Exceedingly Simple Monotone Regression (with Ties)

Jan de Leeuw

Version 01, April 01, 2017

Abstract

A C implementation of Kruskal's up-and-down-blocks monotone regression algorithm for use with .C() is extended to include the three classic ways of handling ties. It is then compared with other implementations.

Contents

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory deleeuwpdx.net/pubfolders/jbkTies has a pdf version, the bib file, the complete Rmd file with the code chunks, and the R source code.

1 Introduction

In a recent Rpub (De Leeuw (2017a)) we presented a `.C()` version of Kruskal's up-and-down-blocks algorithm for monotone (or isotone) regression. In De Leeuw (2016) there is R code that extends monotone regression by adding the three classical ways of handling ties. See Kruskal (1964) for the primary and secondary approach, and De Leeuw (1977) for the tertiary approach, as well as for proofs that all three methods are indeed optimal. The R code for tie handling in De Leeuw (2016) uses lists for blocks of ties, and includes a number of loops over blocks. Thus there is some room for improvement by implementing tie-handling using `.C()`.

The appendix of this follow-up paper presents `.C()` code for monotone regression with the three approaches. We have chosen to make it modular, in the sense that the basic operations all have a separate routine written in C, with correspondng R wrapper. All C routines are written using `.C()` conventions, i.e. they pass by reference and return a void. Internally they do not use any R-based functions and include files, so they can be easily used in other contexts as well. Thus extra storage need in the routines is allocated and freed internally, without involving R.

2 Small Example

```
x <- c(2.1, 2.1, 3.5, 1.9, 3.5, 3.5, 1.9, 2.1, 1.9)
y <- c(2, 1, 6, 5, 4, 7, 8, 9, 3)
w <- c(1, 1, 2, 2, 2, 2, 2, 1, 1)
```

2.1 Sorting

We use the function `mySortDouble()`, a `.C()` interface to a C routine of the same name, to sort x and to put y and w in the correct order. In R parlance, the routine returns `order(x)` as well as `sort (x) = x[order (x)]` , `y[order (x)]` and `w[order (x)]`. The sorting routine is C code taken from De Leeuw (2017b). It uses the system `qsort()` routine, which implements quicksort. Quicksort is not a stable sorting routine, in the sense that it does not guarantee that tieblocks will be sorted in a unique way. This, however, is not a problem in the monotone regression context, because that form of instability will not change the outcome of the regression routine.

```
## [1] 1.9 1.9 1.9 2.1 2.1 2.1 3.5 3.5 3.5
```

```
## [1]   8   3   5   2   1   9   6   7   4
```

```
## [1]   2   1   2   1   1   1   2   2   2
```

```
## [1]   7   9   4   1   2   8   3   6   5
```

The function `tieBlock()`, again a `.C()` interface to a C routine of the same name, uses the sorted x to return the tie-blocks (a vector of the same length as x, using integers to indicate block membership).

```
## [1]   1   1   1   2   2   2   3   3   3
```

We emphasize that in a non-metric multidimensional scaling context (or, more generally, an alternating least squares context) the sorting only has to be done only once, not in every iteration. Thus in repeated calls of the algorithm this first part can be skipped.

2.2 Primary Approach

The primary approach to ties starts with sorting y within blocks. This is done with `sortBlocks()`, again a `.C()` interface. Along with y we sort w and the order vector.

2.2.1 Sort within Blocks

```
## [1]   3   5   8   1   2   9   4   6   7
```

```
## [1]   1   2   2   1   1   1   2   2   2
```

```
## [1]   9   4   7   2   1   8   5   3   6
```

The newly sorted y and w are entered into `jbkPava()`, the monotone regression routine, again a `.C()` interface to C code.

2.2.2 Monotone Regression

```
## [1] 3.000 4.833 4.833 4.833 4.833 5.667 5.667 6.000 7.000
```

2.2.3 Restore Order

Finally `mySortInteger()` ,another `.C()` interface, is used to find the inverse permutation that puts the monotone regression output back in the correct order.

```
## [1]   5   4   8   2   7   9   3   6   1
```

```
## [1] 4.833 4.833 6.000 4.833 5.667 7.000 4.833 5.667 3.000
```

2.2.4 Residual Sum of Squares

Equal to 46.6666666667.

