The *pedigree* functions in R

Terry Therneau and Elizabeth Atkinson

December 16, 2013

Contents

1 Introduction

The pedigree routines came out of a simple need – to quickly draw a pedigree structure on the screen, within R, that was "good enough" to help with debugging the actual routines of interest, which were those for fitting mixed effecs Cox models to large family data. As such the routine had compactness and automation as primary goals; complete annotation (monozygous twins, multiple types of affected status) and most certainly elegance were not on the list. Other software could do that much better.

It therefore came as a major surprise when these routines proved useful to others. Through their constant feedback, application to more complex pedigrees, and ongoing requests for one more feature, the routine has become what it is today. This routine is still not suitable for really large pedigrees, nor for heavily inbred ones such as in animal studies, and will likely not evolve in that way. The authors' fondesthope is that others will pick up the project.

2 Pedigree

The pedigree function is the first step, creating an object of class *pedigree*. It accepts the following input

id A numeric or character vector of subject identifiers.

dadid The identifier of the father.

momid The identifier of the mother.

sex The gender of the individual. This can be a numeric variable with codes of 1=male, 2=female, 3=unknown, 4=terminated, or NA=unknown. A character or factor variable can also be supplied containing the above; the string may be truncated and of arbitrary case. A sex value of 0=male 1=female is also accepted.

status Optional, a numeric variable with 0 = censored and 1 = dead.

relationship Optional, a matrix or data frame with three columns. The first two contain the identifier values of the subject pairs, and the third the code for their relationship: 1 = Monozygotic twin, 2= Dizygotic twin, 3= Twin of unknown zygosity, 4 = Spouse.

famid Optional, a numeric or character vector of family identifiers.

The `famid` variable is placed last as it was a later addition to the code; thus prior invocations of the function that use positional arguments won't be affected. If present, this allows a set of pedigrees to be generated, one per family. The resultant structure will be an object of class `pedigreeList`.

Note that a factor variable is not listed as one of the choices for the subject identifier. This is on purpose. Factors were designed to accomodate character strings whose values came from a limited class – things like race or gender, and are not appropriate for a subject identifier. All of their special properties as compared to a character variable turn out to be backwards for this case, in particular a memory of the original level set when subscripting is done. However, due to the awful decision early on in S to automatically turn every character into a factor — unless you stood at the door with a club to head the package off — most users have become ingrained to the idea of using them for every character variable. (I encourage you to set the global option stringsAsFactors=FALSE to turn off autoconversion – it will measurably improve your R experience). Therefore, to avoid unnecessary hassle for our users the code will accept a factor as input for the id variables, but the final structure does not retain it. Gender and relation do become factors. Status follows the pattern of the survival routines and remains an integer.

We will describe the code in a set of blocks.

⟨*pedigree*⟩≡
```
pedigree <- function(id, dadid, momid, sex, affected, status, relation,
                     famid, missid) {
    ⟨pedigree-error⟩
    ⟨pedigree-parent⟩
    ⟨pedigree-create⟩
    ⟨pedigree-extra⟩
    if (missing(famid)) class(temp) <- 'pedigree'
    else class(temp) <- 'pedigreeList'
    temp
    }
⟨pedigree-subscript⟩
```

2.1 Data checks

The code starts out with some checks on the input data. Is it all the same length, are the codes legal, etc.

⟨*pedigree-error*⟩≡
```
n <- length(id)
if (length(momid) != n) stop("Mismatched lengths, id and momid")
if (length(dadid) != n) stop("Mismatched lengths, id and momid")
if (length(sex  ) != n) stop("Mismatched lengths, id and sex")

# Don't allow missing id values
if (any(is.na(id))) stop("Missing value for the id variable")
if (!is.numeric(id)) {
    id <- as.character(id)
    if (length(grep('^ *$', id)) > 0)
    stop("A blank or empty string is not allowed as the id variable")
  }

# Allow for character/numeric/factor in the sex variable
```

```
if(is.factor(sex))
        sex <- as.character(sex)
codes <- c("male","female", "unknown", "terminated")
if(is.character(sex)) sex<- charmatch(casefold(sex, upper = FALSE), codes,
                                      nomatch = 3)

# assume either 0/1/2/4 =  female/male/unknown/term, or 1/2/3/4
#  if only 1/2 assume no unknowns
if(min(sex) == 0)
        sex <- sex + 1
sex <- ifelse(sex < 1 | sex > 4, 3, sex)
if(all(sex > 2))
        stop("Invalid values for 'sex'")
    else if(mean(sex == 3) > 0.25)
        warning("More than 25% of the gender values are 'unknown'")
sex <- factor(sex, 1:4, labels = codes)
```

Create the variables descibing a missing father and/or mother, which is what we expect both for people at the top of the pedigree and for marry-ins, *before* adding in the family id information. It's easier to do it first. If there are multiple families in the pedigree, make a working set of identifiers that are of the form 'family/subject'. Family identifiers can be factor, character, or numeric.

⟨*pedigree-error*⟩+≡

```
if (missing(missid)) {
    if (is.numeric(id)) missid <- 0
    else missid <- ""
}

nofather <- (is.na(dadid) | dadid==missid)
nomother <- (is.na(momid) | momid==missid)

if (!missing(famid)) {
    if (any(is.na(famid))) stop("The family id cannot contain missing values")
    if (is.factor(famid) || is.character(famid)) {
        if (length(grep('^ *$', famid)) > 0)
            stop("The family id cannot be a blank or empty string")
        }
    #Make a temporary new id from the family and subject pair
    oldid <-id
    id <- paste(as.character(famid), as.character(id), sep='/')
    dadid <- paste(as.character(famid), as.character(dadid), sep='/')
    momid <- paste(as.character(famid), as.character(momid), sep='/')
    }

if (any(duplicated(id))) {
    duplist <- id[duplicated(id)]
```

```
    msg.n <- min(length(duplist), 6)
    stop(paste("Duplicate subject id:", duplist[1:msg.n]))
    }
```

Next check that any mother or father identifiers are found in the identifier list, and are of the right sex. Subjects who don't have a mother or father are founders. For those people both of the parents should be missing.

⟨*pedigree-parent*⟩≡

```
  findex <- match(dadid, id, nomatch = 0)
  if(any(sex[findex] != "male")) {
      who <- unique((id[findex])[sex[findex] != "male"])
      msg.n <- 1:min(5, length(who))  #Don't list a zillion
      stop(paste("Id not male, but is a father:",
                 paste(who[msg.n], collapse= " ")))
      }

  if (any(findex==0 & !nofather)) {
      who <- dadid[which(findex==0 & !nofather)]
      msg.n <- 1:min(5, length(who))  #Don't list a zillion
      stop(paste("Value of 'dadid' not found in the id list",
                 paste(who[msg.n], collapse= " ")))
      }

  mindex <- match(momid, id, nomatch = 0)
  if(any(sex[mindex] != "female")) {
      who <- unique((id[mindex])[sex[mindex] != "female"])
      msg.n <- 1:min(5, length(who))
      stop(paste("Id not female, but is a mother:",
                 paste(who[msg.n], collapse = " ")))
      }

  if (any(mindex==0 & !nomother)) {
      who <- momid[which(mindex==0 & !nomother)]
      msg.n <- 1:min(5, length(who))  #Don't list a zillion
      stop(paste("Value of 'momid' not found in the id list",
                 paste(who[msg.n], collapse= " ")))
      }

  if (any(mindex==0 & findex!=0) || any(mindex!=0 & findex==0)) {
      who <- id[which((mindex==0 & findex!=0) |(mindex!=0 & findex==0))]
      msg.n <- 1:min(5, length(who))  #Don't list a zillion
      stop(paste("Subjects must have both a father and mother, or have neither",
                 paste(who[msg.n], collapse= " ")))
  }

  if (!missing(famid)) {
```

```
    if (any(famid[mindex] != famid[mindex>0])) {
        who <- (id[mindex>0])[famid[mindex] != famid[mindex>0]]
        msg.n <- 1:min(5, length(who))
        stop(paste("Mother's family != subject's family",
                   paste(who[msg.n], collapse=" ")))
        }
    if (any(famid[findex] != famid[findex>0])) {
        who <- (id[findex>0])[famid[findex] != famid[findex>0]]
        msg.n <- 1:min(5, length(who))
        stop(paste("Father's family != subject's family",
                   paste(who[msg.n], collapse=" ")))
        }
    }
```

2.2 Creation

Now, paste the parts together into a basic pedigree. The fields for father and mother are not the identifiers of the parents, but their row number in the structure.

