1 Generalized Deming regression

Deming regression solves the dual minimization

$$\min_{ab} \sum (y_i - \hat{y}_i)^2 + \sum (x - \hat{x})^2$$

The underlying model is that we have true values u and v with

$$\begin{aligned} u &= \alpha + \beta v \\ x_i &= u_i + \epsilon \\ y_i &= v_i + \delta \end{aligned}$$

where ϵ and δ are mean 0 errors. The ususal Deming method assumes that the variances of ϵ and δ are equal. For general Deming regression let them vary such that

$$\begin{aligned} \text{std}(\epsilon) &= \sigma * (c + du) \equiv \sqrt{\kappa_i} \\ \text{std}(\delta) &= \sigma * (e + fv) \equiv \sqrt{\lambda_i} \end{aligned}$$

Standard Deming regression corresponds to $d = f = 0$ and $c = e$, the constant coefficient model of Linnet to $d = f$ and $c = e = 0$.

Ripley and Thompsen (Analyst 1987, 377-383) work out the following equation with a full page of work. If the fitted line is $\alpha + \beta x$, and β is known then the optimal solution for α is

$$w_i = 1/(\lambda_i + \beta^2 \kappa_i)$$
$$\alpha = \frac{\sum w_i (y_i - \beta x_i)}{\sum w_i}$$

Prior case weights simply add a multiplicative factor to w. This allows the use of the optimize routine, which is a very fast for one dimensional maximization. As starting estimates, we know that the slope lies between the two least-squares regressions of x on y and of y on x. The code below allows for model without an intercept.

```
⟨demingfit-afun⟩=
 # The estimate of alpha-hat, given beta
 afun <-function(beta, x, y, wt, xv, yv) {
     w <- wt/(yv + beta^2*xv)
     sum(w * (y - beta*x))/ sum(w)
 }

 minfun <- function(beta, x, y, wt, xv, yv) {
     w <-  wt/(yv + beta^2*xv)
     alphahat <- sum(w * (y - beta*x))/ sum(w)
     sum(w* (y-(alphahat + beta*x))^2)
```

```
 }
 minfun0 <- function(beta, x, y, wt, xv, yv) {
     w <-  wt/(yv + beta^2*xv)
     alphahat <- 0  #constrain to zero
     sum(w* (y-(alphahat + beta*x))^2)
 }
```

⟨*deming.fit1*⟩=
```
 # Fit for fixed std values
 deming.fit1 <- function(x, y, wt, xstd, ystd, intercept) {
     ⟨demingfit-afun⟩
     if (intercept) {
         fit1 <- lm.wfit(cbind(1,x), y, wt/ystd)
         fit2 <- lm.wfit(cbind(1,y), x, wt/xstd)
         init <- sort(c(fit1$coef[2], 1/fit2$coef[2]))

         fit <- optimize(minfun, init, x=x, y=y, wt=wt,
                         xv=xstd^2, yv=ystd^2)
 browser()
         list(coefficients=c(afun(fit$minimum, x,y, wt=wt, xstd^2, ystd^2),
                             fit$minimum))
     }
     else {
         fit1 <- lm.wfit(x, y, wt/ystd)
         fit2 <- lm.wfit(y, x, wt/xstd)
         init <- sort(c(fit1$coef, 1/fit2$coef))

         fit <- optimize(minfun0, init, x=x, y=y, wt=wt,
                         xv=xstd^2, yv=ystd^2)
         list(coefficients=fit$minimum)
     }
 }
```

When called with a pattern argument the code is just a bit more complex, since the weights are updated as well. As starting estimates for the weights we use the data itself. the weights are not allowed to be negative. Given a tentative line $y = a + bx$ what is the closest point on the line to any given data point (x_i, y_i)? We want to find that point x such that $f(x)$ is minimal. Referring again to Ripley, the solution is

