

Speed comparisons *(apply() family)*

Generate a 500×500 matrix of random numbers and try to find the sums of each column, timing each method

1. Two nested loops, one over columns, one over rows within a column
2. One loop, using `sum()` for each column
3. `apply()`
4. matrix multiplication by a vector of 500 1s.

Look at the source code for `apply()`. Why isn't it faster than a `for()` loop?

Speed comparisons *(general optimisation)* You want to estimate the distribution of linear regression estimates when the error distribution is Cauchy rather than Normal. A dumb method is

```
beta<-NULL
for(i in 1:5000){
  x<-1:50
  y<-NULL
  for(j in 1:50){
    y<-c(y,rcauchy(1)+x)
  }
  model<-lm(y~x)
  coefs<-coef(model)
  beta<-c(beta,coefs[2])
}
```

Investigate the improvements from vectorising the random number generation, using `lsfit()` instead of `lm()`, allocating space outside the loop, using `solve(cbind(1,x),y)` instead of `lsfit()` and anything else that occurs to you.

Graph labelling *(substitute)*

Suppose you want to plot the Gamma density function (`dgamma`) for a vector of shape parameters, and label eg the graph for `shape= 1` as $\Gamma(1)$.

```

par(mfrow=c(2,2))
for (i in 1:4)
  curve(dgamma(x,shape=i),ylab=expression(Gamma(i)))

```

labels each curve with $\Gamma(i)$. Use `substitute()` to get the right label.

Simulation for power calculations (*loops, objects*)

An easy way to perform power calculations is simulation. Suppose you are testing a drug that reduces blood pressure. You have 50 people in each of treatment and control groups, and expect the systolic blood pressure to have a mean of 150mmHg and standard deviation of 15mmHg in the control group, and to have a mean of 140mmHg in the treatment group.

1. Assuming the distributions to be approximately Normal, simulate one set of data and perform a *t*-test using the `t.test` function.
2. Using the `names` function, look at the components of the object returned by `t.test`. The *p*-value is `t.test(x,y)$p.value`
3. Write a loop to generate data and perform a *t*-test 1000 times, storing the values in a vector `a`. What is the power of the study (the proportion of times the *p*-value is below 0.05)? Compare the results with those given by `power.t.test`.
4. Suppose in the treated group the standard deviation were increased to 20mmHg. The `power.t.test` function can't handle this, so rewrite your simulation to compute the power.
5. Suppose that instead of having a 10mmHg difference and wanting to find the power you want to find what difference gives 80% power. This involves trying multiple simulations to find one that gives the right answer. Decisions involve whether to simulate new random numbers each time or just to add or subtract a constant from each one, and whether to program your own search routine or use `uniroot`.

Receiver Operating Characteristic curves (*graphics, indexing, efficiency*).

Given a continuous test variable *T* and a binary status variable *D* the receiver operating characteristic (ROC) curve summarises how well *T* predicts *D*. They first arose in radio engineering, but now are most used in

medical diagnostics research. The ROC curve plots the true positive rate $P(T > c|D = 1)$ against the false positive rate $P(T > c|D = 0)$ for every possible threshold c . A perfect test has true positive rate 1 and false positive rate 0; a perfectly useless test has equal true and false positive rates.

1. For any given cutpoint the true and false positive rates can be computed

```
ptrue<-mean(T[D==1]>c)
pfalse<-mean(T[D==0]>c)
```

2. It is only necessary to compute this for observed values of c (and $-\text{Inf}$). Write a `for()` loop to do it.
3. Rewrite the `for()` loop to use `sapply()`. Is it faster? Easier to understand?
4. Write a function to draw the ROC curve from vectors D and T .
5. A way to speed up the calculation is to find a different algorithm. You can rewrite $P(T > c|D == 1)$ as $P(T > c \& D == 1)/P(D == 1)$. The denominator doesn't depend on c . The numerator can be computed by ordering the data appropriately and using the `cumsum()` command, which produces cumulative sums of a vector.
6. The area under the ROC curve is a useful summary of the discriminatory power of T . How would you compute it?
7. What if you only wanted the area under the portion of the curve with $P(D = 0|T > c)$ less than, say, 0.05, because the test would never be operated at a higher false positive rate. Update your function to compute this partial area under the curve.
8. Make your function return a ROC object that has sensible `plot` and `print` methods and a `summary` method that computes partial area under the curve.
9. Use `package.skeleton()` to start producing an R package with these functions.

Data to test your code can be found in the “survival” package, `data(pbc)`. Use bilirubin levels (`T<-pbc$bili`) as the test value, and define the status as two-year survival: `D<-pbc$status==1 & pbc$time<730`.

Clustered data regression (*model frames/formula, language*) In linear regression with clustered data the usual estimate for $\hat{\beta}$ works but the standard errors are wrong. A valid estimate of $\text{var}[\hat{\beta}]$ is

$$(X^T X)^{-1} (U^T U) (X^T X)^{-1}$$

where $U_i = \sum_t x_{it}(y_{it} - \mu_{it})$.

1. Suppose we have a function `mylm(formula,data)` The idiom for creating model matrices is

```
m<-match.call()
m[[1]]<-as.name(''model.frame'')
m<-eval(m,parent.frame()) ## the model frame
X<-model.matrix(terms(formula),m)
Y<-model.response(m)
```

Write a function to compute $\hat{\beta}$ and $(X^T X)^{-1}$.

2. Now we can add a `cluster=` argument to the function. When constructing the model frame the cluster argument will automatically be added. We can extract it with

```
group<-model.extract(m, ''cluster'')
```

and use the `rowsum()` function to compute the collapsed sums U . It is then easy to produce the correct model-robust variance matrix

3. (*tricky*) Suppose we wanted to put the cluster specification in the model formula, as, say, `y~x+id(group)`.

It would be necessary to break this into two formulas `y~x` and `~id(group)`. Look at what `terms(y x+id(group),specials='id')` does. The “specials” attribute identifies which part of the “variables” attribute is `id(group)`. So we can identify the real variables and the clustering variable. One approach to constructing the formulas is seen in the code for `aov` in handling the `Error()` term: use `paste` to produce character strings and then `as.formula` to convert them back to formulas. Try doing this.