⟨*pedigree-create*⟩≡

```
  if (missing(famid))
      temp <- list(id = id, findex=findex, mindex=mindex, sex=sex)
  else temp<- list(famid=famid, id=oldid, findex=findex, mindex=mindex,
                   sex=sex)
```

The last part is to check out the optional features, affected status, survival status, and relationships.

Update by Jason Sinnwell, 5/2011: Allow missing values (NA) in the affected status matrix.

Update by Jason Sinnwell 7/2011: Change relation:id1 and id2 to indx1 and indx2 because they are the index of the id vector. Both *pedigree.trim* and [.*pedigree* now work with these column names.

⟨*pedigree-extra*⟩≡

```
  if (!missing(affected)) {
      if (is.matrix(affected)){
          if (nrow(affected) != n) stop("Wrong number of rows in affected")
          if (is.logical(affected)) affected <- 1* affected
          }
      else {
          if (length(affected) != n)
              stop("Wrong length for affected")

          if (is.logical(affected)) affected <- as.numeric(affected)
          if (is.factor(affected))  affected <- as.numeric(affected) -1
          }
      if(max(affected, na.rm=TRUE) > min(affected, na.rm=TRUE))
        affected <- affected - min(affected, na.rm=TRUE)
      if (!all(affected==0 | affected==1 | is.na(affected)))
```

```
                stop("Invalid code for affected status")
    temp$affected <- affected
    }

if(!missing(status)) {
    if(length(status) != n)
        stop("Wrong length for affected")
    if (is.logical(status)) status <- as.integer(status)
    if(any(status != 0 & status != 1))
        stop("Invalid status code")
    temp$status <- status
    }

if (!missing(relation)) {
    if (!missing(famid)) {
        if (is.matrix(relation)) {
            if (ncol(relation) != 4)
                stop("Relation matrix must have 3 columns + famid")
            id1 <- relation[,1]
            id2 <- relation[,2]
            code <- relation[,3]
            famid <- relation[,4]
            }
        else if (is.data.frame(relation)) {
            id1 <- relation$id1
            id2 <- relation$id2
            code <- relation$code
            famid <- relation$famid
            if (is.null(id1) || is.null(id2) || is.null(code) ||is.null(famid))
            stop("Relation data must have id1, id2, family id and code")
            }
        else stop("Relation argument must be a matrix or a dataframe")
        }
    else {
        if (is.matrix(relation)) {
            if (ncol(relation) != 3)
                stop("Relation matrix must have 3 columns")
            id1 <- relation[,1]
            id2 <- relation[,2]
            code <- relation[,3]
            }
        else if (is.data.frame(relation)) {
            id1 <- relation$id1
            id2 <- relation$id2
            code <- relation$code
            if (is.null(id1) || is.null(id2) || is.null(code))
```

```
            stop("Relation data frame must have id1, id2, and code")
        }
    else stop("Relation argument must be a matrix or a list")
    }

if (!is.numeric(code))
    code <- match(code, c("MZ twin", "DZ twin", "UZ twin", "spouse"))
else code <- factor(code, levels=1:4,
                    labels=c("MZ twin", "DZ twin", "UZ twin", "spouse"))
if (any(is.na(code)))
    stop("Invalid relationship code")

# Is everyone in this relationship in the pedigree?
if (!missing(famid)) {
    temp1 <- match(paste(as.character(famid), as.character(id1), sep='/'),
                   id, nomatch=0)
    temp2 <- match(paste(as.character(famid), as.character(id2), sep='/'),
                   id, nomatch=0)
  }
else {
    temp1 <- match(id1, id, nomatch=0)
    temp2 <- match(id2, id, nomatch=0)
  }

if (any(temp1==0 | temp2==0))
    stop("Subjects in relationships that are not in the pedigree")
if (any(temp1==temp2)) {
    who <- temp1[temp1==temp2]
    stop(paste("Subject", id[who], "is their own spouse or twin"))
    }

# Check, are the twins really twins?
ncode <- as.numeric(code)
if (any(ncode<3)) {
    twins <- (ncode<3)
    if (any(momid[temp1[twins]] != momid[temp2[twins]]))
        stop("Twins found with different mothers")
    if (any(dadid[temp1[twins]] != dadid[temp2[twins]]))
        stop("Twins found with different fathers")
    }
# Check, are the monozygote twins the same gender?
if (any(code=="MZ twin")) {
    mztwins <- (code=="MZ twin")
    if (any(sex[temp1[mztwins]] != sex[temp2[mztwins]]))
        stop("MZ Twins with different genders")
    }
```

```
##Use id index as indx1 and indx2
if (!missing(famid)) {
    temp$relation <- data.frame(famid=famid, indx1=temp1, indx2=temp2, code=code)

  }
else temp$relation <- data.frame(indx1=temp1, indx2=temp2, code=code)
}
```

The final structure will be in the order of the original data, and all the components except `relation` will have the same number of rows as the original data.

2.3 Subscripting

Subscripting of a pedigree list extracts one or more families from the list. We treat character subscripts in the same way that dimnames on a matrix are used. Factors are a problem though: assume that we have a vector x with names "joe", "charlie", "fred", then `x['joe']` is the first element of the vector, but `temp <- factor('joe', 'charlie', 'fred'); z <- temp[1]; x[z]` will be the second element! That is, the standard R approach is not what we want. Our first solution was to create an integer vector with appropriate name and subsctipt it, but a the above did it in when someone used an element of a data frame as the index.

One question is what to do if the family id is a numeric: when the user says [4] do they mean the fourth family in the list or family '4'? The user is responsible to say ['4'] in this case.

In a normal vector invalid subscripts give an NA, e.g. (1:3)[6], but since there is no such object as an "NA pedigree", we emit an error for this. The `drop` argument has no meaning for pedigrees, but must to be a defined argument of any subscript method; we simply ignore it. For both methods updating the father/mother is a minor nuisance; since they must are integer indices to rows they must be recreated after selection.

⟨*pedigree-subscript*⟩≡

```
"[.pedigreeList" <- function(x, ..., drop=F) {
    if (length(list(...)) != 1) stop ("Only 1 subscript allowed")
    ufam <- unique(x$famid)
    if (is.factor(..1) || is.character(..1)) indx <- ufam[match(..1, ufam)]
    else indx <- ufam[..1]

    if (any(is.na(indx)))
            stop(paste("Familiy", (..1[is.na(indx)])[1], "not found"))

    keep <- which(x$famid %in% indx)  #which rows to keep
    for (i in c('id', 'famid', 'sex'))
        x[[i]] <- (x[[i]])[keep]

    kept.rows <- (1:length(x$findex))[keep]
    x$findex <- match(x$findex[keep], kept.rows, nomatch=0)
    x$mindex <- match(x$mindex[keep], kept.rows, nomatch=0)
```

```
    #optional components
    if (!is.null(x$status)) x$status <- x$status[keep]
    if (!is.null(x$affected)) {
        if (is.matrix(x$affected)) x$affected <- x$affected[keep,,drop=FALSE]
        else x$affected <- x$affected[keep]
        }
    if (!is.null(x$relation)) {
        keep <- !is.na(match(x$relation$famid, names(indx)))
        if (any(keep)) {
            x$relation <- x$relation[keep,]
            ##Update twin id indexes
            x$relation$indx1 <- match(x$relation$indx1, kept.rows, nomatch=0)
            x$relation$indx2 <- match(x$relation$indx2, kept.rows, nomatch=0)
            ##If only one family chosen, remove famid
            if (length(indx)==1) {x$relation$famid <- NULL}
            }
        }
    if (length(indx)==1)  class(x) <- 'pedigree'  #only one family chosen
    else class(x) <- 'pedigreeList'
    x
    }
```

For a pedigree, the subscript operator extracts a subset of individuals. We disallow selections that retain only 1 of a subject's parents, since they cause plotting trouble later. Relations are worth keeping only if both parties in the relation were selected.