2.3 Secondary Approach

2.3.1 Make Blocks

In the secondary approach we use `makeBlocks()`, another `.C()` interface, to create block averages and block weights.

```
## [1] 5.800 4.000 5.667
```

```
## [1] 5.000 3.000 6.000
```

2.3.2 Monotone Regression

Then we perform monotone regression on the block values using block weights.

```
## [1] 5.125 5.125 5.667
```

2.3.3 Expand and Restore Order

Finally we use `mySortInteger()`, another `.C()` sorting routine, to expand the monotone regression values in the correct order to a vector of `length (x)`.

```
## [1] 5.125 5.125 5.667 5.125 5.667 5.667 5.125 5.125 5.125
```

2.3.4 Residual Sum of Squares

Equal to 59.2604166667.

2.4 Tertiary Approach

2.4.1 Adjust Means

The tertiary approach forms blocks and does a monotone regression as in the secondary approach. It then adjusts the block means optimally.

```
## [1] 7.325  2.325  4.325  3.125  2.125  10.125 6.000  7.000  4.000
```

2.4.2 Restore Order

```
## [1] 3.125  2.125  6.000  4.325  4.000  7.000  7.325  10.125 2.325
```

2.4.3 Residual Sum of Squares

Equal to 5.16375.

3 Timings

We compare two different implementations of monotone regression with ties. The first is `monregR()`, which does the data-handling and tie-handling in R, using lists. It is basically the code from De Leeuw (2016), with two changes. In the first place the Fortran code for monotone regression from Cran (1980) is replaced by the C code from De Leeuw (2017a). Secondly, the R code in De Leeuw (2016) does not work correctly in the case of unequal weights, and we have corrected that in the current version of `monregR()`. The second function is `monregRC()`, which does data-handling, tie-handling, and monotone regression in C, using the functions we have illustrated earlier in our small example.

In our comparion we use

```
x <- sample (1:blks, n, replace = TRUE)
y <- rnorm (n)
```

where n = 10,000. Since we sample with equal probabilities from `1:blks` the vector x will have (at most) blks different values, so if blks is small there will be many ties, if blks is large there will be only a few ties. We let blks take the values `c(2,10,100,1000,10000)`. Combined with the three options for tie-handling this gives 15 different analyses, and each one is repeated 100 times. We report the elapsed time from `system.time()`.

Results for `monregR()` are

```
##       1       2       3
## 2     0.5570  0.1820  0.1880
## 10    0.3300  0.2330  0.2280
## 100   1.0480  0.7280  0.7460
## 1000  7.7320  5.8450  5.9150
## 10000 49.1820 34.6100 37.7510
```

Results for `monregRC()` are

```
##       1      2      3
## 2     1.1050 0.8130 0.7580
## 10    0.8320 0.7230 0.7220
## 100   0.7010 0.6070 0.6150
## 1000  0.5100 0.4410 0.4660
## 10000 0.4930 0.4630 0.4720
```

Choosing n = 10,000 is similar to what we used in De Leeuw (2017a), but in a typical multidimensional scaling context n = 500 is perhaps more realistic.

Now the results for `monregR()` are

```
##     1      2      3
## 2   0.0330 0.0150 0.0160
## 10  0.0570 0.0290 0.0310
## 100 0.2650 0.0950 0.1070
## 250 0.5820 0.1690 0.1740
## 500 0.7270 0.2380 0.2340
```

and those for `monregRC()` are

```
##     1      2      3
## 2   0.0390 0.0280 0.0320
## 10  0.0380 0.0320 0.0300
## 100 0.0260 0.0250 0.0250
## 250 0.0260 0.0240 0.0230
## 500 0.0280 0.0170 0.0200
```

4 Conclusion

Running time for `monregR()` heavily depends on the number of blocks, because the R code has loops over blocks. For a small number of blocks `monregR()` is even faster than `monregRC()`. If n= 10,000 then up to five times as fast for an expected block size of 5000 (two blocks). But for more blocks `monregR()` rapidly loses its advantage. For n = 500 we find that `monregRC()` is almost always faster, and usually much faster.

