Efficient Programming in S

Workshop on 6 October, 1998
International S-PLUS User Conference
Washington, DC

Presenters:

Bill Venables,	Brian Ripley,
University of Adelaide,	University of Oxford,
South Australia.	Oxford, UK.

`Bill.Venables@adelaide.edu.au`

`ripley@stats.ox.ac.uk`
`http://www.stats.ox.ac.uk/~ripley`

Programme Outline

- fundamnetal issues in the S language,
- large-scale computations in S,
- using compiled code,

—— Lunch ——

- operations on the language itself,
- preparing libraries and GUI code,
- object-oriented progrmming, new-style and old-style classes, generics and methods.
- comments, Q & A, discussion.

Presentation 1

1. Fundamental notions, a review.
2. Visibility issues and the search path.
3. Lazy evaluation and its consequence.
4. Recursion and iteration.
5. Vectorization.

Review of fundamentals

S is a *language* for manipulating *objects*.

What kind of language?

- A *function* language,
- An *interpreted* language,
- An *interactive* language.

The goal of S:
To turn ideas into software, quickly and faithfully.

Historically a language for
algorithm development and
technology transfer.

It is not necessarily the best environment for large, routine production jobs on present day hardware.

What kinds of objects?

- *Atomic* objects: mode *numeric*, *complex*, *character* or *logical*.

- *Language* objects: *function*s, *name*s, *call*s, *expression*s, grouped statements (objects of mode "{"), assignments, *quit*s.

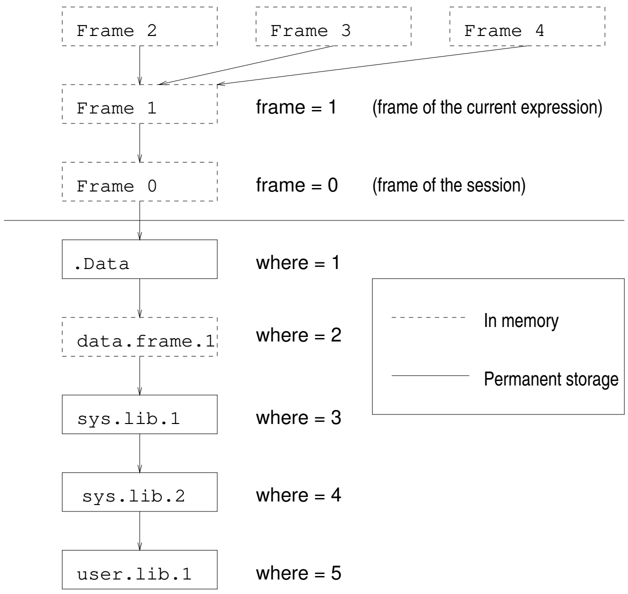
- *Recursive* objects: *list*s, *expression*s, *call*s, *function*s.

No distinction is made between the objects of data analysis and the objects of the language itself.

Objects are normally *permanent* until deliberately discarded.

The **search path** determines the (default) visibility of named objects.

The Search Path and Object Visibility



Consequences of restricted visibility

Consider a function to section each column of a matrix according to a factor and apply a function to each section.

```
f1 = function(X, f, FUN = mean) {
  v = apply(X, 2, function(x)
             tapply(x, f, FUN))
  v
}
```

This will *not* work since `f` and `FUN` are not visible from within the frame of the innermost function.

```
> Iris = as.data.frame.array(iris,
          col.dims = 2, index.dims = 3)
> X = as.matrix(Iris[, -5])
> Species = Iris[[5]]
> table(Species)
 Setosa Versicolor Virginica
     50         50        50
> f1(X, Species)
Problem in FUN(X): Object "f" not found
Debug ?  ( y|n ): n
```


If we do enter the debug phase, things are slightly more helpful than with the old engine, but still very arcane!

```
> f1(X, Species)
Problem in FUN(X): Object "f" not found
Debug ?  ( y|n ): y
Browsing in frame of FUN(...X.sub.i....,
Local Variables: .NAMES, FUN, X

R> where
8: FUN(X) from 7
*7: FUN(...X.sub.i...., .NAMES.... from 6
6: list() from 4
4: lapply(split(newX, rep.int.... from 3
3: apply(X, 2, function(x)  t.... from 2
2: f1(X, Species) from 1
1:  from 1
R> q
```

Asking `where` during the `recover` phase is very much like using `traceback()` after a static dump.

A rash solution: defy the S visibility rules, but taking some precautions by using ‘dot first’ names.

```
> f2  = function(X, .f, .FUN = mean) {
    v = apply(X, 2, function(x)
    tapply(x, get(".f", inherit = T),
           get(".FUN", inherit = T)))
    v
  }
> f2(X, Species)
           Sepal L. Sepal W. Petal L. Petal W.
    Setosa    5.006    3.428    1.462    0.246
Versicolor    5.936    2.770    4.260    1.326
 Virginica    6.588    2.974    5.552    2.026
```

It works, (this time)!

In S3 it still did not work though.

Why?

Lazy evaluation! In S3 at this stage `.f` and `.FUN` were only unevaluated arguments inside `f2`.

Corrected S3 solution, (but equally rash)

```
f2a <- function(X, .f, .FUN = mean) {
  .f <- .f
  .FUN <- .FUN
  apply(X, 2, function(x)
    tapply(x, get(".f", inherit = T),
           get(".FUN", inherit = T)))
}
```

Now it works in S3 and S4, but *NOT* recommended in either.

Second solution: Put copies of `f` and `FUN` in frame 1 where they can be globally seen (and where they will be automatically discarded when no longer needed).

```
f3 = function(X, f, FUN = mean) {
  assign("f", f, frame = 1)
  assign("FUN", FUN, frame = 1)
  apply(X, 2, function(x)
        tapply(x, f, FUN))
}
> f3(X, Species)
           Sepal L. Sepal W. Petal L. Petal W.
    Setosa    5.006    3.428    1.462    0.246
Versicolor    5.936    2.770    4.260    1.326
 Virginica    6.588    2.974    5.552    2.026
```

It works, but the function clearly connot be recursive. Frame 1 *is* globally visible, but best left for the engine to use.

Third (RTFM) solution: Read the help information more carefully for `apply` more carefully and see how additional arguments can be passed on to the argument function:

```
f4 = function(X, f, FUN = mean)
     apply(X, 2, tapply, f, FUN)

> f4(X, Species)
           Sepal L. Sepal W. Petal L. Petal W.
    Setosa    5.006    3.428    1.462    0.246
Versicolor    5.936    2.770    4.260    1.326
 Virginica    6.588    2.974    5.552    2.026
```

Now having a separate function at all seems rather unnecessary. If we can remember the calling protocol we could do it with

```
> apply(X, 2, tapply, Species, mean)
           Sepal L. Sepal W. Petal L. Petal W.
    Setosa    5.006    3.428    1.462    0.246
Versicolor    5.936    2.770    4.260    1.326
 Virginica    6.588    2.974    5.552    2.026
```

Note

`sapply` and `tapply` provide some pre- and post-processing to `lapply` which can be expensive.

In practice this extra processing is often not needed and combinations of `lapply`, `unlist` and `split` can be used.

The following simple front-ends to `lapply` often suffice.

```
> Sapply = function(...) unlist(lapply(...))

> Tapply = function(v, f, FUN, ...)
    unlist(lapply(split(v, f), FUN, ...)
```

Similarly `table` is a convenient interface function to `tabulate`, and can be much slower.

Where speed is necessary `tabulate` should be used as directly as possible.

Lazy evaluation

Arguments are only evaluated if and when they are first used.

Evaluation is done in the parent frame.

Default values are evaluated in the local frame.

Example:

```
> f0 = function(x, y = x) x + y
> f1 = function(x, y = x) y + x
> f0(a <- 1)        # OK since x eval. first
[1] 2
> rm(a)
> f1(a <- 1)        # OK since default first
[1] 2
> rm(a)
> f0(a <- 1, a)     # OK since x eval. first
[1] 2
> rm(a)
> f1(a <- 1, a)     # fails, default not used
Problem: Object "a" not found
```

(No recovery phase this time.)

“`integrate()` cannot integrate constants!”

```
> con = function(x) 1
> integrate(con, 0, 1)$integral
Problem in qf15(f, lower, upper,
   aux = optargs(list(..: f is not vectorized -
   length(f(x))!=length(x)
Debug ?  ( y|n ): n
```

`integrate` expects the integrand function to be *vectorized* wrt the principal argument: given a vector of x's it must produce a result *of the same length*.