⟨*pedigree-subscript*⟩+≡

```
"[.pedigree" <- function(x, ..., drop=F) {
    if (length(list(...)) != 1) stop ("Only 1 subscript allowed")
    if (is.character(..1) || is.factor(..1)) i <- match(..1, x$id)
    else i <- (1:length(x$id))[..1]

    if (any(is.na(i))) paste("Subject", ..1[which(is.na(i))][1], "not found")

    z <- list(id=x$id[i],findex=match(x$findex[i], i, nomatch=0),
              mindex=match(x$mindex[i], i, nomatch=0),
             sex=x$sex[i])
    if (!is.null(x$affected)) {
        if (is.matrix(x$affected)) z$affected <- x$affected[i,, drop=F]
        else z$affected <- x$affected[i]
    }
    if (!is.null(x$famid)) z$famid <- x$famid[i]


    if (!is.null(x$relation)) {
      indx1 <- match(x$relation$indx1, i, nomatch=0)
      indx2 <- match(x$relation$indx2, i, nomatch=0)
```

```
    keep <- (indx1 >0 & indx2 >0)  #keep only if both id's are kept
    if (any(keep)) {
      z$relation <- x$relation[keep,,drop=FALSE]
      z$relation$indx1 <- indx1[keep]
      z$relation$indx2 <- indx2[keep]
    }
  }

  if (!is.null(x$hints)) {
      temp <- list(order= x$hints$order[i])
      if (!is.null(x$hints$spouse)) {
          indx1 <- match(x$hints$spouse[,1], i, nomatch=0)
          indx2 <- match(x$hints$spouse[,2], i, nomatch=0)
          keep <- (indx1 >0 & indx2 >0)  #keep only if both id's are kept
          if (any(keep))
              temp$spouse <- cbind(indx1[keep], indx2[keep],
                                   x$hints$spouse[keep,3])
          }
      z$hints <- temp
      }

  if (any(z$findex==0 & z$mindex>0) | any(z$findex>0 & z$mindex==0))
      stop("A subpedigree cannot choose only one parent of a subject")
  class(z) <- 'pedigree'
  z
  }
```

2.4 As Data.Frame

Convert the pedigree to a data.frame so it is easy to view when removing or trimming individuals with their various indicators. The relation and hints elements of the pedigree object are not easy to put in a data.frame with one entry per subject. These items are one entry per subject, so are put in the returned data.frame: id, findex, mindex, sex, affected, status. The findex and mindex are converted to the actual id of the parents rather than the index.

Can be used with as.data.frame(ped) or data.frame(ped). Specify in Namespace file that it is an S3 method.

⟨*as.data.frame.pedigree*⟩≡

```
as.data.frame.pedigree <- function(x, ...) {

  dadid <- momid <- rep(0, length(x$id))
  dadid[x$findex>0] <- x$id[x$findex]
  momid[x$mindex>0] <- x$id[x$mindex]
  df <- data.frame(id=x$id, dadid=dadid, momid=momid, sex=x$sex)
```

```
  if(!is.null(x$affected))
    df$affected = x$affected

  if(!is.null(x$status))
    df$status = x$status
  return(df)
}
```

This function is useful for checking the pedigree object with the $findex$ and $mindex$ vector instead of them replaced with the ids of the parents. This is not currently included in the package.

⟨*ped2df*⟩≡

```
ped2df <- function(ped) {
  df <- data.frame(id=ped$id, findex=ped$findex, mindex=ped$mindex, sex=ped$sex)
  if(!is.null(ped$affected))
    df$affected = ped$affected

  if(!is.null(ped$status))
    df$status = ped$status

  return(df)

}
```

2.5 Printing

It usually doesn't make sense to print a pedigree, since the id is just a repeat of the input data and the family connections are pointers. Thus we create a simple summary.

⟨*print.pedigree*⟩≡

```
print.pedigree <- function(x, ...) {
    cat("Pedigree object with", length(x$id), "subjects")
    if (!is.null(x$famid)) cat(", family id=", x$famid[1], "\n")
    else cat("\n")
    cat("Bit size=", bitSize(x)$bitSize, "\n")
    }

print.pedigreeList <- function(x, ...) {
    cat("Pedigree list with", length(x$id), "total subjects in",
        length(unique(x$famid)), "families\n")
    }
```

3 Kinship matrices

The kinship matrix is foundational for random effects models with family data. For n subjects it is an $n \times n$ matrix whose ij element contains the expected fraction of alleles that are identical by descent (IBD)for subject i and j. Note that the diagonal elements of the matrix will be 0.5 not 1: if I randomly sample two alleles of one of your genes, with replacement, 1/2 the time I get a father/father or mother/mother pair (IBD) and the other 1/2 the time get one of each. The truely astute reader will recognize that values ¿.5 can occur due to inbreeding, but I'll leave that discussion for others.

The algorithm used is that found in K Lange, *Mathematical and Statistical Methods for Genetic Analysis*, Springer 1997, page 71–72. It starts by setting the rows/columns for founders to .5 time the identity matrix, they require no further processing. Parents must be processed before their children, and then a child's kinship is a sum of the kinship's for his/her parents.

Start by using the `kindepth` routine to label each subject's depth in the pedigree. The initial matrix suffices for all those of depth 0, then process depth 1, etc. This guarrantees that parent's precede children. Founders are given a fake parent with id of n+1 who is unrelated to himself – a little trick that avoids some if-else logic.

The most non-obvious part of the algorithm is the inner loop over `i`. It looks like a natural candidate for S-vectorization, but you cannot. The key is `kmat[mom,] + kmat[dad,]` : as we walk through a set of siblings these vectors change, the ith element goes from 0 to the appropriate value for that sib. The dependence of each sib on prior ones is what creates the correct between-sib correlation terms. The impact of the inner loop is not so dreadful, however, since this function is run once per family. A study may have thousands of subjects but individual families within it are more modest in size.

The program can be called with a pedigree, a pedigree list, or raw data. The first argument is `id` instead of the more generic `x` for backwards compatability.

⟨*kinship*⟩≡

```
kinship <- function(id, ...) {
    UseMethod('kinship')
    }

kinship.default <- function(id, dadid, momid, ...) {
    n <- length(id)
    if (n==1)
        return(matrix(.5,1,1, dimnames=list(id, id)))
    if (any(duplicated(id))) stop("All id values must be unique")
    kmat <- diag(n+1) /2
    kmat[n+1,n+1]    <- 0

    pdepth <- kindepth(id, dadid, momid)
    mrow <- match(momid, id, nomatch=n+1) #row number of the mother
    drow <- match(dadid, id, nomatch=n+1) #row number of the dad

    for (depth in 1:max(pdepth)) {
        indx <- (1:n)[pdepth==depth]
        for (i in indx) {
```

```
            mom <- mrow[i]
            dad <- drow[i]
            kmat[i,]  <- kmat[,i] <- (kmat[mom,] + kmat[dad,])/2
            kmat[i,i] <- (1+ kmat[mom,dad])/2
            }
        }

    kmat <- kmat[1:n,1:n]
    dimnames(kmat) <- list(id, id)
    kmat
    }
```

The method for a pedigree object is and almost trivial modification. Since the mother and father are already indexed into the id list it has two lines that are different, those that create mrow and drow. Otherwise it is a complete repeat.

⟨*kinship*⟩+≡

```
kinship.pedigree <- function(id, ...) {
    n <- length(id$id)
    if (n==1)
        return(matrix(.5,1,1, dimnames=list(id$id, id$id)))
    if (any(duplicated(id$id))) stop("All id values must be unique")
    kmat <- diag(n+1) /2
    kmat[n+1,n+1]    <- 0

    pdepth <- kindepth(id)
    mrow <- ifelse(id$mindex ==0, n+1, id$mindex)
    drow <- ifelse(id$findex ==0, n+1, id$findex)

    for (depth in 1:max(pdepth)) {
        indx <- (1:n)[pdepth==depth]
        for (i in indx) {
            mom <- mrow[i]
            dad <- drow[i]
            kmat[i,]  <- kmat[,i] <- (kmat[mom,] + kmat[dad,])/2
            kmat[i,i] <- (1+ kmat[mom,dad])/2
            }
        }

    kmat <- kmat[1:n,1:n]
    dimnames(kmat) <- list(id$id, id$id)
    kmat
    }
```

For the Minnesota Family Cancer Study there are 461 families and 29114 subjects. The raw kinship matrix would be 29114 by 29114 which is over 5 terabytes of memory, something that clearly won't work within S. The solution is to store the overall matrix as a sparse Matrix object. Each family forms a single block. For this study we have `n <- table(minnbreast$famid);`

`sum(n*(n+1)/2)` or 1.07 million entries; assuming that only the lower half of each matrix is stored. The actual size is actually smaller than this, since each family's matrix will have zeros in it — founders for instance are not related — and those zeros are also not stored.