$$\hat{x}_i = w_i[\lambda x_i + \kappa\beta(y - \alpha)]$$

⟨*deming.fit2*⟩=
```
 # Fit when there is a pattern argument
 deming.fit2 <-  function(x, y, wt, stdpat, intercept,
                          tol=.Machine$double.eps^0.25) {
     ⟨demingfit-afun⟩
```

```
    xstd <- stdpat[1] + stdpat[2]*pmax(x,0)
    ystd <- stdpat[3] + stdpat[4]*pmax(y,0)
    if (sum(xstd>0) <2 || sum(ystd>0) <2)
        stop("initial weight estimates must be positive for at least 2 points")


    ifit <- deming.fit1(x, y, wt, xstd, ystd, intercept)
    if (intercept) {
        alpha <- ifit$coefficients[1]
        beta <- ifit$coefficients[2]
    }
    else {
        alpha <- 0
        beta <- ifit$coefficients
    }

    # 20 to stop any runaway failures.  usually 1-2 suffice.
    for(i in 1:20) {
        w <- 1/(ystd^2 + beta^2*xstd^2)
        newx <- w*(ystd^2*x + xstd^2* beta*(y- alpha))
        newy <- alpha + beta*newx
        xstd <- stdpat[1] + stdpat[2]*pmax(0, newx)
        ystd <- stdpat[3] + stdpat[4]*pmax(0, newy)

        if (intercept)
            fit <- optimize(minfun, c(.2, 5)*beta, x=x, y=y, wt=wt,
                        xv=xstd^2, yv=ystd^2)
        else fit <- optimize(minfun0,  c(.2, 5)*beta, x=x, y=y, wt=wt,
                        xv=xstd^2, yv=ystd^2)

        if (abs(fit$minimum - beta)/(abs(beta)+tol) <= tol) break
        beta <- fit$minimum
        if (intercept) alpha <- afun(beta, x, y, wt=wt, xstd^2, ystd^2)
    }

    if (intercept)
         list(coefficients=c(afun(fit$minimum, x,y, wt=wt, xstd^2, ystd^2),
                            fit$minimum))
    else list(coefficients=c(0, fit$minimum))
}
```

The main deming routine starts with the standard material for creating a data frame.

⟨*deming*⟩=
```
deming <-  function(formula, data, subset, weights, na.action,
                   ccv=FALSE, xstd, ystd, stdpat,
```

```
                  conf= .95, nboot=0, dfbeta=FALSE,
                  x=FALSE, y=FALSE, model=TRUE) {
    Call <- match.call()
    # create a call to model.frame() that contains the formula (required)
    #  and any other of the relevant optional arguments
    # then evaluate it in the proper frame
    indx <- match(c("formula", "data", "weights", "subset", "na.action",
                    "xstd", "ystd"),
                  names(Call), nomatch=0)
    if (indx[1] ==0) stop("A formula argument is required")
    temp <- Call[c(1,indx)]  # only keep the arguments we wanted
    temp[[1]] <- as.name('model.frame')  # change the function called
    mf <- eval(temp, parent.frame())
    Terms <- terms(mf)
    n <- nrow(mf)

    ⟨deming-check⟩
    ⟨deming-compute⟩
    ⟨deming-se⟩
    ⟨deming-finish⟩
 }

⟨deming.fit1⟩

⟨deming.fit2⟩

⟨deming.print⟩
```

Check all of the options for legality. Tedious but simple

```
⟨deming-check⟩=
if (n < 3) stop("less than 3 non-missing observations in the data set")

xstd <- model.extract(mf, "xstd")
ystd <- model.extract(mf, "ystd")
Y <- model.response(mf, type="numeric")
if (is.null(Y))
    stop ("a response variable is required")
wt <- model.weights(mf)
if (length(wt)==0) wt <- rep(1.0, n)

usepattern <- FALSE
if (is.null(xstd)) {
    if (!is.null(ystd))
    stop("both of xstd and ystd must be given, or neither")
    if (missing(stdpat)) {
        if (ccv) stdpat <- c(0,1,0,1)
```

```
        else     stdpat <- c(1,0,1,0)
    }
    else {
        if (any(stdpat <0) || all(stdpat[1:2] ==0) || all(stdpat[3:4]==0))
            stop("invalid stdpat argument")
    }
    if (stdpat[2] >0 || stdpat[4] >0) usepattern <- TRUE
    else {xstd <- rep(stdpat[1], n); ystd <- rep(stdpat[3], n)}
} else  {
    if (is.null(ystd))
        stop("both of xstd and ystd must be given, or neither")
    if (!is.numeric(xstd)) stop("xstd must be numeric")
    if (!is.numeric(ystd)) stop("ystd must be numeric")
    if (any(xstd <=0)) stop("xstd must be positive")
    if (any(ystd <=0)) stop("ystd must be positive")
}
if (conf <0 || conf>=1) stop("invalid confidence level")
if (!is.logical(dfbeta)) stop("dfbeta must be TRUE or FALSE")
```