5 Code

5.1 R code

```
dyn.load("jbkTies.so")

mySortDouble <- function (x, y, w = rep(1, length(x))) {
  n <- length (x)
  xind <- rep (0, n)
  h <-
    .C(
      "mySortDouble",
      x = as.double (x),
      y = as.double (y),
      w = as.double (w),
      xind = as.integer (xind),
      n = as.integer(n)
    )
  return (h)
}

mySortInteger <- function (x) {
  n <- length (x)
  xind <- rep (0, n)
  h <-
    .C(
      "mySortInteger",
      x = as.integer (x),
      xind = as.integer (xind),
      n = as.integer(n)
    )
  return (h)
}

tieBlock <- function (x) {
  n <- length (x)
  h <-
    .C(
      "tieBlock",
      x = as.double (x),
      iblks = as.integer (rep(0, n)),
      as.integer (n),
      nblk = as.integer (0)
```

```
    )
  return (h)
}

makeBlocks <- function  (x, w = rep (1, length(x)), iblks) {
  n <- length (x)
  nblk <- max (iblks)
  xblks <- rep (0, nblk)
  wblks <- rep (0, nblk)
  h <-
    .C(
      "makeBlocks",
      x = as.double (x),
      w = as.double (w),
      xblks = as.double (xblks),
      wblks = as.double (wblks),
      iblks = as.integer (iblks),
      n = as.integer(n),
      nblk = as.integer (nblk)
    )
  return (h)
}

sortBlocks <- function (y, w, xind, iblks) {
  n <- length (y)
  nblk <- max (iblks)
  h <- .C(
    "sortBlocks",
    y = as.double (y),
    w = as.double (w),
    xind = as.integer (xind),
    as.integer(iblks),
    as.integer (n),
    as.integer (nblk)
  )
  return (h)
}

jbkPava <- function (x, w = rep(1, length(x))) {
  h <-
    .C("jbkPava",
       x = as.double(x),
       w = as.double (w),
       n = as.integer(length(x)))
```

```
  return (h)
}

monregR <-
  function (x,
            y,
            w = rep (1, length (x)),
            ties = 1) {
    f <- sort(unique(x))
    g <- lapply(f, function (z)
      which(x == z))
    n <- length (x)
    k <- length (f)
    if (ties == 1) {
      h <- lapply (g, function (x)
        y[x])
      f <- lapply (g, function (x)
        w[x])
      m <- rep (0, n)
      for (i in 1:k) {
        ii <- order (h[[i]])
        g[[i]] <- g[[i]][ii]
        h[[i]] <- h[[i]][ii]
        f[[i]] <- f[[i]][ii]
      }
      r <- jbkPava (unlist(h), unlist(f))$x
      s <- r[order (unlist (g))]
    }
    if (ties == 2) {
      h <- lapply (g, function (x)
        y[x])
      f <- lapply (g, function (x)
        w[x])
      s <- sapply(1:k, function(i) sum (h[[i]] * f[[i]]))
      r <- sapply (f, sum)
      m <- s / r
      r <- jbkPava (m, r)$x
      s <- rep (0, n)
      for (i in 1:k)
        s[g[[i]]] <- r[i]
    }
    if (ties == 3) {
      h <- lapply (g, function (x)
        y[x])
```

```
      f <- lapply (g, function (x)
        w[x])
      s <- sapply(1:k, function(i) sum (h[[i]] * f[[i]]))
      r <- sapply (f, sum)
      m <- s / r
      r <- jbkPava (m, r)$x
      s <- rep (0, n)
      for (i in 1:k)
        s[g[[i]]] <- y[g[[i]]] + (r[i] - m[i])
    }
    return (s)
  }

monregRC <- function (x,
                      y,
                      w = rep(1, length(x)),
                      ties = 1) {
  a <- mySortDouble(x, y, w)
  b <- tieBlock (a$x)$iblks
  if (ties == 1) {
    c <- sortBlocks (a$y, a$w, a$xind, b)
    d <- jbkPava (c$y, c$w)$x
    e <- mySortInteger (c$xind)$xind
    return (d[e])
  }
  if (ties == 2) {
    c <- makeBlocks (a$y, a$w, b)
    d <- jbkPava (c$xblks, c$wblks)$x
    e <- mySortInteger (a$xind)$xind
    return (d[b][e])
  }
  if (ties == 3) {
    c <- makeBlocks (a$y, a$w, b)
    d <- jbkPava (c$xblks, c$wblks)$x
    u <- a$y - (c$xblks - d)[b]
    e <- mySortInteger (a$xind)$xind
    return (u[e])
  }
}
```