The result of each per-family call to kinship will be a symmetric matrix. We first turn each of these into a dsCMatrix object, a sparse symmetric form. The `bdiag` function is then used to paste all of these individual sparse matrices into a single large matrix.

Why don't we use `(i in famlist)` below? A numeric subscript of `[9]` selects the ninth family, not the family labeled as 9, so a numeric family id would not act as we wished. If all of the subject ids are unique, across all families, the final matrix is labeled with the subject id, otherwise it is labeled with family/subject.

⟨*kinship*⟩+≡

```
kinship.pedigreeList <- function(id, ...) {
    famlist <- unique(id$famid)
    nfam <- length(famlist)
    matlist <- vector("list", nfam)
    idlist  <- vector("list", nfam) #the possibly reorderd list of id values

    for (i in 1:length(famlist)) {
        tped <- id[i]  #pedigree for this family
        temp <- try(kinship(tped), silent=TRUE)
        if (class(temp)=="try-error")
            stop(paste("In family", famlist[i], ":", temp))
        else matlist[[i]] <- as(forceSymmetric(temp), "dsCMatrix")
        idlist[[i]] <- tped$id
    }

    result <- bdiag(matlist)
    if (any(duplicated(id$id)))
        temp <-paste(rep(famlist, sapply(idlist, length)),
                     unlist(idlist), sep='/')
    else temp <- unlist(idlist)

    dimnames(result) <- list(temp, temp)
    result
}
```

The older `makekinship` function, from before the creation of pedigreeList objects, accepts the raw identifier data, along with a special family code for unrelated subjects, as produced by the `makefamid` function. All the unrelated subjects are put at the front of the kinship matrix in this case rather than within the family. Because unrelateds get put into a fake family, we cannot create a rational family/subject identifier; the id must be unique across families. We include a copy of the routine for backwards compatability, but do not anticipate any new usage of it. Like most routines, this starts out with a collection of error checks.

⟨*makekinship*⟩≡

```
makekinship <- function(famid, id, father.id, mother.id, unrelated=0) {
    n <- length(famid)
```

```
    if (length(id)    != n) stop("Mismatched lengths: famid and id")
    if (length(mother.id) != n) stop("Mismatched lengths: famid and mother.id")
    if (length(father.id) != n) stop("Mismatched lengths: famid and father.id")
    if (any(is.na(famid)))  stop("One or more subjects with missing family id")
    if (any(is.na(id)))     stop("One or more subjects with a missing id")
    if (is.numeric(famid)) {
        if (any(famid <0))      stop("Invalid family id, must be >0")
        }

    if (any(duplicated(id))) stop("Subject ids must be unique")

    famlist <- sort(unique(famid))  #same order as the counts table
    idlist <- id            # will be overwritten, but this makes it the
                            #  correct data type and length
    counts <- table(famid)
    cumcount <- cumsum(counts)
     if (any(famid==unrelated)) {
        # Assume that those with famid of 0 are unrelated uniques
        #   (usually the marry-ins)
        temp <- match(unrelated, names(counts))
        nzero <- counts[temp]
        counts <- counts[-temp]
        famlist <- famlist[famlist != unrelated]
        idlist[1:nzero] <- id[famid== unrelated]
        cumcount <- cumsum(counts) + nzero
        }
    else nzero <- 0

    mlist <- vector('list', length(counts))
    for (i in 1:length(counts)) {
        who <- (famid == famlist[i])
        if (sum(who) ==1) mlist[[i]] <- Matrix(0.5)  # family of size 1
        else {
            mlist[[i]] <- kinship(id[who], mother.id[who], father.id[who])
            }
        idlist[seq(to=cumcount[i], length=counts[i])] <- id[who]
        }

    if (nzero>0) mlist <- c(list(Diagonal(nzero)), mlist)
    kmat <- forceSymmetric(bdiag(mlist))
    dimnames(kmat) <- list(idlist, idlist)
    kmat
}
```

4 Pedigree alignment

An *aligned* pedigree is an object that contains a pedigree along with a set of information that allows for pretty plotting. This information consists of two parts: a set of vertical and horizontal plotting coordinates along with the identifier of the subject to be plotted at each position, and a list of connections to be made between parent/child, spouse/spouse, and twin/twin. Creating this aligment turned out to be one of the more difficult parts of the project, and is the area where significant further work could be done. All the routines in this section completely ignore the `id` component of a pedigree; everyone is indexed solely by their row number in the object.

4.1 Hints

The first part of the work has to do with a `hints` list for each pedigree. It consists of 3 parts:

- The left to right order in which founders should be processed.
- The order in which siblings should be listed within a family.
- For selected spouse pairs, who is on the left/right, and which of the two should be the anchor, i.e., determine where the marriage is plotted.

The default starting values for all of these are simple: founders are processed in the order in which they appear in the data set, children appear in the order they are found in the data set, husbands are to the left of their wives, and a marriage is plotted at the leftmost spouse. A simple example where we want to bend these rules is when two families marry, and the pedigrees for both extend above the wedded pair. In the joint pedigree the pair should appear as the right-most child in the left hand family, and as the left-most child in the right hand family. With respect to founders, assume that a family has three lineages with a marriage between 1 and 2, and another between 2 and 3. In the joint pedigree the sets should be 1, 2, 3 from left to right.

The hints consist of a list with two components. The first is a vector of numbers of the same length as the pedigree, used to order the female founders and to order siblings within family. For subjects not part of either of these the value can be arbitrary. The second is a 3 column matrix of spouse pairs, each row indicates the left-hand member of the pair, the right-hand member, and which of the two is the anchor, i.e., directly connected to thier parent. Double and triple marriages can start to get interesting.

The `autohint` routine is used to create an initial hints list. It is a part of the general intention to make the routine do "pretty good" drawings automatically. The basic algorithm is trial and error.

- Start with the simplest possible hints (user input is accepted)
- Call align.pedigree to see how this works out
- Fix any spouses that are not next to each other but could be.
- Any fix on the top level mixes up everything below, so we do the fixes one level at a time.

The routine makes no attempt to reorder founders. It just isn't smart enoughto figure that out.

The first thing to be done is to check on twins. They are a nuisance, since twins need to move together. The `ped$relation` object has a factor in it, so first turn that into numeric. We create

3 vectors: `twinrel` is a matrix containing pairs of twins and their relation, it is a subset of the incoming `relation` matrix. The `twinset` vector identifies twins, it is 0 for anyone who is not a part of a multiple-birth set, and a unique id for each member of a set. We use the minimum row number of the members of the set as the id. `twinord` is a starting order vector for the set; it mostly makes sure that there are no ties (who knows what a user may have used for starting values.)

⟨*autohint*⟩≡

```
autohint <- function(ped, hints) {
    if (!is.null(ped$hints)) return(ped$hints)  #nothing to do
    n <- length(ped$id)
    depth <- kindepth(ped, align=TRUE)

    if (is.null(ped$relation)) relation <- NULL
    else  relation <- cbind(as.matrix(ped$relation[,1:2]),
                            as.numeric(ped$relation[,3]))
    if (!is.null(relation) && any(relation[,3] <4)) {
        temp <- (relation[,3] < 4)
        twinlist <- unique(c(relation[temp,1:2]))  #list of twin id's
        twinrel  <- relation[temp,,drop=F]

        twinset <- rep(0,n)
        twinord <- rep(1,n)
        for (i in 2:length(twinlist)) {
            # Now, for any pair of twins on a line of twinrel, give both
            #  of them the minimum of the two ids
            # For a set of triplets, it might take two iterations for the
            #  smallest of the 3 numbers to "march" across the threesome.
            #  For quads, up to 3 iterations, for quints, up to 4, ....
            newid <- pmin(twinrel[,1], twinrel[,2])
            twinset[twinrel[,1]] <- newid
            twinset[twinrel[,2]] <- newid
            twinord[twinrel[,2]] <- pmax(twinord[twinrel[,2]],
                                         twinord[twinrel[,1]]+1)
            }
        }
    else {
        twinset <- rep(0,n)
        twinrel <- NULL
        }
    ⟨autohint-shift⟩
    ⟨autohint-init⟩
    ⟨autohint-fixup⟩
    list(order=horder, spouse=sptemp)
    }
```

Next is an internal function that rearranges someone to be the leftmost or rightmost of his/her siblings. The only real complication is twins – if one of them moves the other has to move too. And we need to keep the monozygotics together within a band of triplets. Algorithm: if the person to be moved is part of a twinset, first move all the twins to the left end (or right as the case may be), then move all the monozygotes to the left, then move the subject himself to the left.