(In S-PLUS 3.4 this silently gives a wrong answer!)

```
> con1 = function(x) 1 + 0*x
> integrate(con1, 0, 1)$integral
[1] 1
```

More subtly, suppose we wish to integrate a function of the form

$$f(x) = (a_1x - b_1)(a_2x - b_2)\cdots(a_nx - b_n)$$

that is, a polynomial in factor form. The temptation is to code this as

```
> fab = function(x, a, b) prod(a*x - b)
```


How could we code this integrand in a vectorized way (wrt `x`)?

More generally we consider five self-contained functions that evaluate integrals of this kind.

1. Using a recursive integrand function.

```
fp1 = function(a, b, lo = 0, up = 1) {
  fxab = function(x, a, b)
    if(length(a) == 0) rep(1, length(x)) else
      (a[1]*x - b[1])*Recall(x, a[-1], b[-1])
  integrate(fxab,lo,up, a = a, b = b)$integral
}
```

The internal function will *not* work unless `Recall` is used to handle the recursive call, because of frame visibility. (Syntax error if `else` is on the next line.)

Not especially practical, but it works in fair time.

Still works if `a` and `b` are empty.

2. Use an explicit loop.

```
fp2 = function(a, b, lo = 0, up = 1) {
  fxab = function(x, a, b) {
    f = rep(1, length(x))
    for(i in seq(along = a))
      f = f * (a[i] * x - b[i])
    f
  }
  integrate(fxab,lo,up, a = a, b = b)$integral
}
```

Simple, transparent and iterative. Still works if `a` and `b` are empty but relies on `seq(along = a)` instead of, e.g., `1:length(a)`.

3. Using `apply` and `outer`.

```
fp3 = function(a, b, lo = 0, up = 1) {
  fxab = function(x, a, b)
    drop(apply(a %o% x - b, 2, prod))
  integrate(fxab,lo,up, a = a, b = b)$integral
}
```

The `drop()` may not be necessary. Will `not` work if `outer(a, x)` is replaced by `outer(x, a)`. Will *not* work if `a` or `b` are empty.

4. Construct the integrand with a and b coded in.

```
fp4 = function(a, b, lo = 0, up = 1) {
  expr = paste("(", a, " * x - ", b, ")",
               collapse = "*")
  fxab = substitute(function(x) body,
          list(body = parse(text = expr)[[1]]))
  integrate(fxab, lo, up)$integral
}
```

Somewhat resource hungry if `a` and `b` are large.
Will *not* work for empty `a` or `b`.

5. Do not use `integrate` at all.

```
fp5 = function(a, b, lo = 0, up = 1) {
  p = 1
  for(i in seq(along = a))
    p = c(0, a[i] * p) - c(b[i] * p, 0)
  ui = li = 0
  for(v in c(rev(p)/length(p):1, 0)) {
    ui = up * ui + v
    li = lo * li + v
  }
  ui - li
}
```

Will work if `a` or `b` are empty.
Vectorized for `lo` and `up`.

Notes

• A standard convention would be to allow *recycling* of the `a` and `b` arguments. This is automatic with `fp4` but for the others it has to be done manually, e.g.

```
l = max(length(a), length(b))
a = rep(a, len = l)
b = rep(b, len = l)
```

• We should check if they work! With $a_i = 1$ and $b_i = 0, 0, \frac{1}{2}, \frac{1}{2}, 1, 1$ the exact integral is $1/840$. At the same time we will get `fp4` to report the function it has constructed.

```
> trace(fp4, tracer = Quote(print(fxab)),  # S4
        at = 3, print = F)
> a = rep(1,6)
> b = rep(0:2/2, 2)
> 840 * c(fp1 = fp1(a, b), fp2 = fp2(a, b),
          fp3 = fp3(a, b), fp4 = fp4(a, b),
          fp5 = fp5(a, b))

function(x)
(1 * x - 0) * (1 * x - 0.5) * (1 * x - 1) *
      (1 * x - 0) * (1 * x - 0.5) * (1 * x - 1)

 fp1 fp2 fp3 fp4 fp5
   1   1   1   1   1
> untrace()                # Don't forget this!
```


Presentation 2

1. Resource monitoring.

2. Simple simulation techniques.

3. Working with large objects.

4. Efficient code practices.

5. Tips on memory management.

Working with iterations and large objects

As most S-PLUS programmers fairly quickly discover, writing code that exhausts even the largest RAM and stymies the system (either for your own personal workstation, or, even worse for colleagues who use the same shared system), is easy but not useful! 'Page thrashing' becomes a term rapidly well understood.

S-PLUS is *not* the best program for large, routine jobs but with care it can be made to handle almost any problem, even the large and routine ones.

This talk will give some hints on

- monitoring resources (time and memory),
- using memory efficiently,
- using inbuilt vectorization and other devices to cut down resource expenditure.

Measuring resource usage

`proc.time` is a primary timing function. Memory measurement changed with S-PLUS 4.0.

1. S-PLUS 3.4 and earlier: `memory.size`
2. S-PLUS 4.0 and later: `mem.tally.reset`, `mem.tally.report`

Memory allocation is handled by the C function `S_alloc`.

Cache memory: Used for holding function definitions in memory for fast later access.

Evaluation memory: Used for temporaries needed in the evaluation of expressions. (Not always fully reclaimed.)

`mem.tally.report` gives the maximum simultaneous allocation of memory, of both kinds, since the last call to `mem.tally.reset`.

A Resource measuring function

```
resources = function(expr) {
  l = sys.parent(1)
  if(l == 1) l = F
  on.exit(cat("Timing stopped at:",
               proc.time() - time, "\n"))
  expr = substitute(expr)
  time = proc.time()
  mem.tally.reset()
  w = eval(expr, local = l)      # "the goods"!
  new.time = proc.time()
  mem = mem.tally.report()
  on.exit()
  names(mem) = c("Cache", "Working")
  if(length(new.time) == 3)
    new.time = c(new.time, 0, 0)
  if(length(time) == 3)
    time = c(time, 0, 0)
  time = new.time - time
  time[3] = max(time[3], time[1] + time[2])
  print(c(CPU = time[1] + time[2],
    Elapsed = time[3], "% CPU" = round((100 *
              (time[1] + time[2]))/time[3], 1),
    Child = time[4] + time[5], mem))
  invisible(w)
}
```

(Based on an original by Bill Dunlap.)

The cost of growing a large object.
First with S-PLUS 3.4 on a SparcStation 1/170, 64Mb RAM:

```
> resources({  # grow a large object----------
    X <- NULL
    for(i in 1:100)
    X <- cbind(X, sort(rnorm(100)))
  })
    CPU Elapsed % CPU Child Cache Working
   0.31    0.31   100     0     0 2650265
```

(The cache should probably be zero!) Next with the current S-PLUS 5.0 on the same machine:

```
>  resources({  # grow a large object----------
    X = NULL
    for(i in 1:100)
    X = cbind(X, sort(rnorm(100)))
  })
  CPU Elapsed % CPU Child Cache Working
 0.23    0.23   100     0     0  223465
```

(It is supposed to be) a better policy is to allocate sufficient storage initially, if possible.

```
> resources({  # fill a large object-----------
    X = matrix(0, 100, 100)
    for(i in 1:100)
    X[, i] = sort(rnorm(100))
  })
  CPU Elapsed % CPU Child cache Working
 0.82       1    82     0 13352 8077536 (S4)
 0.23    0.23   100     0     0  223465 (S3)
```

The use inbuilt vectorized functions to avoid S loops can be useful.

```
> resources({  # avoid explicit loops----------
    X = rnorm(10000)
    X = matrix(X[order(rep(1:100, 100), X)],
                100, 100)
  })
  CPU Elapsed % CPU Child cache Working
 0.21    0.21   100     0 59024  343620 (S4)
 0.19       1    19     0 11312  632322 (S3)

> resources(X <- apply(
    matrix(rnorm(10000), 100, 100), 2, sort))
  CPU Elapsed % CPU Child cache Working
 1.59       2  79.5     0 96152  212556 (S4)
  0.3     0.3   100     0 72720  811208 (S3)
```


Loops

S is designed to be able to back out from uncompleted calculations, so that the memory used in intermediate calculations is retained until they are committed.

This applies to `for`, `while` and `repeat` loops, for which none of the steps are committed until the whole loop is completed.