5.2 C code

```c
#include <math.h>
#include <stdlib.h>
#include <stdbool.h>

struct block {
    double value;
    double weight;
    int size;
    int previous;
    int next;
};

struct quadruple {
    double value;
    double result;
    double weight;
    int index;
};

struct triple {
    double value;
    double weight;
    int index;
};

struct pair {
    int value;
    int index;
};

int myCompDouble (const void *, const void *);
int myCompInteger (const void *, const void *);
void mySortDouble (double *, double *, double *, int *, const int *);
void mySortInteger (int *, int *, const int *);
void mySortInBlock (double *, double *, int *, int *);
void tieBlock (double *, int *, const int *, int *);
void makeBlocks (double *, double *, double *, double *, int *, const int *, const int *
void sortBlocks (double *, double *, int *, const int *, const int *, const int *);
void jbkPava (double *, double *, const int *);

int myCompDouble (const void *px, const void *py) {
```

```
    double x = ((struct quadruple *)px)->value;
    double y = ((struct quadruple *)py)->value;
    return (int)copysign(1.0, x - y);
}

int myCompInteger (const void *px, const void *py) {
    int x = ((struct pair *)px)->value;
    int y = ((struct pair *)py)->value;
    return (int)copysign(1.0, x - y);
}

void mySortInBlock (double *x, double *w, int *xind, int *n) {
    int nn = *n;
    struct triple *xi =
        (struct triple *) calloc((size_t) nn, (size_t) sizeof(struct triple));
    for (int i = 0; i < nn; i++) {
        xi[i].value = x[i];
        xi[i].weight = w[i];
        xi[i].index = xind[i];
    }
    (void) qsort(xi, (size_t)nn, (size_t)sizeof(struct triple), myCompDouble);
    for (int i = 0; i < nn; i++) {
        x[i] = xi[i].value;
        w[i] = xi[i].weight;
        xind[i] = xi[i].index;
    }
    free(xi);
    return;
}


void mySortDouble (double *x, double *y, double *w, int *xind, const int *n) {
    int nn = *n;
    struct quadruple *xi =
        (struct quadruple *) calloc((size_t) nn, (size_t) sizeof(struct quadruple));
    for (int i = 0; i < nn; i++) {
        xi[i].value = x[i];
        xi[i].result = y[i];
        xi[i].weight = w[i];
        xi[i].index = i + 1;
    }
    (void) qsort(xi, (size_t)nn, (size_t)sizeof(struct quadruple), myCompDouble);
    for (int i = 0; i < nn; i++) {
        x[i] = xi[i].value;
```

```
        y[i] = xi[i].result;
        w[i] = xi[i].weight;
        xind[i] = xi[i].index;
    }
    free(xi);
    return;
}

void mySortInteger (int *x, int *k, const int *n) {
    int nn = *n;
    struct pair *xi =
        (struct pair *) calloc((size_t) nn, (size_t) sizeof(struct pair));
    for (int i = 0; i < nn; i++) {
        xi[i].value = x[i];
        xi[i].index = i + 1;
    }
    (void) qsort(xi, (size_t)nn, (size_t)sizeof(struct pair), myCompInteger);
    for (int i = 0; i < nn; i++) {
        x[i] = xi[i].value;
        k[i] = xi[i].index;
    }
    free(xi);
    return;
}

void tieBlock (double *x, int *iblks, const int *n, int *nblk) {
    iblks[0] = 1;
    for (int i = 1; i < *n; i++) {
        if (x[i - 1] == x[i]) {
            iblks[i] = iblks[i - 1];
        } else {
            iblks[i] = iblks[i - 1] + 1;
        }
    }
    *nblk = iblks[*n - 1];
    return;
}

void makeBlocks (double *x, double *w, double *xblks, double *wblks, int *iblks, const i
    for (int i = 0; i < *nblk; i++) {
        xblks[i] = 0.0;
        wblks[i] = 0.0;
    }
    for (int i = 0; i < *n; i++) {
```

```