⟨*autohint-shift*⟩≡

```
shift <- function(id, sibs, goleft, hint, twinrel, twinset) {
    if (twinset[id]> 0)  {
        shift.amt <- 1 + diff(range(hint[sibs]))  # enough to avoid overlap
        twins <- sibs[twinset[sibs]==twinset[id]]
        if (goleft)
         hint[twins] <- hint[twins] - shift.amt
        else hint[twins] <- hint[twins] + shift.amt

        mono  <- any(twinrel[c(match(id, twinrel[,1], nomatch=0),
                           match(id, twinrel[,2], nomatch=0)),3]==1)
        if (mono) {
        #
        # ok, we have to worry about keeping the monozygotics
        #  together within the set of twins.
        # first, decide who they are, by finding those monozygotic
            #  with me, then those monozygotic with the results of that
            #  iteration, then ....  If I were the leftmost, this could
            #  take (#twins -1) iterations to get us all
            #
        monoset <- id
        rel2 <- twinrel[twinrel[,3]==1, 1:2, drop=F]
        for (i in 2:length(twins)) {
            newid1 <- rel2[match(monoset, rel2[,1], nomatch=0),2]
            newid2 <- rel2[match(monoset, rel2[,2], nomatch=0),1]
            monoset <- unique(c(monoset, newid1, newid2))
            }
        if (goleft)
               hint[monoset]<- hint[monoset] - shift.amt
        else   hint[monoset]<- hint[monoset] + shift.amt
        }
        }

    #finally, move the subject himself
    if (goleft) hint[id] <- min(hint[sibs]) -1
    else        hint[id] <- max(hint[sibs]) +1

    hint[sibs] <- rank(hint[sibs])  # aesthetics -- no negative hints
    hint
```

```
    }
```

Now, get an ordering of the pedigree to use as the starting point. The numbers start at 1 on each level. We don't need the final "prettify" step, hence align=F. If there is a hints structure entered, we retain it's non-zero entries, otherwise people are put into the order of the data set. We allow the hints input to be only an order vector Twins are then further reordered.

⟨*autohint-init*⟩≡

```
if (!missing(hints)) {
    if (is.vector(hints)) hints <- list(order=hints)
    if (is.matrix(hints)) hints <- list(spouse=hints)
    if (is.null(hints$order)) horder <- integer(n)
    else horder <- hints$order
    }
else horder <- integer(n)

for (i in unique(depth)) {
    who <- (depth==i & horder==0)
    if (any(who)) horder[who] <- 1:sum(who) #screwy input - overwrite it
    }

if (any(twinset>0)) {
    # First, make any set of twins a cluster: 6.01, 6.02, ...
    #  By using fractions, I don't have to worry about other sib's values
    for (i in unique(twinset)) {
        if (i==0) next
        who <- (twinset==i)
        horder[who] <- mean(horder[who]) + twinord[who]/100
        }

    # Then reset to integers
    for (i in unique(ped$depth)) {
        who <- (ped$depth==i)
        horder[who] <- rank(horder[who])  #there should be no ties
        }
    }

if (!missing(hints)) sptemp <- hints$spouse
else sptemp <- NULL
plist <- align.pedigree(ped, packed=TRUE, align=FALSE,
                        hints=list(order=horder, spouse=sptemp))
```

The result coming back from align.pedigree is a set of vectors and matrices:

n vector, number of entries per level

nid matrix, one row per level, numeric id of the subject plotted here

spouse integer matrix, one row per level, subject directly to my right is my spouse (1), a double marriage (2), or neither (0).

fam matrix, link upward to my parents, or 0 if no link.

Now, walk down through the levels one by one. A candidate subject is one who appears twice on the level, once under his/her parents and once somewhere else as a spouse. Move this person and spouse the the ends of their sibships and add a marriage hint. Figure 1 shows a simple case. The input data set has the subjects ordered from 1–11, the left panel is the result without hints which processes subjects in the order encountered. The return values from `align.pedigree` have subject 9 shown twice. The first is when he is recognized as the spouse of subject 4, the second as the child of 6–7.

The basic logic is

1. Find a subject listed multiple times on a line (assume it is a male). This means that he has multiple connections, usually one to his parents and the other to a spouse tied to her parents. (If the spouse were a marry-in she would have been placed alongside and there would be no duplication.)

2. Say subject x is listed at locations 2, 8, and 12. We look at one pairing at a time, either 2-8 or 8-12. Consider the first one.

   - If position 2 is associated with siblings, rearrange them to put subject 2 on the right. If it is associated with a spouse at this location, put that spouse on the right of her siblings.
   - Repeat the work for position 8, but moving targets to the left.
   - At either position, if it is associated with a spouse then add a marriage. If both ends of the marriage are anchored, i.e., connected to a family, then either end may be listed as the anchor in the output; follow the suggestion of the duporder routine. If only one is, it is usually better to anchor it there, so that the marriage is processed by`align.pedigree` when that family is. (At least I think so.)

This logic works 9 times out of 10, at least for human pedigrees. We'll look at more complex cases below when looking at the `duporder` (order the duplicates) function, which returns a matrix with columns 1 and 2 being a pair of duplicates, and 3 a direction. Note that in the following code `idlist` refers to the row numbers of each subject in the pedigree, not to their label `ped$id`.

⟨*autohint-fixup*⟩≡
```
  ⟨autohint-find⟩
  ⟨autohint-duporder⟩
  maxlev <- nrow(plist$nid)
  for (lev in 1:maxlev) {
      idlist <- plist$nid[lev,1:plist$n[lev]] #subjects on this level
      dpairs <- duporder(idlist, plist, lev, ped)  #duplicates to be dealt with
      if (nrow(dpairs)==0) next;
      for (i in 1:nrow(dpairs)) {
          anchor <- spouse <- rep(0,2)
          for (j in 1:2) {
              direction <- c(FALSE, TRUE)[j]
              mypos <- dpairs[i,j]
              if (plist$fam[lev, mypos] >0) {
```