The same applies even more forcefully to `source`

Vectorization to avoid loops often pays off.

Example: Find the distribution of $ab - cd$ where a, b, c and d are independent uniform digits.

1. A naive loop method.

```
> resources({
    v = numeric(10000)
    i = 0
    for(a in 0:9)
      for(b in 0:9)
        for(d in 0:9)
          for(e in 0:9)
            v[i = i + 1] = a*b - d*e
    fr = tabulate(v + 82)
  })
   CPU Elapsed % CPU Child cache Working
 31.57    33    95.7     0     0 9295876 (S4)
  3.2      3.2   100     0  6464  437256 (S3)
```

2. A loop-free vectorized method.

```
> resources({ v = outer(0:9, 0:9)
    fr = tabulate(v <- outer(v, v, "-") + 82)
  })
  CPU Elapsed % CPU Child  cache Working
  0.1    0.17   100     0 141768  214616 (S4)
  0.11   0.11   100     0  22624  214240 (S3)
```


Moral: Use all the mathematical cunning you can

Example: Taking bootstrap samples:

```
> resources({
    mat = matrix(0, 1000, 6)
    for(i in 1:1000)
      mat[i, ] = sample(x, replace = T)
  })
 CPU  Elapsed % CPU Child  cache  Working
 7.53      19  39.6     0 365512 48627808
 1.45    1.45   100     0      0   319592
```

1000 sample of size 6 *with replacement* is the same as taking *one* sample of size 6000 with replacement and dividing it into 1000 lots:

```
> resources(mat <-
   matrix(sample(x, 6000, replace=T), 1000, 6))
 CPU    Elapsed % CPU Child cache Working
 0.03      0.03   100     0    40  140416
 0.02      0.02   100     0  2424  144944
```

Of course it may also pay to think about what you are going to do next with the samples!

Example Eric Gibson, S-news, May 1997.

Simulate the distribution of $1 - r^2$ where r is the correlation coefficient between a sorted normal sample of size 10 with a fixed vector, a. (Used in connexion with the Shapiro-Wilk test for normality.) Use 10000 samples.

Original method: (`a` is specified)

```
> resources({
    tmp = matrix(10, 10000, 1)
    tmp = t(apply(tmp, 1, rnorm))
    tmp = t(apply(tmp, 1, sort))
    SWstat = function(y)
          {return(1-cor(matrix(y, 10, 1),a)^2)}
    tmp = apply(tmp, 1, SWstat)
    tmp = sort(tmp)
})
Timing stopped at: 985.39 48.42 1110.75 0 0 (S4)
    CPU Elapsed % CPU Child  Cache  Working
 760.93     825  92.2     0      0 12134784 (S3)
```

[Exercise: Pick all unnecessary steps in the code without changing the essential method.]

If y is one such sorted normal sample then the correlation coefficient is

$$r = \sum_{i=1}^{10} y_i^\star a_i^\star$$

where $y_i^\star = (y_i - \overline{y})/\sqrt{\sum(y_i - \overline{y})^2}$, that is, y mean corrected and normalized to unit length. $a_i^\star$ is defined similarly.

```
 > resources({
    a = a - mean(a)
    a = a/sqrt(sum(a^2))

    tmp = rnorm(10*10000)
    tmp = matrix(tmp[order(rep(1:10000, 10),
                            tmp)], 10, 10000)
    mns = rep(1,10) %*% tmp / 10
    tmp = tmp - rep(mns, rep(10, 10000))
    cln = sqrt(rep(1,10) %*% (tmp * tmp))
    tmp = tmp/rep(cln, rep(10, 10000))
    rm(mns, cln)
    tmp = sort(1 - (a %*% tmp)^2)
  })
  CPU Elapsed % CPU Child cache Working
 3.08       7    44     0  3368 2066180 (S4)
 3.58    3.58   100     0 52520 6076362 (S3)
```

We could make it more streamlined looking by using `scale`, but the gain in both time and memory use is negative:

```
> resources({
  tmp = rnorm(10*10000)
  tmp = scale(matrix(tmp[
     order(rep(1:10000,10), tmp)], 10, 10000))
  tmp = sort(1 - (t(scale(a)) %*% tmp / 9)^2)
})
 CPU Elapsed % CPU Child cache Working
 3.4       4    85     0 88688 2078772 (S4)
 3.49   3.49   100     0 11312 7936968 (S3)
```

We persist with the longer ‘hand’ method and look at extra ways of managing time and memory use.

There may be some advantage in splitting it into several sections and working with a loop.

```
> resources({  verbose = T
  a = a - mean(a)
  a = a/sqrt(sum(a^2))
  sws = numeric(10000)
  ind = rep(1:1000, 10)
  ten = rep(10, 1000)
  u = rep(1,10)
  for(i in 0:9) {
    tmp = rnorm(10*1000)
    tmp = matrix(tmp[order(ind, tmp)],10,1000)
    tmp = tmp - rep(u %*% tmp / 10, ten)
    tmp = tmp/rep(sqrt(u %*% (tmp * tmp)),ten)
    sws[1000*i + 1:1000] = 1 - (a %*% tmp)^2
    if(verbose) cat("Slab",i + 1,"finished.\n")
  }
  rm(tmp, ind, ten, u)
  sws = sort(sws)
})
Slab 1 finished.
Slab 2 finished.
....
Slab 10 finished.
  CPU Elapsed % CPU Child  cache Working
 3.72       9  41.3     0 358432  423196 (S4)
 2.95       3  98.3     0      0  760683 (S3)
```

In S-PLUS 3.4, not much saving in time but a big saving of memory.

Capital 'F' For loops:

It is sometimes advantageous to partition some of the simulation and perform it in a sub-process using `For` loops.

```
> resources({  verbose = T
  a = a - mean(a)
  a = a/sqrt(sum(a^2))
  sws = numeric(10000)
  ind = rep(1:1000, 10)
  ten = rep(10, 1000)
  u = rep(1,10)
  For(i = 0:9, {
    tmp = rnorm(10*1000)
    tmp = matrix(tmp[order(ind, tmp)],10,1000)
    tmp = tmp - rep(u %*% tmp / 10, ten)
    tmp = tmp/rep(sqrt(u %*% (tmp * tmp)),ten)
    sws[1000*i + 1:1000] = 1 - (a %*% tmp)^2
    if(verbose) cat("Slab",i + 1,"finished.\n")
  })
  rm(tmp, ind, ten, u)
  sws = sort(sws)
})
Slab 1 finished.
....
Slab 10 finished.
  CPU Elapsed % CPU Child  cache Working
 0.47      13   3.6  5.65 460680  356628 (S4)
 0.07       3   2.3  3.58  52520  307352 (S3)
```


Increasing the grain size may speed up the calculation (but at the cost of memory). (This failed on S-PLUS 5.0 for Linux with a floating point exception.)

```
> resources({  verbose = T
  a = a - mean(a)
  a = a/sqrt(sum(a^2))
  sws = numeric(10000)
  ind = rep(1:1000, 10)
  ten = rep(10, 1000)
  u = rep(1,10)
  For(i = 0:9, {
    tmp = rnorm(10*1000)
    tmp = matrix(tmp[order(ind, tmp)],10,1000)
    tmp = tmp - rep(u %*% tmp / 10, ten)
    tmp = tmp/rep(sqrt(u %*% (tmp * tmp)),ten)
    sws[1000*i + 1:1000] = 1 - (a %*% tmp)^2
    if(verbose) cat("Slab",i + 1,"finished.\n")
  }, grain.size = 5)  # spawn 2 child processes
  rm(tmp, ind, ten, u)
  sws = sort(sws)
})
Slab 1 finished.
....
Slab 10 finished.
 CPU Elapsed % CPU Child  cache Working
 0.4      11   3.6  5.61 468328  363044 (S4)
0.08       4     2  3.43  12928  307352 (S3)
```

Other variations are possible.

Envelopes for normal scores plots by simulation

A graphical model checking device for linear models.

- Fit the linear model, calculate the residuals and standardise them to unit variance.

- Generate N (≈ 100) normal samples, and for each

  - fit the same linear model and

  - calculate the standardized residuals.

- Sort each standardized residuals sample (original and generated).

- the envelope consists of the lower and upper 0.025 quantiles of the N generated standardized residuals at each ordered position.