Now do the computation. If the std is self referencing, i.e., either the ccv argument is used or the stdpat argument with nozero term for elements 2 and 4, then we need to use the iterative routine deming.fit2; otherwise we can use the simpler one.

```
⟨deming-compute⟩=
X <- model.matrix(Terms, mf)
if (ncol(X) != (1 + attr(Terms, "intercept")))
    stop("Deming regression requires a single predictor variable")
xx <- X[,ncol(X)]  #actual regressor

if (!usepattern)
    fit <- deming.fit1(xx, Y, wt, xstd, ystd,
                       intercept= attr(Terms, "intercept"))
else
    fit <- deming.fit2(xx, Y, wt, stdpat,
                      intercept= attr(Terms, "intercept"))
names(fit$coefficients) <- dimnames(X)[[2]]
yhat <- fit$coefficients[1] + fit$coefficients[2]*xx
fit$residuals <- Y-yhat
```

Jackknife or bootstrap estimates of error

```
⟨deming-se⟩=
if (nboot > 0) {
    # Compute a bootstrap estimate of variance
    stop("bootstrap not yet done")
}
else if (conf>0) {
```

```
    # jackknife it
    delta <- matrix(0., nrow=n, ncol=2)
    for (i in 1:n) {
        if (usepattern)
            tfit <-deming.fit2(xx[-i], Y[-i], wt[-i], stdpat,
                                   intercept= attr(Terms, "intercept"))
        else
            tfit <-deming.fit1(xx[-i], Y[-i], wt[-i], xstd[-i], ystd[-i],
                                   intercept= attr(Terms, "intercept"))

        delta[i,] <- fit$coefficients - tfit$coefficients
        fit$variance <- t(delta) %*% delta
        if (dfbeta) fit$dfbeta <- delta
    }
    z <- -qnorm((1- conf)/2)
    se <- sqrt(diag(fit$variance))
    ci <- cbind(fit$coefficients - z*se,
                fit$coefficients + z*se)
    dimnames(ci) <- list(names(fit$coefficients),
                         paste(c("lower", "upper"), format(conf)))
    fit$ci <- ci
}
```

And last, all the little bits for cleaning up

```
⟨deming-finish⟩=
if (x) fit$x <- X
if (y) fit$y <- Y
if (model) fit$model <- mf

na.action <- attr(mf, "na.action")
if (length(na.action)) fit$na.action <- na.action
fit$n <- length(Y)
fit$terms <- Terms
class(fit) <- "deming"
fit$call <- Call
fit
```

```
⟨deming.print⟩=
print.deming <- function(x, ...) {
    cat("\nCall:\n", deparse(x$call), "\n\n", sep = "")
    cat("n=", x$n)
    if (length(x$na.action))
        cat("  (", naprint(x$na.action), ")\n", sep='')
    else cat("\n")

    if (!is.null(x$ci)) {
```

```
    table <- matrix(0., nrow=2, ncol=4)
    table[,1] <- x$coefficients
    if (is.null(x$variance))  table[,2] <- x$std
    else table[,2] <- sqrt(diag(x$variance))
    table[,3:4] <- x$ci

    dimnames(table) <- list(c("Intercept", "Slope"),
                            c("Coef", "se(coef)", dimnames(x$ci)[[2]]))
}
else {
    table <- matrix(0., nrow=2, ncol=2)
    table[,1] <- x$coefficients
    if (is.null(x$variance))  table[,2] <- x$std
    else table[,2] <- sqrt(diag(x$variance))

    dimnames(table) <- list(c("Intercept", "Slope"),
                            c("Coef", "se(coef)"))
}
print(table, ...)
cat("\n   Scale=", format(x$scale), "\n")
invisible(x)
}
```