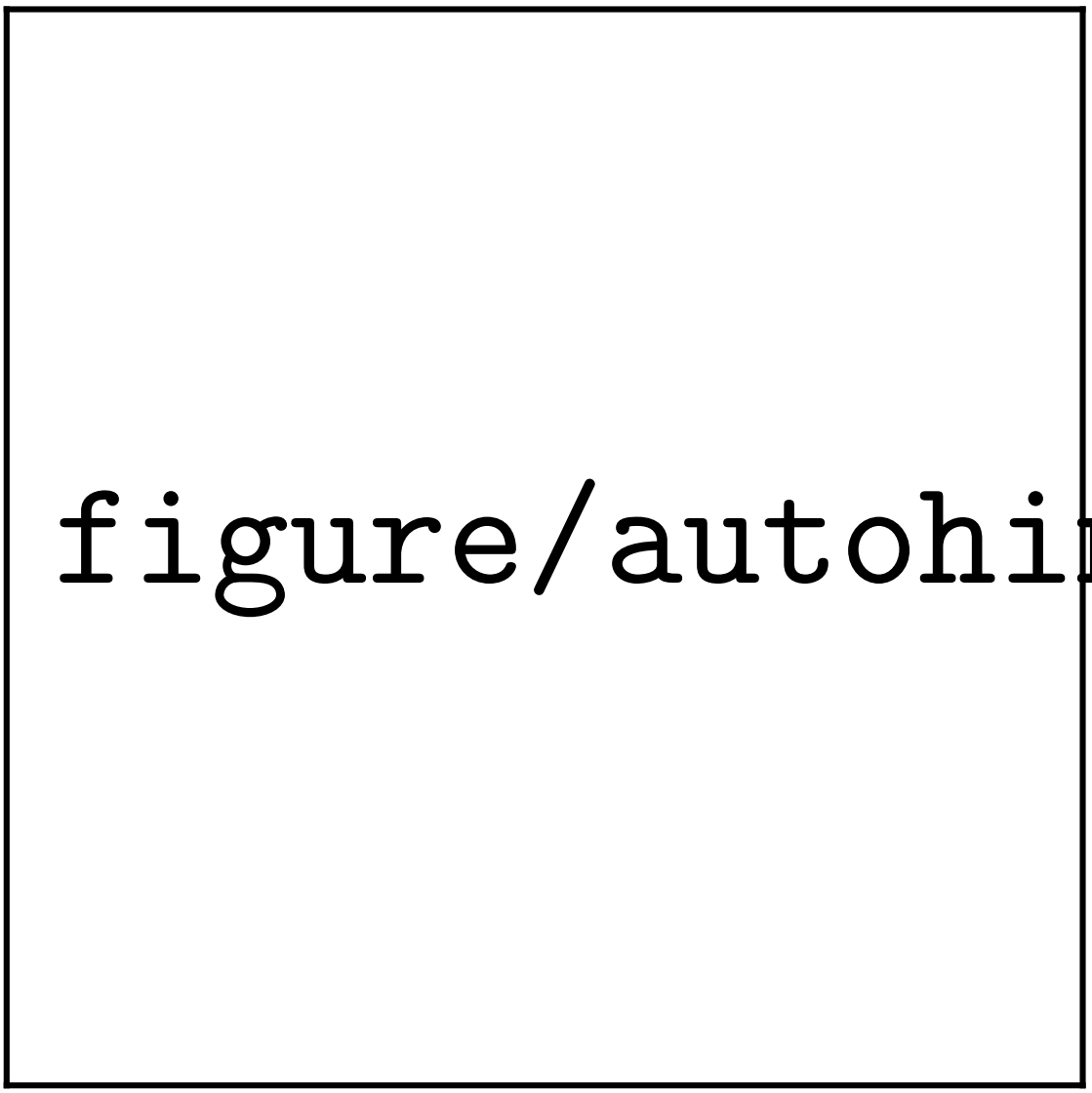

Figure 1: A simple pedigree before (left) and after (right) the autohint computations.

```
        # Am connected to parents at this location
        anchor[j] <- 1  #familial anchor
        sibs <- idlist[findsibs(mypos, plist, lev)]
        if (length(sibs) >1)
            horder <- shift(idlist[mypos], sibs, direction,
                            horder, twinrel, twinset)
        }
    else {
        #spouse at this location connected to parents ?
        spouse[j] <- findspouse(mypos, plist, lev, ped)
        if (plist$fam[lev,spouse[j]] >0) { # Yes they are
            anchor[j] <- 2  #spousal anchor
            sibs <- idlist[findsibs(spouse[j], plist, lev)]
            if (length(sibs) > 1)
                horder <- shift(idlist[spouse[j]], sibs, direction,
                            horder, twinrel, twinset)
            }
        }
    }
```

At this point the most common situation will be what is shown in figure 1. The variable `anchor` is (2,1) showing that the left hand copy of subject 9 is connected to an anchored spouse and the right hand copy is himself anchored. The proper addition to the spouselist is `(4, 9, dpairs)`, where the last is the hint from the dpairs routine as to which of the parents is the one to follow further when drawing the entire pedigree. (When drawing a pedigree and there is a child who can be reached from multiple founders, we only want to find the child once.)

The double marry-in found in figure **??**, subject 11, leads to value of (2,2) for the `anchor` variable. The proper addition to the `sptemp` matrix in this case will be two rows, (5, 11, 1) indicating that 5 should be plotted left of 11 for the 5-11 marriage, with the first partner as the anchor, and a second row (11, 9, 2). This will cause the common spouse to be plotted in the middle.

Multiple marriages can lead to unanchored subjects. In the left hand portion of figure 2 we have two double marriages, one on the left and one on the right with anchor values of (0,2) and (2,0), respectively. We add two marriages to the return list to ensure that both print in the correct left-right order; the 14-4 one is correct by default but it's easier to output a line than check sex orders.

The left panel of figure 2 shows a case where subject 11 marries into the pedigree but also has a second spouse. The `anchor` variable for this case will be (2, 0); the first instance of 11 has a spouse tied into the tree above, the second instance has no upward connections. In the top row, subject 6 has values of (0, 0) since neither connection has an upward parent. In the right hand panel subject 2 has an anchor variable of (0,1).

⟨*autohint-fixup*⟩+≡

```
        # add the marriage(s)
        id1 <- idlist[dpairs[i,1]]  # i,1 and i,2 point to the same person
        id2 <- idlist[spouse[1]]
        id3 <- idlist[spouse[2]]
```

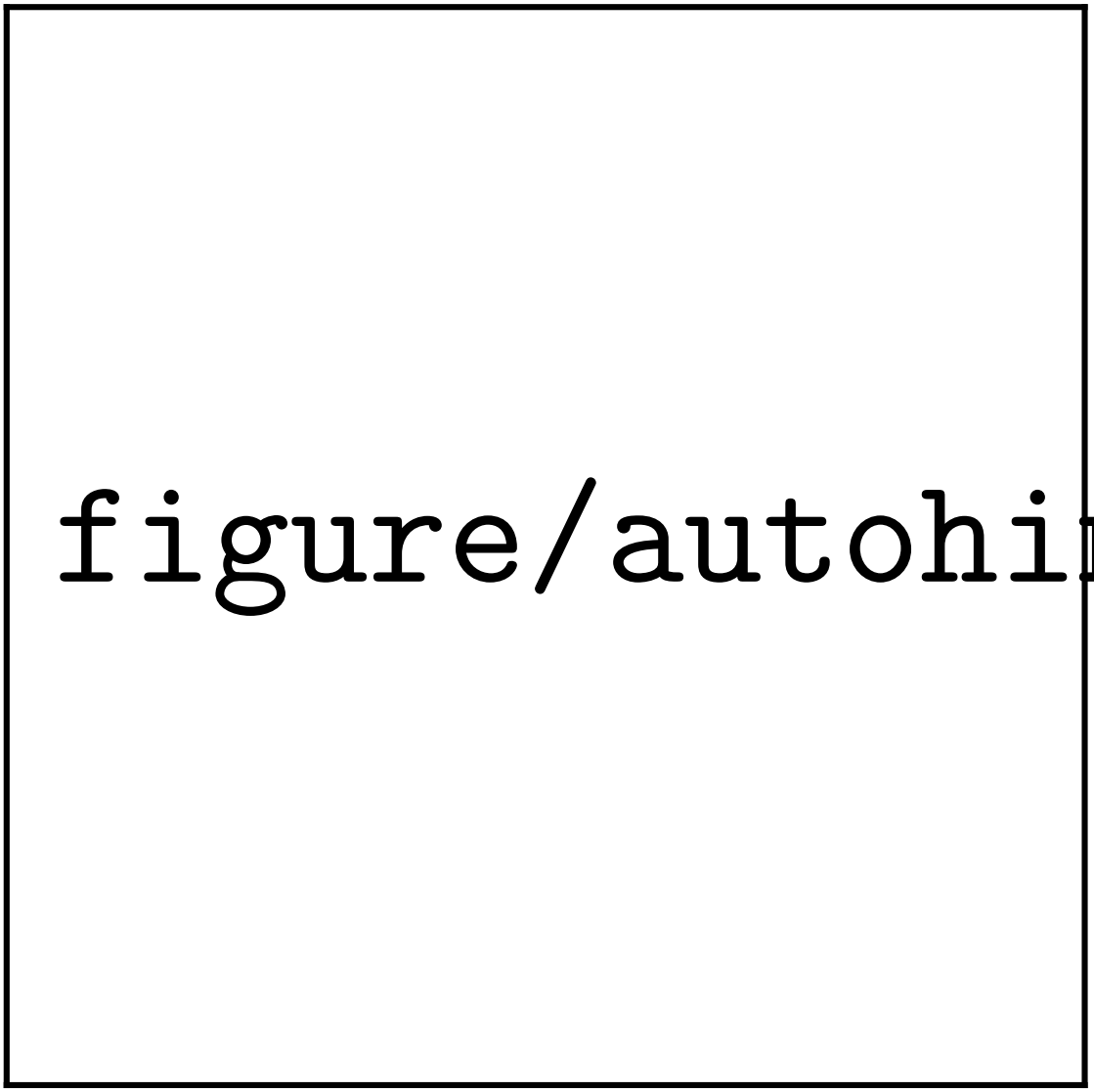

Figure 2: Pedigrees with multiple marriages

```
    temp <- switch(paste(anchor, collapse=''),
                   "21" = c(id2, id1, dpairs[i,3]),   #the most common case
                   "22" = rbind(c(id2, id1, 1), c(id1, id3, 2)),
                   "02" = c(id2, id1, 0),
                   "20" = c(id2, id1, 0),
                   "00" = rbind(c(id1, id3, 0), c(id2, id1, 0)),
                   "01" = c(id2, id1, 2),
                   "10" = c(id1, id2, 1),
                   NULL)

    if (is.null(temp)) {
        warning("Unexpected result in autohint, please contact developer")
        return(list(order=1:n))  #punt
      }
    else sptemp <- rbind(sptemp, temp)
    }
#
# Recompute, since this shifts things on levels below
#
plist <- align.pedigree(ped, packed=TRUE, align=FALSE,
                        hints=list(order=horder, spouse=sptemp))
}
```

For the case shown in figure **??** the `duporder` function will return a single row array with values (2, 6, 1), the first two being the positions of the duplicated subject. The anchor will be 2 since that is the copy connected to parents The direction is TRUE, since the spouse is to the left of the anchor point. The id is 9, sibs are 8, 9, 10, and the shift function will create position hints of 2,1,3, which will cause them to be listed in the order 9, 8, 10.