One way *not* to calculate envelopes:

```
hat <- matrix(0, 31, 3)
hat[, 1] <- 1
hat[, 2:3] <- as.matrix(dat[, 1:2]
hat <- hat %*%
   solve((t(hat) %*% hat)) %*% t(hat)
ident <- diag(31)
epsilon <- matrix(0, 31, 100)
e <- matrix(0, 31, 100)
e1 <- numeric(31)
e2 <- numeric(31)
for(i in 1:100) {
  epsilon[, i] <- rnorm(31, 0, 1)
  e[, i] <- (ident-hat)%*%epsilon[, i]
  e[, i] <- sort(e[, i])
}
for(i in 1:31) {
  eo <- sort(e[i, ])
  e1[i] <- eo[5]
  e2[i] <- eo[95]
}
ylim <- range(res, e1, e2)
par(pty = "s")
qqnorm(res, ylab = "Std. Residuals",
       ylim = ylim)
par(new = T)
qqnorm(e1, axes = F, xlab = "", ylab = "",
       type = "e", ylim = ylim)
par(new = T)
qqnorm(e2, axes = F, xlab = "", ylab = "",
       type = "e", ylim = ylim)
```

A functional version of the preceding method, (with some of the more egregious errors removed):

```
env2 = function(obj) {
  res = resid(obj)
  n = length(res)
  H = model.matrix(obj)
  p = ncol(H)
  res = res/sqrt(sum(res^2)/(n-p))
  H = H %*% solve(t(H) %*% H)  %*% t(H)
  ident = diag(n)
  epsilon = matrix(0, n, 100)
  e = matrix(0, n, 100)
  e1 = numeric(n)
  e2 = numeric(n)
  for(i in 1:100) {
    epsilon[, i] = rnorm(n, 0, 1)
    e[, i] = (ident - H) %*% epsilon[, i]
    e[, i] = sort(e[, i])/sqrt(sum(e[, i]^2)/(n-p))
  }
  for(i in 1:n) {
    eo = quantile(e[i, ], c(0.025, 0.975))
    e1[i] = eo[1]
    e2[i] = eo[2]
  }
  ylim = range(res, e1, e2)
  oldpar = par(pty = "s")
  on.exit(par(oldpar))
  qqnorm(res, pch = 1, ylim = ylim,
         ylab = "Std. Residuals",
         main = deparse(obj$call$formula))
  par(new = T)
  qqnorm(e1, axes = F, xlab = "", ylab = "",
         type = "l", ylim = ylim)
  par(new = T)
  qqnorm(e2, axes = F, xlab = "", ylab = "",
         type = "l", ylim = ylim)
  invisible(obj)
}
```


An alternative envelope function

```
env1 =
function(object, nsamples = 100, alpha = 0.05) {
  n = length(r <- resid(object))
  p = length(coef(object))
  Y = scale(qr.resid(qr(model.matrix(object)),
        matrix(rnorm(n * nsamples), n, nsamples)),
                     F, T) * sqrt((n - 1)/(n - p))
  Y[, ] = Y[order(col(Y), Y)]
  Y = matrix(Y[order(row(Y), Y)], nsamples, n)
  x0 = quantile(1:nsamples, c(alpha/2, 1-alpha/2))
  if(all(x0 %% 1 == 0))
    elim = t(Y[x0, ])
  else {
    x1 = c(floor(x0), ceiling(x0))
    elim = cbind(
          Y[x1[1], ] + (Y[x1[3], ] - Y[x1[1], ])/
          (x1[3] - x1[1])*(x0[1] - x1[1]),
          Y[x1[2], ] + (Y[x1[4], ] - Y[x1[2], ])/
          (x1[4] - x1[2])*(x0[2] - x1[2]))
  }
  res = sort(r)
  res = res/sqrt(sum(res^2)/(n-p))
  res = structure(list(res=res, elim=elim),
           label = deparse(object$call$formula))
  oldClass(res) = "envelope"
  res
}
```

Having an (old-style) "`envelope`" object allows us to write `plot` and `print` methods for them.

```
plot.envelope = function(x, ...) {
  n = length(x$res)
  ylim = range(x$res[c(1,n)], x$elim[c(1,n),])
  nscores = qnorm(ppoints(n))
  oldpar = par(pty = "s")
  on.exit(par(oldpar))
  plot(nscores, x$res, pch = 1, ylim = ylim,
       xlab = "Normal scores",
       ylab = "Sorted residuals",
       main = attr(x, "label"))
  lines(nscores, x$elim[,1])
  lines(nscores, x$elim[,2])
  invisible(x)
}

print.envelope = function(x, ...) {
  lo = x$res < x$elim[,1]
  hi = x$res > x$elim[,2]
  flash = rep("", length(lo))
  flash[lo] = "<"
  flash[hi] = ">"
  if(any(lo | hi))
    print(cbind(do.call("data.frame",x),
                        flash=flash)[lo | hi,])
  else
    cat("All points within envelope\n")
  invisible(x)
}
```

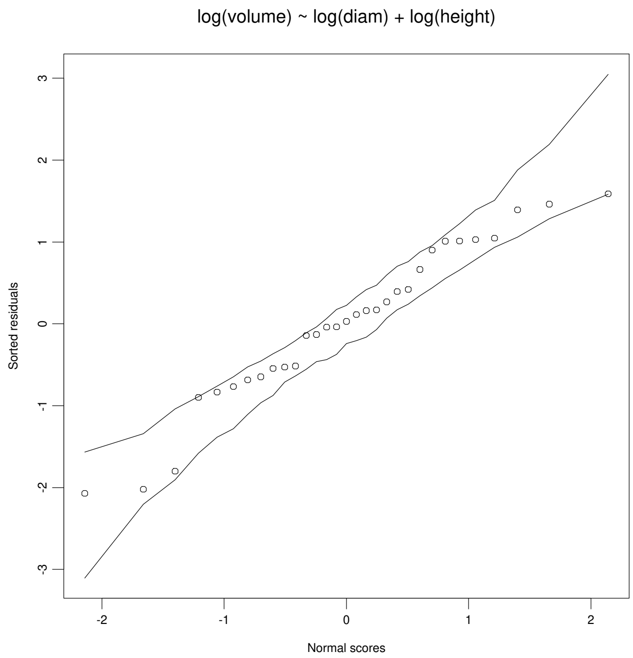

Black Cherry Tree Data

Predict volume from height and diameter.

```
> trees
   diam height volume
 1  8.3     70   10.3
 2  8.6     65   10.3
....
30 18.0     80   51.0
31 20.6     87   77.0
> fm = lm(log(volume)~log(diam)+log(height),
           trees)

> resources(env2(fm))
   CPU Elapsed % CPU Child  cache Working
 9.2       9.2   100     0 801488 8378731 (S4)
 1.42        2  70.8     0 334512  586296 (S3)

> print(resources(plot(env1(fm))))
  CPU Elapsed % CPU Child  cache Working
 0.97       1    97     0 115032  346624 (S4)
 0.23    0.23   100     0  78376  394962 (S3)
```



Regression is a HUGE problem

In May, 1996 Sarah Parish asked on S-news:

I am having memory problems trying to do various regressions on a Sun with 48Mb ram. As a start, I am trying to fit an `lm` *to 17065 observations, regressing on a factor variable with 107 values. Splus demands a huge amount of memory (see below). Is there anything I can do to get round this? I want to proceed to add further terms, probably another 30 effects. Can anyone give me an idea of how much memory I would need to install on the Sun to do this - or should I be using some other software?*

```
> options(object.size=20000000)
> lmch = lm(chol ~ dachol+cc, data = lipdat)
  Error in .Fortran("dqrls",: Unable to obtain
  requested dynamic memory (this request is
  for 15351616 bytes, 68056560 bytes already
  in use)
```

V and R responded independently.

