1. Introduction. This is a "mock-up" of how the full power of literate programming might be useful to `R` programmers. It presents a simple (but interesting) function that computes the Kolmogorov–Smirnov statistic from empirical data. Then it shows some examples of that function in use.

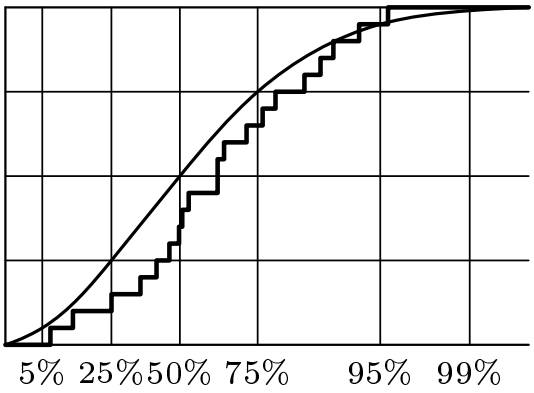
(The author apologizes for any awkward coding, due to the fact that this is actually his very first attempt to program in `R`.)

⟨ Define the function 5 ⟩
⟨ Test the function 9 ⟩

2. The Kolmogorov–Smirnov statistic. The *empirical distribution function* $F_n(x)$ of a set of samples X_1, X_2, ..., X_n is defined to be

$$F_n(x) = \frac{\text{number of } X_1, X_2, \ldots, X_n \text{ that are} \le x}{n}.$$

For example, the jagged line shown here is an empirical distribution function taken from §3.3.1 of the book *Seminumerical Algorithms*:



3. The curved line in that illustration is another cumulative distribution function, $F(x)$. One way to measure goodness of fit, proposed by A. N. Kolmogorov in 1933 and modified by N. V. Smirnov in 1939, is to compute

$$K_n^+ = \sqrt{n} \sup_{-\infty<x<+\infty} \big(F_n(x) - F(x)\big);$$
$$K_n^- = \sqrt{n} \sup_{-\infty<x<+\infty} \big(F(x) - F_n(x)\big).$$

Here K_n^+ measures the greatest amount of deviation when F_n is greater than F, and K_n^- measures the maximum deviation when F_n is less than F. The normalizing factor $\sqrt{n}$ ensures that K_n^+ and K_n^- will converge to a limiting distribution as $n \to \infty$, if the X's are independent samples from F.

4. The given data comes from distribution F if and only if $F(X_j)$ has the uniform distribution between 0 and 1.

Hence the obvious way to compute K_n^+ and K_n^- is to start by sorting the X's so that $X_1 \le X_2 \le \cdots \le X_n$. Then we can compare $Y_j = F(X_j)$ to the "ideal" value j/n.

However, the time needed for sorting is of order $n \log n$. We will use an improved method suggested by T. Gonzalez, S. Sahni, and W. R. Franta [*ACM Transactions on Mathematical Software* **3** (1977), 60–64], who noticed that *linear* time suffices if we place the unsorted samples into n bins.

Indeed, if we put Y_j into bin $\lfloor Y_j \rfloor$, the final statistics depend only on the smallest and largest elements in each of the n bins. Other elements in those bins don't contribute to the extremes that are measured by K_n^+ and K_n^-.

5. Coding details.

⟨ Define the function 5 ⟩ ≡

```
lit.ks.test=function(x,p) {
  n=length(x)
  y=n*p(x)
  ⟨ Return FALSE if y doesn't make sense 8 ⟩;
  ⟨ Insert y into bins, remembering extreme values 6 ⟩;
  ⟨ Find the overall extreme values lo, hi 7 ⟩;
  return (c(lo,hi)/sqrt(n))
}
```

This code is used in section 1.

6. The main idea is to keep track of $lb[j]$ and $ub[j]$, the smallest and largest element of bin $j-1$, as well as $count[j]$, the total number of elements in that bin, for $1 \le j \le n$.

(This is a situation where C programmers wish that `R` had 0-origin indexing.)

⟨ Insert y into bins, remembering extreme values 6 ⟩ ≡

```
lb=1:n
ub=count=array(0,dim=n)
for (j in 1:n) {
  yy=y[j]; k=yy%/%1+1
  if (yy<lb[k]) lb[k]=yy
  if (yy>ub[k]) ub[k]=yy
  count[k]=count[k]+1
}
```

This code is used in section 5.

7. Here's the key logic that makes it all work.

⟨ Find the overall extreme values *lo*, *hi* 7 ⟩ ≡

```
hi=lo=j=0
for (k in 1:n) {
  if (count[k]) {
    if (lb[k]-j>lo) lo=lb[k]-j
    j=j+count[k]
    if (j-ub[k]>hi) hi=j-ub[k]
  }
}
```

This code is used in section 5.

8. Of course a good subroutine intended for general use will check to see that bad parameters haven't been supplied. Otherwise the computer might hang up with subscripts out of range.

⟨ Return FALSE if y doesn't make sense 8 ⟩ ≡

```
OK=FALSE
if (sum(y<0)) message('That CDF has negative values!')
else if (sum(y>n)) message('That CDF has values > 1!')
else if (sum(y==n)) {
  for (j in 1:n)
    if (y[j]==n)
      message('x[',j,'] is too high: ',x[j])
} else OK=TRUE
if (!OK) return(FALSE)
```

This code is used in section 5.

9. A few unit tests. First let's make sure that those error messages are properly issued.

⟨ Test the function 9 ⟩ ≡

```
  lit.ks.test(-1, function(x) return(x))
That CDF has negative values!
[1] FALSE
  lit.ks.test(2, function(x) return(x))
That CDF has values > 1!
[1] FALSE
  lit.ks.test(c(.6,.23,pi,-.1,1), punif)
x[3] is too high: 3.14159265358979
x[5] is too high: 1
[1] FALSE
```

See also section 10.
This code is used in section 1.

10. And finally, we also want to obtain a valid result when we supply valid parameters.

(The built-in system function *ks.test* computes the slightly different statistics $D_n^- = K_n^-/\sqrt{n}$ and $D_n^+ = K_n^+/\sqrt{n}$.)

⟨ Test the function 9 ⟩ +≡

```
  x=c(.999,.21,.64,.87,.22)
  lit.ks.test(x,punif)/sqrt(5)
[1] 0.27 0.18
  ks.test(x,punif,alternative="less")
D^- = 0.27, p-value = 0.4134
alternative: the CDF lies below the null hypothesis
  ks.test(x,punif,alternative="greater")
D^+ = 0.18, p-value = 0.651
alternative: the CDF lies above the null hypothesis
```

11. Index.

⟨ Define the function 5 ⟩ Used in section 1.
⟨ Find the overall extreme values *lo*, *hi* 7 ⟩ Used in section 5.
⟨ Insert y into bins, remembering extreme values 6 ⟩ Used in section 5.
⟨ Return `FALSE` if y doesn't make sense 8 ⟩ Used in section 5.
⟨ Test the function 9, 10 ⟩ Used in section 1.

KS