The value of spouse is 3 (third position in the row), subjects 3,4, and 5 are reordered, and finally the line (4,9,1) is added to the sptemp matrix. In this particular case the final element could be a 1 or a 2, since both are connected to their parents.

Figure 3 shows a more complex case with several arcs. In the upper left is a double marry-in. The `anchor` variable in the above code will be (2,2) since both copies have an anchored spouse. The left and right sets of sibs are reordered (even though the left one does not need it), and two lines are added to the sptemp matrix: (5,11,1) and (11,9,2).

On the upper right is a pair of overlapping arcs. In the final tree we want to put sibling 28 to the right of 29 since that will allow one node to join, but if we process the subjects in lexical order the code will first shift 28 to the right and then later shift over 29. The duporder function tries to order the duplicates into a matrix so that the closest ones are processed last. The definition of close is based first on whether the families touch, and second on the actual distance. The third column of the matrix hints at whether the marriage should be plotted at the left (1) or right (2) position of the pair. The goal for this is to spread apart families of cousins; in the example to not have the children of 28/31 plotted under the 21/22 grandparents, and those for 29/32 under the 25/26 grandparents. The logic for this column is very ad hoc: put children near the edges.

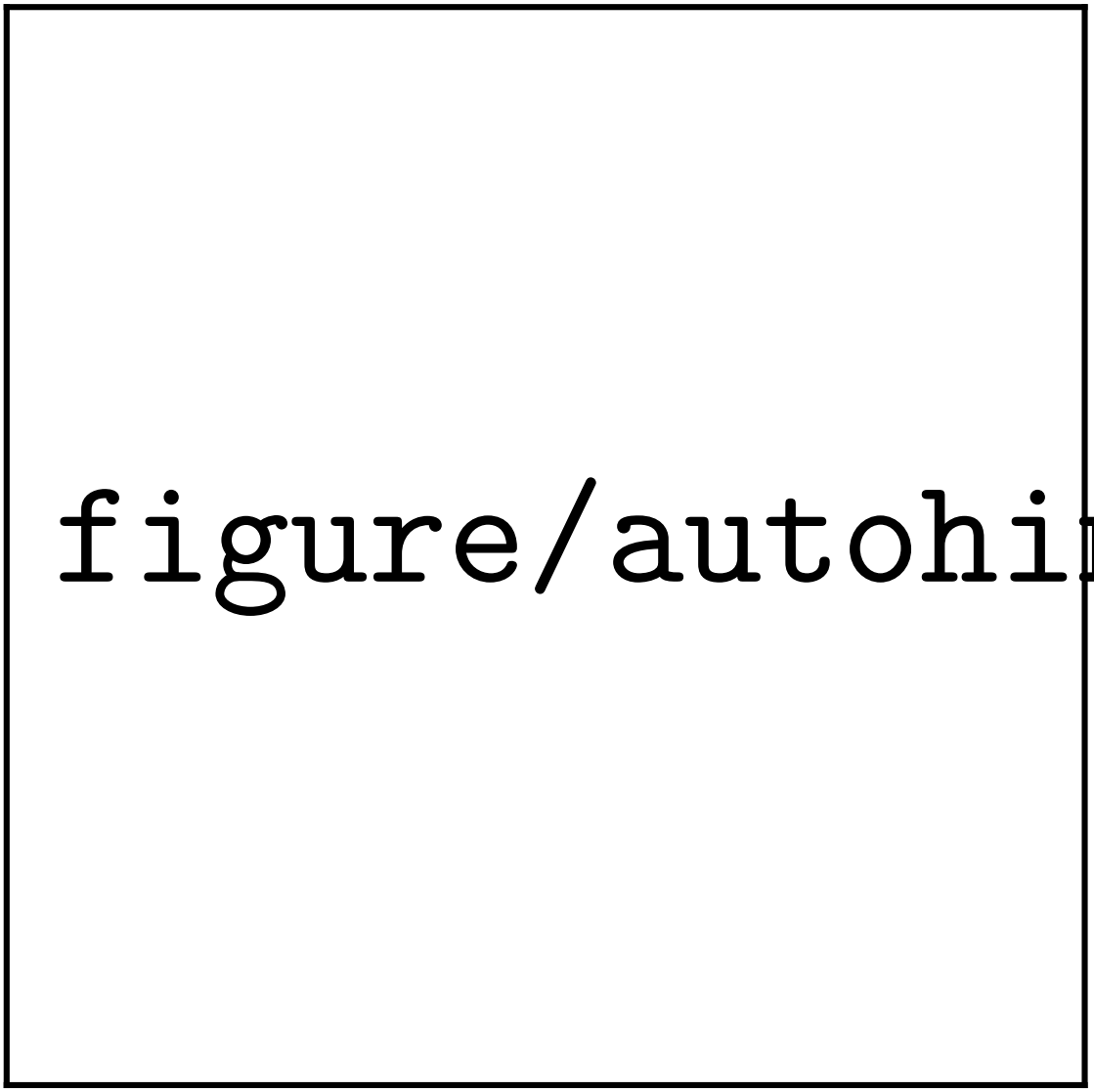

Figure 3: A more complex pedigree.

⟨*autohint-duporder*⟩≡

```
duporder <- function(idlist, plist, lev, ped) {
    temp <- table(idlist)
    if (all(temp==1)) return (matrix(0L, nrow=0, ncol=3))

    # make an intial list of all pairs's positions
    # if someone appears 4 times they get 3 rows
    npair <- sum(temp-1)
    dmat <- matrix(0L, nrow=npair, ncol=3)
    dmat[,3] <- 2; dmat[1:(npair/2),3] <- 1
    i <- 0
    for (id in unique(idlist[duplicated(idlist)])) {
        j <- which(idlist==id)
        for (k in 2:length(j)) {
            i <- i+1
            dmat[i,1:2] <- j[k + -1:0]
            }
        }
    if (nrow(dmat)==1) return(dmat)  #no need to sort it

    # families touch?
    famtouch <- logical(npair)
    for (i in 1:npair) {
        if (plist$fam[lev,dmat[i,1]] >0)
             sib1 <- max(findsibs(dmat[i,1], plist, lev))
        else {
            spouse <- findspouse(dmat[i,1], plist, lev, ped)
            ##If spouse is marry-in then move on without looking for sibs
                if (plist$fam[lev,spouse]==0) {famtouch[i] <- F; next}
            sib1 <- max(findsibs(spouse, plist, lev))
            }

        if (plist$fam[lev, dmat[i,2]] >0)
            sib2 <- min(findsibs(dmat[i,2], plist, lev))
        else {
            spouse <- findspouse(dmat[i,2], plist, lev, ped)
            ##If spouse is marry-in then move on without looking for sibs
                if (plist$fam[lev,spouse]==0) {famtouch[i] <- F; next}
            sib2 <- min(findsibs(spouse, plist, lev))
            }
        famtouch[i] <- (sib2-sib1 ==1)
        }
    dmat[order(famtouch, dmat[,1]- dmat[,2]),, drop=FALSE ]
    }
```

Finally, here are two helper routines. Finding my spouse can be interesting – suppose we have a listing with Shirley, Fred, Carl, me on the line with the first three marked as spouse=TRUE – it means that she has been married to all 3 of us. First we find the string from rpos to lpos that is a marriage block; 99% of the time this will be of length 2 of course. Then find the person in that block who is opposite sex, and check that they are connected. The routine is called with a left-right position in the alignment arrays and returns a position.