We consider here only the problem of finding the regression coefficients. Take a case with 4 regressors in addition to the factor.

```
> fac = factor(sample(1:107, 17065, rep=T))
> range(table(fac)) # all classes sampled
[1] 123 196
> N = 17065
> X0 = matrix(rnorm(4 * N), N, 4)
> y = (1:107)[fac] + X0 %*% rep(1,4) +
         rnorm(N)/10
> (107 + 4) * N * 8
[1] 15153720
```

Method V (modified Gram-Schmidt process):

1. Correct all $x-$ and $y-$variables by subtracting factor class means.
2. Regress the corrected y on the corrected xs giving the regression coefficients.
3. Adjust the original y by removing the regression components on the x's, and find factor means, giving the factor effects.


```
> Vreg = function(fac, X0, y) {
  mc = function(x, f)   # 'mean correct'
    x - (unlist(lapply(split(x, f), mean)))[f]
  rc = function(X, y)   # 'regression coeff'
    solve(crossprod(X), crossprod(X, y))
  b2 = rc(apply(X0, 2, mc, fac), mc(y, fac))
  b1 = unlist(lapply(split(y - X0%*%b2, fac),
                     mean))
  c(b1, b2)
}

> resources(b2 <- Vreg(fac, X0, y))
   CPU Elapsed % CPU Child  cache Working
 15.82      17  93.1     0 246976 1044256 (S4)
  1.81       6  30.2     0  70296 5898328 (S3)

> round(as.vector(b2), 2)
  [1]   0.99   1.99   3.00   3.99   5.00   6.00
  [7]   6.99   7.99   9.01  10.00  11.00  12.00
....
[103] 103.01 104.00 105.00 106.00 106.99   1.00
[109]   1.00   1.00   1.00
```

It seems to work. Residual sums of squares and standard errors for regression coefficients are now not difficult to calculate.

Method R (as simplified by V):

Use known patterns in the $X^\top X$ and $X^\top y$ matrices to construct them without forming X explicitly.

```
> Rreg = function(fac, X, y) {
  ni = tabulate(unclass(fac))
  p1 = length(ni)
  p = ncol(X)
  XX = matrix(0, p1 + p, p1 + p)
  diag(XX) = c(ni, rep(0, p))
  XX[p1 + (1:p), p1 + (1:p)] = crossprod(X)
  XX[p1 + (1:p), 1:p1] =
    t(XX[1:p1, p1 + (1:p)] =
      matrix(apply(X, 2, function(x, f)
                   unlist(lapply(split(x, f),
                                 sum)), fac),
             p1, p))
  b = c(unlist(lapply(split(y, fac), sum)),
        crossprod(X, y))
  drop(solve(XX, b))
}
> resources(b1 <- Rreg(fac, X0, y))
   CPU Elapsed % CPU Child  cache Working
 10.39   10.39   100     0  59904 1132868 (S4)
  1.12    1.12   100     0  17776 3531952 (S3)

> range(b1-b2)
[1] -1.421085e-13  9.947598e-14
```


Tips From Chambers, *Programming with Data*, page 171 *et seq.*

- Memory fragmentation after a large calculation can be ameliorated using `reloadS` (but not yet. . . !)
- Break long computations into a sequence of tasks. Use the general event management facilities and `setReader` in particular.
- Building a large object in stages, preferentially use S *replacement* functions.
- For character matching use `string` rather than `character` and the `stringTable` facility for repeated matching.
- Calling C or Fortran you can minimize copying using the extra `CLASSES` and `COPY` arguments to the interface functions.
- Use `readRaw` for large amounts of atomic data.

Tips from V&R2.

- `remove` can be used to remove unneeded intermediate results:

```
remove("temp", frame = sys.nframe())
```

- Use `get` with an explicit frame or with the argument `immediate=T` instead of letting the parser find a data object.

- Use `assign` with `immediate=T` or a direct call to `dbwrite` to save the memory associated with backing-out an assignment.

- Code the innermost calculations as `C` or `Fortran` functions for fully efficient computations.

Presentation 3

1. Structure of language objects, Concepts.

2. Constructing language objects.

3. Parsing, substituting and evaluation.

4. An example: differentiation of mathematical expressions.

5. A case study: A statistical function for transformations of parameters.

Explorations in the language itself

The S language has tools for manipulating language objects themselves.

In S you have the option of using S to construct your ‘command’ *and* to evaluate it.

The evaluator evaluates *calls*. These can be objects of mode `numeric`, `name` or `call`.

An *expression* is a vector of *call* objects (very like a list).

In S4 `Quote` is available to quote one particular call.

We can put together *call*s and evaluate them using quite low-level tools:

```
> v = vector("expression", 3)
> v[[1]] = as.name("+")
> v[[2]] = as.name("x")
> v[[3]] = as.name("y")
> mode(v) = "call"
> v
x + y
> eval(v, local = list(x = 1:5, y = -1))
[1] 0 1 2 3 4

> v = Quote(x+y)
> eval(v, local = list(x = 1:5, y = -1))
[1] 0 1 2 3 4

> eval(Quote(x + y),
       local = list(x = 1:5, y = -1))
[1] 0 1 2 3 4
```

Coercion from `list` to `call` is also possible:

```
> v = lapply(c("+", "x", "y"), as.name)
> mode(v) = "call"
> v
x + y
```

The `do.call` function is initially very mysterious but is in fact very simple:

```
> do.call
function(what, args = list())
{
    if(!is.list(args))
        stop("args should be a list")
    if(length(args))
        this.call = c(as.name(what), args)
    else this.call = list(as.name(what))
    mode(this.call) = "call"
    eval(this.call, local = sys.parent(1))
}
> do.call("+", list(e1 = 1:3, e2 = 3:1))
[1] 4 4 4
```

Using `do.call` can often incur memory overheads if large objects or frames are copied in memory.

The lower-level function, `call`, can also be used directly.

If F is the distribution function, f the probability density function of a single observation, the pdf of the rth *order statistic* in a sample of size n is

$$\binom{n}{r-1} F(x)^{r-1}\{1 - F(x)\}^{n-r} f(x)$$

```
# function that creates the density function
# of an order statistic
d.order = function(n, r, pfunc, dfunc) {
  con = as(exp(lgamma(n + 1) - lgamma(r) -
            lgamma(n - r + 1)), "integer")
  c1 = call(substitute(pfunc), as.name("x"))
  c2 = call("^", c1, r - 1)
  c3 = call("^", call("-", 1, c1), n - r)
  c4 = call("*", con, call("*", c2,
        call("*", c3, call(substitute(dfunc),
                              as.name("x")))))
  substitute(function(x) body, list(body = c4))
}

> d.9.6 = d.order(9, 6, pnorm, dnorm)
> d.9.6
function(x)
503*(pnorm(x)^5 * ((1 - pnorm(x))^3*dnorm(x)))
```

(Taken from the help file for `call`, with adaptations.)

A simpler version using **`substitute`**

```
> pdf.order = function(n, r, pfun, dfun)
       substitute(function(x) {
                    Fx = p(x)
                    K*Fx^r1*(1 - Fx)^nr*f(x)
                  },
                  list(p = substitute(pfun),
                       f = substitute(dfun),
                       r1 = r - 1, nr = n - r,
         K = as(exp(lgamma(n + 1) - lgamma(r) -
               lgamma(n - r + 1)), "integer")))
> pdf.order(9, 6, pnorm, dnorm)
function(x)
{
        Fx = pnorm(x)
        503 * Fx^5 * (1 - Fx)^3 * dnorm(x)
}
```

`substitute` may be used to suppress evaluation of arguments, or make substitutions in a template language entity.

The first argument to `substitute`, `Quote` and `expression` are *quoted*, that is, left unevaluated even after 'lazy evaluation'.

The following table comes from the white book.

Category	Syntactic construction	Mode
Literals	`number`	`"numeric"`
	`string`	`"character"`
	`name`	`"name"`
	`comment`	`"comment"`
	`function ( formals ) expr`	`"function"`
Calls	`expr infix expr`	`"call"`
	`expr %anything% expr`	
	`unary expr`	
	`expr ( arglist )`	
	`expr [ arglist ]`	
	`expr [[ arglist ]]`	
	`expr $ fname`	
Assignment	`expr expr`	`"<-"`
	`expr _ expr`	
	`expr -> expr`	
	`expr <<- expr`	`"<<-"`
Conditional	`if ( expr ) expr`	`"if"`
	`if ( expr ) expr else expr`	
Iteration	`for ( name in expr ) expr`	`"for"`
	`repeat expr`	`"repeat"`
	`while ( expr ) expr`	`"while"`
Flow	`break`	`"break"`
	`next`	`"next"`
	`return ( expr )`	`"return"`
	`( expr )`	`"("`
	`{ exprlist }`	`"{"`

Some language manipulation tools

`Quote` Returns unevaluated argument as an object of mode `"call"`. (New to S4).

`expression` Returns unevaluated arguments as components of an object of mode `"expression"`.

`call` Constructor for objects of mode `call`. The components of `call`s are of mode `name` or `numeric` (or `integer` in S4).

`as.name` Coercion function from (generally) character string to *name*.

`mode<-` Direct coercion from one mode to another. Most coercions work.

