Efficient Programming in S

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

Presenters: Bill Venables and Brian Ripley

Software Examples

An electronic version of the software in this handout is available from

`ftp://arcola.stats.adelaide.edu.au/pub/Washington`

in files `SoftS3.q` *and* `SoftS4.q`.

1 Fundamentals of S programming

1.1 Polynomial integration functions

```
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
}

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
}

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
}

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
}
```

```
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
}
```

2 Large computations in S

2.1 Resource monitoring functions, S-PLUS 3.4 and later

```
mem.tally.report <- function(message) {
  value <- .C("S_tle_mem_tally_report",
    integer(2),
    integer(2),
    as.integer(2))[[2]]
  names(value) <- c("new database", "evaluation")
  if(!missing(message))
    attr(value, "message") <- message
  class(value) <- "mem.tally"
  value
}

mem.tally.reset <- function()
invisible(.C("S_tle_mem_tally_reset"))

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)
  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)
}
```

2.2 Envelope functions, reasonable and unreasonable

```
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))
  class(res) <- "envelope"
  res
}

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)
}

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)
}

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)
}
```

2.3 Large regression examples

```
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)
}

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))
}
```

3 Operations on the language itself

3.1 Order statistic density function generators

```
# 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))
}

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")))
```

3.2 Extended `D` and `make.call`

```
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))),
    "-" = switch(length(expr),
      ,
      make.call("-", D(expr[[2]], name)),
      make.call("-", D(expr[[2]], name), D(expr[[3]], name))),
    "*" = make.call("+", make.call("*", D(expr[[2]], name),
      expr[[3]]), make.call("*", expr[[2]], D(expr[[3]], name))),
    "/" = make.call("-", make.call("/", D(expr[[2]], name),
      expr[[3]]), make.call("/", make.call("*", expr[[2]],
      D(expr[[3]], name)), make.call("^", expr[[3]], 2))),
    "^" = ,
```

```
    "**" = switch(mode(expr[[3]]),
      logical = ,
      numeric = make.call("*", expr[[3]], make.call("*", D(expr[[
        2]], name), make.call("^", expr[[2]], expr[[3]] - 1))),
      make.call("+", make.call("*", make.call("^", expr[[2]],
        make.call("-", expr[[3]], 1)), make.call("*", expr[[3]],
        D(expr[[2]], name))), make.call("*", make.call("^",
        expr[[2]], expr[[3]]), make.call("*", make.call("log",
        expr[[2]]), D(expr[[3]], name))))),
    exp = make.call("*", expr, D(expr[[2]], name)),
    log = make.call("/", D(expr[[2]], name), expr[[2]]),
    cos = make.call("*", make.call("sin", expr[[2]]), make.call(
      "-", D(expr[[2]], name))),
    sin = make.call("*", make.call("cos", expr[[2]]), D(expr[[2
      ]], name)),
    tan = make.call("/", D(expr[[2]], name), make.call("^",
      make.call("cos", expr[[2]]), 2)),
    cosh = make.call("*", make.call("sinh", expr[[2]]), D(expr[[
      2]], name)),
    sinh = make.call("*", make.call("cosh", expr[[2]]), D(expr[[
      2]], name)),
    tanh = make.call("/", D(expr[[2]], name), make.call("^",
      make.call("cosh", expr[[2]]), 2)),
    atan = make.call("/", D(expr[[2]], name), make.call("+", 1,
      make.call("^", expr[[2]], 2))),
    sqrt = D(call("^", expr[[2]], 0.5), name),
    pnorm = make.call("*", make.call("dnorm",   # Extension
    expr[[2]]), D(expr[[2]], name)),
    dnorm = make.call("*", make.call("*",   # Extension
    make.call("-", expr[[2]]), expr), D(expr[[2]], name)),
    stop(paste("Function", expr[[1]],
      "is not listed in the derivatives table"))),
  NA)

make.call <- function(fcall, arg1, arg2) {
#
# file MASS/modified/D.q based on S-PLUS 3.2 code
# (C) 1993 StatSci Inc.
# Amended by W. N. Venables and B. D. Ripley.
# Changes (C) 1994-8  W. N. Venables and B. D. Ripley
# May be used only with a licensed copy of S-PLUS 3.2 or later.
#
  result <- switch(fcall,
    "+" = if(is.zero(arg1)) arg2 else if(is.zero(arg2))
      arg1
    else if(is.uminus(arg1))
      make.call("-", arg2, arg1[[2]])
    else if(is.uminus(arg2))
      make.call("-", arg1, arg2[[2]])
```

```
    else call("+", arg1, arg2),
    "-" = if(missing(arg2)) {
      if(is.zero(arg1))
        0
      else if(is.uminus(arg1))
        arg1[[2]]
      else call("-", arg1)
    }
    else {
      if(is.zero(arg2))
        arg1
      else if(is.zero(arg1))
        make.call("-", arg2)
      else if(is.uminus(arg1))
        make.call("-", make.call("+", arg1[[2]], arg2))
      else if(is.uminus(arg2))
        make.call("+", arg1, arg2[[2]])
      else call("-", arg1, arg2)
    }
    ,
    "*" = if(is.zero(arg1)) 0 else if(is.zero(arg2))
      0
    else if(is.one(arg1))
      arg2
    else if(is.one(arg2))
      arg1
    else if(is.uminus(arg1))
      make.call("-", make.call("*", arg1[[2]], arg2))
    else if(is.uminus(arg2))
      make.call("-", make.call("*", arg1, arg2[[2]]))
    else call("*", arg1, arg2),
    "/" = if(is.zero(arg1)) 0 else if(is.zero(arg2))
      NA
    else if(is.one(arg2))
      arg1
    else if(is.uminus(arg1))
      make.call("-", make.call("/", arg1[[2]], arg2))
    else if(is.uminus(arg2))
      make.call("-", make.call("/", arg1, arg2[[2]]))
    else call("/", arg1, arg2),
    "^" = if(is.zero(arg2)) 1 else if(is.zero(arg1))
      0
    else if(is.one(arg1))
      1
    else if(is.one(arg2))
      arg1
    else call("^", arg1, arg2),
# dummy to soak up all other cases
    if(missing(arg2)) call(fcall, arg1) else call(fcall, arg1,
```

```
      arg2))
  if(length(result) == 2 && is.atomic(result[[2]]) && result[[1
    ]] != "-")
    result <- eval(result)
  if(length(result) == 3 && is.atomic(result[[2]]) && is.atomic(
    result[[3]]))
    result <- eval(result)
  result
}
```

