 0.6)
```



To add components in the plot region, one can use e.g.

- `points()` and `lines()` to add points or line segments;
- `text()` to add text;
- `abline()` to add a complete line;
- `legend()` to draw a legend;
- `polygon()`, `segments()`, `arrows()`, `symbols()`

Make sure to read the help pages of `points()` and `lines()` to know about `pch` for plot characters and `type` for plot and line types!

To add component outside the plot region, there are

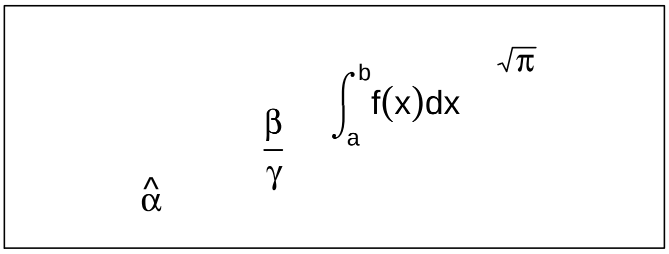
- `title()` to add a title, subtitle, x or y axis labels
- `mtext()` to draw text in the margins;
- `axis()` to add axes.

Base graphics can be controlled by using `par()` to set graphical parameters, see the help page for `par()` for details. E.g.,

- `mfrow` and `mfcol` set multi-figure plotting, using a rectangular layout for subsequent plots instead of starting a new page for each plot
- `mar` sets the plot margins to the given lines of text
- `oma` sets the outer margins (for multi-figure layouts)

R in fact allows mathematical annotation a la LaTeX, but using R syntax: see `?plotmath` for details.

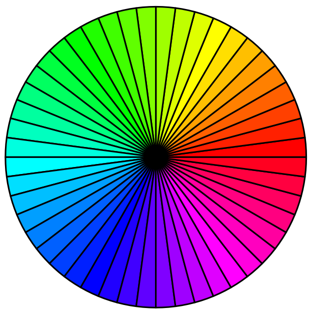
```
R> op <- par(cex = 1.3)
R> plot(1:6, 1:6, type = "n", xlab = "", ylab = "", axes = FALSE)
R> text(2, 2, expression(hat(alpha)))
R> text(3, 3, expression(frac(beta, gamma)))
R> text(4, 4, expression(integral(f(x) * dx, a, b)))
R> text(5, 5, expression(sqrt(pi)))
R> box()
R> par(op)
```



R also allows extremely flexible control of color in graphics.

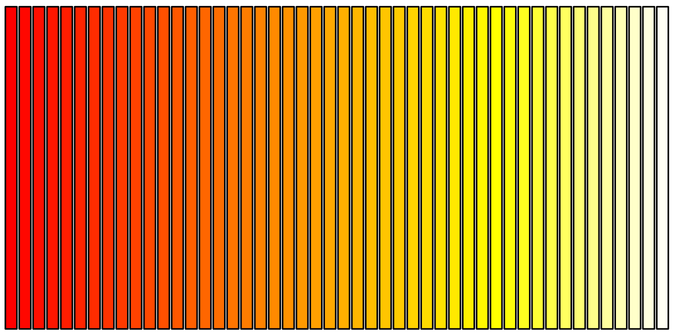
A simple rainbow color wheel:

```
R> pie(rep(1, 48), labels = "", col = rainbow(48), radius = 1)
```



A divergent heat color palette:

```
R> barplot(rep(1, 48), col = heat.colors(48), axes = FALSE)
```



In general, correctly using color is very hard . . .