Some language manipulation tools, (cont'd)

`parse` Using `text = "str"` converts a string into an *expression*.

`deparse` Coerces an expression or call back to a character string (vector) *in a similar form*.

`substitute` Allows substitutions in template expressions.

`eval` Evaluates a `call` or `expression` in the context of a `frame`

`all.vars` Extracts all unique simple names from an expression or call and returns them as a character string vector.

Formulae

The function `"~"` is similar to a special case of `expression`. Unevaluated arguments are returned as a single object of mode `"call"`, but with class `"formula"`.

```
> get("~")
function(x, y)
{
        x = sys.call()
        oldClass(x) <- "formula"
        x
}
> f = log(y) ~ a + b/c | d
> f
log(y) ~ a + b/c | d
> mode(f)
[1] "call"
> oldClass(f) = NULL
> deparse(f)
[1] "log(y) ~ a + b/c | d"
> as.character(f)
[1] " ~ "         "log(y)"      "a + b/c | d"
```

Formulae provide another way of quoting arguments, giving the result class `formula`. `Quote` is similar but gives a result of class (and mode) `call`.

Symbolic differentiation

The `D` function is the basic differentiation function.

```
> v = Quote(cos((x + y)^2))
> v
cos((x + y)^2)
> D(v, "y")
 - (sin((x + y)^2) * (2 * (x + y)))
```

Components of formulae can also be differentiated.

```
> form = ~ b0 + b1*exp(-x/theta)
> D(form[[2]], "theta")
b1 * (exp( - x/theta) * x/theta^2)
```

`make.call` is an ancillary function used by `D`.

Some code excerpts from `D` give the flavour:

```
> D
function(expr, name)
switch(mode(expr),
  logical = ,
  numeric = ,
  complex = 0,
  name = if(as.character(expr)==name) 1 else 0,
  expression = D(expr[[1]], name),
  "(" = D(expr[[2]], name),
  call = switch(as.character(expr[[1]]),
    "+" = switch(length(expr),
      ,
      D(expr[[2]], name),
        make.call("+", D(expr[[2]], name),
                          D(expr[[3]], name))),
....
exp = make.call("*", expr, D(expr[[2]], name)),
....
atan = make.call("/", D(expr[[2]], name),
         make.call("+", 1, make.call("^",
                               expr[[2]], 2))),
pnorm = make.call("*", make.call("dnorm",
               expr[[2]]), D(expr[[2]], name)),
....
```

If functions carried with them an expression for differentiation as an attribute, this function would be much simpler and self extending.

A practical problem: The Jacobian of a transformation

Given a series of functions $f_1(x, y, \ldots)$, $f_2(x, y, \ldots)$, …, the Jacobian matrix is the matrix of partial derivatives

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \cdots \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

```
> jacobian(~x^2+y^2, ~atan(x/y))
                                x :
x^2 + y^2 :                   2 * x
atan(x/y) :        1/y/(1 + (x/y)^2)
                                y :
x^2 + y^2 :                   2 * y
atan(x/y) :  - (x/y^2/(1 + (x/y)^2))
```

(Non-standard `print` method.)

```
> jacobian(~x^2+y^2, ~atan(x/y), x = 1, y = 2)
            x    y
x^2 + y^2 2.0  4.0
atan(x/y) 0.4 -0.2
```

The workhorse is a function, `.jacobian`

```
".jacobian" <- function(formulae, variables = NULL) {
  if(is.null(variables))
    variables = all.vars(formulae)
  switch(mode(variables),
         numeric = {
           values = as.list(variables)
           variables = names(values)
         },
         list = {
           values = variables
           variables = names(variables)
         },
         character = {
           values <- NULL
         },
         stop("invalid ‘variables’"))
  nf = length(formulae)
  nv = length(variables)
  fnames = names(formulae)
  if(is.null(fnames))
    fnames = unlist(lapply(formulae, deparse))
  if(any(i <- fnames == ""))
    fnames[i] = sapply(formulae[i], deparse)
  if(is.null(values)) {
    J = list()
    for(i in 1:nf) {
      tmp = list()
      for(j in 1:nv)
        tmp[[variables[j]]] = D(formulae[[i]],
                                variables[j])
      J[[fnames[i]]] = tmp
    }
    oldClass(J) = "jacobian"
    return(J)
  } else {
    M = matrix(0, nf, nv,
      dimnames = list(fnames, variables))
    for(i in 1:nf)
      for(j in 1:nv)
        M[i, j] = eval(D(formulae[[i]], variables[j]),
                       local = values)
    M
  }
}
```


Notes

- `funs` is an expression with the functions as components.

  `vars` is either a character vector of names, (leading to a list of lists of calls as the result) or a numeric vector with named components (leading to a numeric matrix result).

- Given `D` this function is largely housekeeping.

- `.jacobian` is probably not to be called directly by users (very often).

Two front-end functions to `.jacobian` are now easy to write as more convenient interfaces.

```
jacobian = function(...) {
  m = list(...)
  l = unlist(lapply(m, is.list))
  f = unlist(lapply(m, inherits, "formula"))
  formulae = c(unlist(m[l], recur = F), m[f])
  formulae = as.expression(lapply(formulae,
      function(x) x[[length(x)]]))
  variables = unlist(m[!(l | f)])
  .jacobian(formulae, variables)
}

print.jacobian = function(x, ...) {
  rjustify = function(str) {
    m = max(n <- nchar(str))
    blanks = paste(rep(" ", m),
      collapse = "")
    paste(substring(blanks, 0, m - n), str,
          sep = "")
  }
  j = lapply(x, function(y)
            lapply(y, deparse))
  M = matrix(rjustify(as.vector(unlist(j))),
    length(j), length(j[[1]]),
    dimnames = list(paste(names(j[[1]]), ":"),
      paste(names(j), ":")))
  print(t(M), quote = F)
  invisible(x)
}
```


Notes

- `jacobian` allows functions to be specified as (one sided) formulae and variables as (named) numeric arguments or as numeric vector arguments (with names). (See example).
- `print.transf` allows succinct printing of transformation objects.
- `as.list` (and *sometimes* `as(, "list")`) in S4 convert a named object into a named list (if possible). In S3 the names were discarded.
- By convention `print` methods always return the object (invisibly!) so that the object `.Last.value` is correctly maintained.
- Silent display is now done by `show` rather than `print` but for objects without a specific method for `show` the print method is still used.

Transformation objects and transforms of them

```
setClass("transf",
   representation(beta = "named", SE = "named",
                  Vcov = "matrix"))

print.transf = function(x, ...) {
  print(data.frame(beta = x@beta, SE = x@SE))
  invisible(x)
}

setMethod("show", "transf",
         function(object) print.transf(object))

setGeneric("transf", function(theta, Vcov, ...)
           standardGeneric("transf"))

setDefaultMethod("transf",
                 function(theta, Vcov, ...) {
  m = as.expression(substitute(f(...))[-1])
  J = .jacobian(m, theta)
  theta = as.list(theta)
  beta = unlist(lapply(m, eval, local = theta))
  Vb = J %*% Vcov %*% t(J)
  SE = sqrt(diag(Vb))
  names(SE) = names(beta)
  new("transf", beta = beta, SE = SE, Vcov=Vb)
})
```


Transformation objects have *slots* for a named vector of coefficients and a variance matrix. In addition they contain the standard errors as an extra *slot*.

The function `trans` is generic and generates a transformation object in a fairly standard way.

What if we want to transform the transforms?

The information is in the object to allow this, so we need only write a *method* for calls to `transf` with this *signature*.

```
setMethod("transf", signature(theta="transf"),
         function(theta, Vcov, ...) {
           m = match.call()
           V = theta@Vcov
           beta = theta@beta
           m$theta = Quote(beta)
           m$Vcov = Quote(V)
           eval(m)
         })
```

Example

Binomial regression models:

$$p(x) = \Phi\left(\frac{x-\mu}{\sigma}\right) = \Phi(\beta_1 + \beta_2 x)$$

Back-transformation:

$$\begin{aligned}\mu &= -\beta_1/\beta_2 \\ \sigma &= 1/\beta_2\end{aligned}$$

```
> kalythos
  Age  N  B
1  20 50  6
2  35 50 17
3  45 50 26
4  55 50 37
5  70 50 44
```

Fitting a probit model, extracting coefficients and estimate of variance matrix:

```
> kaly1 = glm(B/N ~ Age, binomial(probit),
                          kalythos, weight = N)
> beta = structure(coef(kaly1),
                         names = c("b1", "b2"))
> beta
      b1       b2
 -2.1023 0.048147
> V = vcov(kaly1)
```


Calculating estimates of μ and σ:

```
> mu.sig = transf(beta, V,
                       mu = -b1/b2, sig = 1/b2)
> mu.sig
      beta     SE
 mu 43.663 1.8448
sig 20.770 2.5388
```

A series of percentile cutoffs in the distribution of the age at blindness:

```
> c0 = qnorm(0.7)
> c1 = qnorm(0.9)
> cutoffs = transf(mu.sig,
      p10 = mu - c1*sig,
      p30 = mu - c0*sig,
      p50 = mu,
      p70 = mu + c0*sig,
      p90 = mu + c1*sig)
> cutoffs
      beta     SE
p10 17.046 3.8238
p30 32.772 2.3312
p50 43.663 1.8448
p70 54.555 2.2174
p90 70.281 3.6547
```

Presentation 4.