3.3 Jacobian and transformation functions for S4

```
".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
  }
}

jacobian <- function(...) {
  m = list(...)
  l = unlist(lapply(m, is.list))
  f = unlist(lapply(m, inherits, "formula"))
  formulae = c(unlist(m[l], recursive = 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)
}

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)
})

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)
         })
```

4 A preliminary version of an S4 polynomial class

```
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[1] == 0) p = p[-1]
  p
}

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

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())
          })

.dividePolynomial = function(p, q) {
  lq = length(q)
  if(lq == 1)
    return(p/q)
  p = rev(p)
  q = rev(q)
  r = numeric(0)
  while(length(p) >= length(q)) {
    d = p[1]/q[1]
    r = c(d, r)
    p[1:lq] = p[1:lq] - d * q
    p = p[-1]
  }
  if(length(r) == 0)
    r = 0
  r
}

.multiplyPolynomial = function(p, q) {
  if(length(q) == 1 || length(p) == 1)
    return(p * q)
  m = outer(p, q)
  m = unlist(lapply(split(m, row(m) + col(m)), sum))
  as.vector(m)
}
```

```
.remainderPolynomial = function(p, q) {
  lq = length(q)
  if(lq == 1)
    return(0)
  p = rev(p)
  q = rev(q)
  r = numeric(0)
  while(length(p) >= length(q)) {
    d = p[1]/q[1]
    r = c(d, r)
    p[1:lq] = p[1:lq] - d * q
    p = p[-1]
  }
  rev(p)
}

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)
}

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)

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
}

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

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

"poly.calc.coef" = function(x, y) {
  if(missing(y)) {
    ## Polynomial with specifice zeros
    p = 1
    for(xi in x)
      p = c(0, p) - c(xi * p, 0)
    return(p)
```

```
  }
  ## Lagrange interpolation formula
  if(any(toss = duplicated(x))) {
    y = y[!toss]
    x = x[!toss]
  }
  if((m = length(x)) != length(y))
    stop("length(x) != length(y)")
  if(m <= 1)
    return(y)
  r = 0
  for(i in 1:m)
    r = r + (y[i] * Recall(x[ - i]))/prod(x[i] - x[ - i])
  r
}

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

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

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))
```

```
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))

polyZeros <- function(z) {
        r = polyroot(z@.Data)
        zapsmall(Re(r)) + zapsmall(Im(r)) * (1i)
}

solve.polynomial <- function(a, b, ...)
  if(missing(b)) polyZeros(a) else polyZeros(a-b)
```