        xblks [iblks [i] - 1] += w[i] * x[i];
        wblks [iblks [i] - 1] += w[i];
    }
    for (int i = 0; i < *nblk; i++) {
        xblks[i] = xblks[i] / wblks[i];
    }
    return;
}

void sortBlocks (double *y, double *w, int *xind, const int *iblks, const int *n, const
    int *nblks = (int *) calloc((size_t) * nblk, sizeof(int));
    for (int i = 0; i < *n; i++) {
        nblks[iblks[i] - 1]++;
    }
    int k = 0;
    for (int i = 0; i < *nblk; i++) {
        int nn = nblks[i];
        (void) mySortInBlock (y + k, w + k, xind + k, &nn);
        k += nn;
    }
    free (nblks);
    return;
}

void jbkPava (double *x, double *w, const int *n) {
    struct block *blocks = calloc ((size_t) * n, sizeof(struct block));
    for (int i = 0; i < *n; i++) {
        blocks[i].value = x[i];
        blocks[i].weight = w[i];
        blocks[i].size = 1;
        blocks[i].previous = i - 1; // index first element previous block
        blocks[i].next = i + 1;     // index first element next block
    }
    int active = 0;
    do {
        bool upsatisfied = false;
        int next = blocks[active].next;
        if (next == *n) upsatisfied = true;
        else if (blocks[next].value > blocks[active].value) upsatisfied = true;
        if (!upsatisfied) {
            double ww = blocks[active].weight + blocks[next].weight;
            int nextnext = blocks[next].next;
            blocks[active].value = (blocks[active].weight * blocks[active].value + block
            blocks[active].weight = ww;
```

```
            blocks[active].size += blocks[next].size;
            blocks[active].next = nextnext;
            if (nextnext < *n)
                blocks[nextnext].previous = active;
            blocks[next].size = 0;
        }
        bool downsatisfied = false;
        int previous = blocks[active].previous;
        if (previous == -1) downsatisfied = true;
        else if (blocks[previous].value < blocks[active].value) downsatisfied = true;
        if (!downsatisfied) {
            double ww = blocks[active].weight + blocks[previous].weight;
            int previousprevious = blocks[previous].previous;
            blocks[active].value = (blocks[active].weight * blocks[active].value + block
            blocks[active].weight = ww;
            blocks[active].size += blocks[previous].size;
            blocks[active].previous = previousprevious;
            if (previousprevious > -1)
                blocks[previousprevious].next = active;
            blocks[previous].size = 0;
        }
        if ((blocks[active].next == *n) && downsatisfied) break;
        if (upsatisfied && downsatisfied) active = next;
    } while (true);
    int k = 0;
    for (int i = 0; i < *n; i++) {
        int blksize = blocks[i].size;
        if (blksize > 0.0) {
            for (int j = 0; j < blksize; j++) {
                x[k] = blocks[i].value;
                k++;
            }
        }
    }
    free (blocks);
}
```

References

Cran, G. W. 1980. "Algorithm AS 149: Amalgamation of Means in the Case of Simple Ordering." *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, 209–11.

De Leeuw, J. 1977. "Correctness of Kruskal's Algorithms for Monotone Regression with Ties." *Psychometrika* 42: 141–44.

———. 2016. “Exceedingly Simple Isotone Regression with Ties.” 2016.
———. 2017a. “Exceedingly Simple Monotone Regression.” 2017.
———. 2017b. “Exceedingly Simple Sorting with Indices.” 2017.
Kruskal, J. B. 1964. “Nonmetric Multidimensional Scaling: a Numerical Method.” *Psychometrika* 29: 115–29.