Design and Implementation
of a Polynomial Class

- Structure of polynomial objects.
- Constructor function and coercion.
- Operations on polynomials.
- Group methods (are NBD).
- Applications and examples.

Problem: finding a polynomial with given zeros in power series form.

1. The direct way:

$$\begin{aligned} P(x) &= \prod_{i=1}^{p}(x-\xi_i) \\ &= (-1)^p(\xi_1\xi_2\ldots\xi_p) + \cdots + \\ &\quad (\sum_{i<j}\xi_i\xi_j)x^{p-2} - (\sum_i \xi_i)x^{p-1} \\ &\qquad + x^p \end{aligned}$$

2. The recursive (or iterative) way:

$$\begin{aligned} P_0(x) &= 1 \\ P_{k+1}(x) &= (x-\xi_{k+1})P_k(x) \\ &= xP_k(x) - \xi_{k+1}P_k(x) \end{aligned}$$

Finding the coefficients of a monic polynomial with given zeros.

1. The iterative way.

```
poly.from.zeros = function(zeros) {
    p = 1
    for(xi in zeros)
      p = c(0, p) - c(xi * p, 0)
    p
}
```

1a. A slick (look, no braces!) recursive way:

```
  poly.from.zeros1a = function(z)
      if(!length(z)) 1 else
        (function(x, z) c(0, x) - z * c(x, 0))(
          Recall(z[-1]), z[1])
```


2. The “obvious” way.

```
poly.from.zeros2 = function(zeros)
  c(unlist(lapply(sapply(
      length(zeros):1, function(x, z, s)
        s(length(z), x, z), zeros,
          function(n, k, set)
          if(k == 0) numeric(0) else
          if(k == n) matrix(set[1:n], nrow = 1)
          else rbind(cbind(set[1],
            Recall(n - 1, k - 1, set[-1])),
                   Recall(n - 1, k, set[-1]))),
            function(x) (-1)^ncol(x) *
                  sum(apply(x, 1, prod)))), 1)
```

No ‘explicit loops’, no braces but obfuscated in the extreme and utterly impractical.

Some timings.

```
> resources(poly.from.zeros(-5:5))
  CPU Elapsed % CPU Child Cache Working
 0.05    0.05   100     0     0   23444

> resources(poly.from.zeros1a(-5:5))
  CPU Elapsed % CPU Child Cache Working
 0.13    0.13   100     0     0  350832

> resources(poly.from.zeros2(-5:5))
    CPU Elapsed % CPU Child   Cache Working
 110.75  110.94  99.8     0 1280784  248448
 59.367      66  89.9     0       0  405720 S+3
```


Lagrange interpolation

Problem: given (x_i, y_i) find $P(x)$ such that $P(x_i) = y_i$.

Solution: Put

$$L_i(x) = \prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)}$$

then

$$P(x) = \sum_{i=1}^{p} y_i L_i(x)$$

```
poly.from.values = function(x, y) {
  p = 0
  for(i in seq(along = x)) {
    Li = poly.from.zeros(x[ -i]))/
             prod(x[i] - x[ -i]
    p = p + y[i] * Li
  }
  p
}
```

$\boxed{\texttt{poly.calc.coef}}$ is a function that acts like `poly.from.zeros` when given one argument, and `poly.from.values` when given two.

Polynomial multiplication

Put $A(x) = \sum_{i=0}^{p} a_i x^i$, $B(x) = \sum_{i=0}^{q} b_i x^i$ and $C(x) = A(x)B(x) = \sum_{i=0}^{p+q} c_i x^i$.

Then $\quad c_k = \sum_{i+j=k} a_i b_j, \quad k = 0, 1, \ldots, (p+q)$

First the 'slick' way:

```
> poly.multiply = function(p, q)
  if(length(q)==1 || length(p)==1)  p*q else {
    unlist(lapply(split(outer(p, q),
    outer(1:length(p),1:length(q), "+")), sum))
}
```

A variant uses more memory:

```
> poly.multiply1 = function(p, q)
  if(length(q)==1 || length(p)==1)  p*q else {
    unlist(lapply(split(m <- outer(p, q),
      row(m) + col(m)), sum))
}
```


Since this is the *convolution* operation of two vectors we could make it a binary operator.
Now consider an obvious loop way:

```
"%+%" = function(p, q) {
  if(length(p) < length(q)) {
    r = q;    q = p;    p = r
  }
  r = rep(0, length(p) + length(q) - 1)
  j = 0:(length(p) - 1)
  for(i in 1:length(q))
    r[j] = r[j <- j + 1] + q[i] * p # <- needed
  r
}

> p = 1:100
> r = 1:200
> resources(poly.multiply(p, r))
   CPU Elapsed % CPU Child Cache Working
 14.01   14.31  97.9     0 26624  371324
> resources(poly.multiply1(p, r))
   CPU Elapsed % CPU Child Cache Working
 15.11   15.19  99.5     0 33280  371324
> resources(p %+% r)
  CPU Elapsed % CPU Child Cache Working
 1.06     1.1  96.4     0  9984   38380
```

The loop wins hands down!

Polynomial division

Essentially *whole value* division.

A neat (if indulgent) recursive solution is possible:

```
> poly.divide = function(p, q)
{
  ln = length(p)
  ld = length(q)
  if(ln < ld)     return(0)
  r = p[ln]/q[ld]
  if(ln == ld)    return(r)
  i = (ln - ld) + 1:ld
  p[i] = p[i] - r * q
  c(Recall(p[-ln], q), r)
}
> poly.divide(1:10 %+% 1:200, 1:200)
 [1]  1  2  3  4  5  6  7  8  9 10
```


The more conventional iterative solution:

```
.dividePolynomial = function(p, q) {
  lq = length(q)
  if(lq == 1)
    return(p/q)
  p = rev(p)
  q = rev(q)
  r = rep(0, length(p))
  i = 0
  while(length(p) >= length(q)) {
    i = i + 1
    d = p[1]/q[1]
    r[i] = d
    p[1:lq] = p[1:lq] - d * q
    p = p[-1]
  }
  if(i == 0) 0 else r[i:1]
}
> .dividePolynomial(1:200 %+% 1:10, 1:200)
 [1]  1  2  3  4  5  6  7  8  9 10
> resources(poly.divide(1:10 %+% 1:200, 1:200))
  CPU Elapsed % CPU Child Cache Working
 0.36    0.36   100     0  3328   99348
> resources(.dividePolynomial(1:10 %+% 1:200,
                                        1:200))
  CPU Elapsed % CPU Child Cache Working
 0.34    0.34   100     0  6656   61592
```

Both assume the divisor polynomial is not 0.

Polynomial objects in S (mostly S4)

- univariate polynomials only, and always in the same variable, (effectively).
- represented as a numeric vector of coefficients of length $n + 1$ ($n =$ degree), in *power series* form without holes,
- belonging to class `"polynomial"`
- we allow numbers to be used in 'polynomial arithmetic' to stand for constant polynomials.

A class and constructor function

```
setClass("polynomial",
         representation(.Data = "numeric"),
         validity = function(object) {
           if(!is(object@.Data, "numeric"))
             return("bad coefficient vector")
           if(any(is.na(object@.Data) |
                         is.inf(object@.Data)))
            return("NAs in coefficient vector")
           ld = length(object@.Data)
           if(ld == 1) return(TRUE)
           if(object@.Data[ld] == 0)
             return("zero leading coefficient")
           TRUE
         }
)

clip = function(p) {
  p = as(p, "numeric")
  while((l = length(p)) > 1 &&
        p[l] == 0) p = p[-l]
  p
}

polynomial = function(p = c(0, 1))
  new("polynomial", .Data = clip(p))
```

Ensures leading coefficient is not zero, unless the polynomial is constant.

Show and tell, coercion to character.

```
make.poly.char = function(p) {
  lp = length(p) - 1
  names(p) = 0:lp
  tap = as(abs(p), "character")
  if(any(toss <- match(tap, c("0", "-0"),
                       nomatch = F))) {
    p = p[!toss]
    if(length(p) == 0)
      return("0")
    tap = tap[!toss]
  }
  sp = c("- ", "+ ")[(p > 0) + 1]
  np = names(p)
  p = tap
  p[p == "1" & np != "0"] = ""
  pow = paste("x", "^", np, sep = "")
  pow[np == "0"] = ""
  pow[np == "1"] = "x"
  stars = rep("*", length(p))
  stars[p == "" | pow == ""] = ""
  p = paste(sp, p, stars, pow, sep = "",
          collapse = " ")
  while((ch = substring(p, 1, 1)) == "+" ||
        ch == " ") p = substring(p, 2)
  p
}
```



```
setIs("polynomial", "character", coerce =
      function(object)
      make.poly.char(object@.Data))
```

Polynomials are turned automatically into character strings when used as such.

```
print.polynomial = function(x, ...) {
  p = make.poly.char(zapsmall(x@.Data))
  ow = max(35, options("width")$width - 3)
  m2 = 0
  np = nchar(p)
  while(m2 < np) {
    m1 = m2 + 1
    m2 = m2 + ow
    if(m2 < np)
      while(m2 > m1 &&
       substring(p, m2, m2) != " ") m2 = m2 - 1
    cat("# ", substring(p, m1, m2), "\n")
  }
  invisible(x)
}

setMethod("show", "polynomial",
     function(object) print.polynomial(object))
```

Some first examples

```
> polynomial()
#  x

> polynomial(1:4)
#  1 + 2*x + 3*x^2 + 4*x^3

> ZeroPoly
function(x)
polynomial(poly.calc.coef(x))

> ZeroPoly(0:3)
#  - 6*x + 11*x^2 - 6*x^3 + x^4

> Lagrange
function(x, y)
polynomial(poly.calc.coef(x, y))

> Lagrange(0:3, (0:3)^3 + 2)
#  2 + x^3
```

Polynomial with ugly coefficients will still look ugly.

Arithmetic on polynomials

The arithmetic operators, `+`, `-`, `*`. &c are all generic functions of two arguments, `e1` and `e2` for which methods to handle polynomial arguments can be supplied in the usual way.

It is sometimes easier, though, to supply a single method for the whole generic *group* to which the operators belong.

The smallest such group is the `"Arith"` group:

```
> getGroupMembers("Arith")
[1] "+"   "-"   "*"   "^"   "%%"  "%/%" "/"
```

We will ignore the logical operators and the comparison operators for now.

A workhorse for polynomial arithmetic

```
doArithPolynomial = function(e1, e2) {
  e1 = unlist(unclass(e1))
  e2 = unlist(unclass(e2))
  e1.op.e2 = switch(.Generic,
    "+" = , "-" = {
      l = length(e1) - length(e2)
      e1 = c(e1, rep(0, max(0,  - l)))
      e2 = c(e2, rep(0, max(0, l)))
      callGeneric(e1, e2)
    },
    "*" = .multiplyPolynomial(e1, e2),
    "/" = ,
    "%/%" = .dividePolynomial(e1, e2),
    "%%" = .remainderPolynomial(e1, e2),
    "^" = if(length(e2) > 1 || e2 < 0 ||
      (e2 %% 1) != 0) {
      stop("positive integer powers only")
    } else {
      switch(as.character(e2),
             "0" = 1, "1" = e1,
             {
               m = e1
               for(i in 2:e2)
                 m = .multiplyPolynomial(m, e1)
               m
             })
    })
  polynomial(e1.op.e2)
}
```


Setting methods

```
setMethod("Arith", signature(e1 = "polynomial",
                             e2 = "missing"),
          function(e1, e2) {
            e1 = e1@.Data
            polynomial(callGeneric())
          })

setMethod("Arith", signature(e1 ="polynomial",
                             e2 ="polynomial"),
          doArithPolynomial)

setMethod("Arith", signature(e1 ="numeric",
                             e2 ="polynomial"),
          doArithPolynomial)

setMethod("Arith", signature(e1 ="integer",
                             e2 ="polynomial"),
          doArithPolynomial)

setMethod("Arith", signature(e1 = "polynomial",
                             e2 = "numeric"),
          doArithPolynomial)

setMethod("Arith", signature(e1 = "polynomial",
                             e2 = "integer"),
          doArithPolynomial)
```

Polynomial arithmetic only admits polynomials, numerics and integers.

Making functions from polynomials

The most efficient way to evaluate a polynomial is in the 'Horner' partially factorized form. The first step is to write a helper function to unroll the horner calculation into a series of simple assignments, all enclosed in a single 'open brace' language construct.

```
makeBrace <- function(p) {
  p <- rev(p)
  m <- length(p)
  ex <- vector("expression", m + 2)
  ex[[1]] <- structure(expression(p, 0),
                       mode = "<-")
  for(i in 1:length(p))
    ex[[i+1]] <- substitute(p <- v + x * p,
                            list(v = p[i]))
  ex[[m + 2]] <- as("p", "name")
  mode(ex) <- "{" #}
  ex
}
> makeBrace(1:3)
{
        p <- 0
        p <- 3 + x * p
        p <- 2 + x * p
        p <- 1 + x * p
        p
}
```


Now we need only put the brace into the body of a suitable function shell:

```
Polynomial2Function <- function(object)
          substitute(function(x) body,
          list(body = makeBrace(object@.Data)))

setIs("polynomial", "function",
                  coerce = Polynomial2Function)
```

At this point, polynomials can be coerced to functions, but they can also be simply *used* as functions if need be.

More examples

```
> x = polynomial()
> x
#  x
> p = (x-2)^3
> p
#  - 8 + 12*x - 6*x^2 + x^3
> fp = as(p, "function")
> fp
function(x)
{
        p <- 0
        p <- 1. + x * p
        p <- -6. + x * p
        p <- 12. + x * p
        p <- -8. + x * p
        p
}
> integrate(fp, 0, 1)$integral
[1] -3.75
> fp(-4:3)
[1] -216 -125  -64  -27   -8   -1    0    1
> p(-4:3)
[1] -216 -125  -64  -27   -8   -1    0    1
> p(x+2)
#  x^3
> (p(x+2))(1:3)
[1]  1  8 27
```


The Hermite polynomials, a recurrence relation.

$$H_0(x) = 1, H_1(x) = x,$$
$$H_{n+1}(x) = xH_n(x) - nH_{n-1}(x)$$

```
> H = list(polynomial(1), x <- polynomial())
> for (n in 2:5)
           H[[n+1]] = x*H[[n]] - (n-1)*H[[n-1]]
> names(H) = paste("H", 0:5, sep = "")
> H
$H0:
#  1

$H1:
#  x

$H2:
#  - 1 + x^2

$H3:
#  - 3*x + x^3

$H4:
#  3 - 6*x^2 + x^4

$H5:
#  15*x - 10*x^3 + x^5
```

Other group generic methods

```
setMethod("Math", "polynomial",
 function(x) {
   switch(.Generic, ceiling =, floor =, trunc =
          polynomial(callGeneric(x@.Data)),
          stop(paste(.Generic,
               "not allowed on polynomials"))
   )
 })

setMethod("Math2", "polynomial",
          function(x, digits) {
            x = x@.Data
            polynomial(callGeneric())
          })


> p = polynomial(runif(4))
> p
#  0.9606592 + 0.93746*x + 0.0441019*x^2 +
#  0.7646185*x^3
> round(p, 2)
#  0.96 + 0.94*x + 0.04*x^2 + 0.76*x^3
>
```


Other methods

- Symbolic differentiation and integration of polynomial objects is easy.
- Zeros of polynomials `polyroot` can be used for a method for the `solve` generic.
- Extraction and replacement methods for `"["` would be useful for manipulation of coefficients.
- Summary method: find the stationary points, points of inflexion and zeros.
- Methods for `plot`, `lines` and `points` are obvious. If no $x-$range is specified, cover the real parts of the critical points, perhaps.