⟨*autohint-find*⟩≡
```
findspouse <- function(mypos, plist, lev, ped) {
    lpos <- mypos
    while (lpos >1 && plist$spouse[lev, lpos-1]) lpos <- lpos-1
    rpos <- mypos
    while(plist$spouse[lev, rpos]) rpos <- rpos +1
    if (rpos==lpos) stop("autohint bug 3")

    opposite <-ped$sex[plist$nid[lev,lpos:rpos]] != ped$sex[plist$nid[lev,mypos]]
    if (!any(opposite)) stop("autohint bug 4")  # no spouse
    spouse <- min((lpos:rpos)[opposite])  #can happen with a triple marriage
    spouse
    }
```

The findsibs function starts with a position and returns a position as well.

⟨*autohint-find*⟩+≡
```
findsibs <- function(mypos, plist, lev) {
    family <- plist$fam[lev, mypos]
    if (family==0) stop("autohint bug 6")
    which(plist$fam[lev,] == family)
    }
```

4.2 Align.pedigree

The top level routine for alignment has 5 arguments

ped a pedigree or pedigreeList object. In the case of the latter we loop over each family separately.

packed do we allow branches of the tree to overlap? If FALSE the drawing is much easier, but final drawing can take up a huge amount of space.

width the minimum width for a packed pedigree. This affects only small pedigrees, since the minimum possible width for a pedigree is the largest number of individiuals in one of the generations.

align should the final step of alignment be done? This tries to center children under parents, to the degree possible.

a hints object. This is normally blank and autohint is invoked.

The result coming back from align.pedigree is a set of vectors and matrices:

n vector, number of entries per level

nid matrix, one row per level, numeric id of the subject plotted here

pos the horizontal position for plotting

spouse integer matrix, one row per level, subject directly to my right is my spouse (1), a double marriage (2), or neither (0).

fam matrix, link upward to my parents, or 0 if no link.

⟨*align.pedigree*⟩≡
```
align.pedigree <- function(ped, packed=TRUE, width=10,
                           align=TRUE, hints=ped$hints) {
    if (class(ped)== 'pedigreeList') {
        nped <- length(unique(ped$famid))
        alignment <- vector('list', nped)
        for (i in 1:nped) {
            temp <- align.pedigree(ped[i], packed, width, align)
            alignment[[i]] <- temp$alignment
            }
        ped$alignment <- alignment
        class(ped) <- 'pedigreeListAligned'
        return(ped)
        }

    if (is.null(hints)) hints <- autohint(ped)
    else     hints <- check.hint(hints, ped$sex)

    ⟨align-setup⟩
    ⟨align-founders⟩
    ⟨align-finish⟩
    }
```

Start with some setup. Throughout this routine the row number is used as a subject id (ignoring the actual id label).

- Check that everyone has either two parents or none (a singleton will just confuse us).
- Verify that the hints are correct.
- The relation data frame, if present, has a factor in it. Turn that into numeric.
- Create the `spouselist` array. This has 4 columns
  1. Husband index (4= 4th person in the pedigree structure)
  2. Wife index
  3. Plot order: 1= husband left, 2=wife left
  4. Anchor: 1=left member, 2=right member, 0= not yet determined

As the routine proceeds a spousal pair can be encountered multiple times; we take them out of this list when the "connected" member is added to the pedigree so that no marriage gets added twice.

- To detect duplicates on the spouselist we need to create a unique (but temporary) spouse-pair id using a simple hash.

When importing data from autohint, that routine's spouse matrix has column 1 = subject plotted on the left, 2 = subject plotted on the right. The `spouselist` array has column 1=husband, 2=wife. Hence the clumsy looking ifelse below. The autohint format is more congenial to users, who might modify the output, the spouselist format easier for the code.

⟨*align-setup*⟩≡

```
n <- length(ped$id)
dad <- ped$findex; mom <- ped$mindex  #save typing
if (any(dad==0 & mom>0) || any(dad>0 & mom==0))
        stop("Everyone must have 0 parents or 2 parents, not just one")
level <- 1 + kindepth(ped, align=TRUE)

horder <- hints$order   # relative order of siblings within a family

if (is.null(ped$relation)) relation <- NULL
else  relation <- cbind(as.matrix(ped$relation[,1:2]),
                        as.numeric(ped$relation[,3]))

if (!is.null(hints$spouse)) { # start with the hints list
    tsex <- ped$sex[hints$spouse[,1]]  #sex of the left member
    spouselist <- cbind(0,0,  1+ (tsex!='male'),
                        hints$spouse[,3])
    spouselist[,1] <- ifelse(tsex=='male', hints$spouse[,1], hints$spouse[,2])
    spouselist[,2] <- ifelse(tsex=='male', hints$spouse[,2], hints$spouse[,1])
    }
else spouselist <- matrix(0L, nrow=0, ncol=4)

if (!is.null(relation) && any(relation[,3]==4)) {
    # Add spouses from the relationship matrix
    trel <- relation[relation[,3]==4,,drop=F]
    tsex <- ped$sex[trel[,1]]
    trel[tsex!='male',1:2] <- trel[tsex!='male',2:1]
    spouselist <- rbind(spouselist, cbind(trel[,1],
                                          trel[,2],
                                          0,0))
    }
if (any(dad>0 & mom>0) ) {
    # add parents
    who <- which(dad>0 & mom>0)
    spouselist <- rbind(spouselist, cbind(dad[who], mom[who], 0, 0))
```

```
    }

  hash <- spouselist[,1]*n + spouselist[,2]
  spouselist <- spouselist[!duplicated(hash),, drop=F]
```

The `alignped` routine does the alignment using 3 co-routines:

alignped1 called with a single subject, returns the subtree founded on this subject, as though it were the only tree

alignped2 called with a set of sibs, calls alignped1 and alignped3 multiple times to create a joint pedigree

alignped3 given two side by side plotting structures, merge them into a single one

Call `alignped1` sequentially with each founder pair and merge the results. A founder pair is a married pair, neither of which has a father.

⟨*align-founders*⟩≡

```
  noparents <- (dad[spouselist[,1]]==0 & dad[spouselist[,2]]==0)
   ##Take duplicated mothers and fathers, then founder mothers
  dupmom <- spouselist[noparents,2][duplicated(spouselist[noparents,2])] #Founding mothers with mul
  dupdad <- spouselist[noparents,1][duplicated(spouselist[noparents,1])] #Founding fathers with mul
  foundmom <- spouselist[noparents&!(spouselist[,1] %in% c(dupmom,dupdad)),2] # founding mothers
  founders <-  unique(c(dupmom, dupdad, foundmom))
  founders <-  founders[order(horder[founders])]  #use the hints to order them
  rval <- alignped1(founders[1], dad, mom, level, horder,
                       packed=packed, spouselist=spouselist)

  if (length(founders)>1) {
      spouselist <- rval$spouselist
      for (i in 2:length(founders)) {
          rval2 <- alignped1(founders[i], dad, mom,
                             level, horder, packed, spouselist)
          spouselist <- rval2$spouselist
          rval <- alignped3(rval, rval2, packed)
          }
      }
```

Now finish up. There are 4 tasks to doS

1. For convenience the lower level routines kept the spouse and nid arrays as a single object – unpack them
2. In the spouse array a 1 in position i indicates that subject i and i+1 are joined as a marriage. If these two have a common ancestor change this to a 2, which indicates that a double line should be used in the plot.
3. Add twins data to the output.
4. Do final alignment

⟨*align-finish*⟩≡
```
  #
  # Unhash out the spouse and nid arrays
  #
  nid    <- matrix(as.integer(floor(rval$nid)), nrow=nrow(rval$nid))
  spouse <- 1L*(rval$nid != nid)
  maxdepth <- nrow(nid)

  # For each spouse pair, find out if it should be connected with
  #  a double line.  This is the case if they have a common ancestor
  ancestor <- function(me, momid, dadid) {
      alist <- me
      repeat {
          newlist <- c(alist, momid[alist], dadid[alist])
          newlist <- sort(unique(newlist[newlist>0]))
          if (length(newlist)==length(alist)) break
          alist <- newlist
          }
      alist[alist!=me]
      }
  for (i in (1:length(spouse))[spouse>0]) {
      a1 <- ancestor(nid[i], mom, dad)
      a2 <- ancestor(nid[i+maxdepth],mom, dad)  #matrices are in column order
      if (any(duplicated(c(a1, a2)))) spouse[i] <- 2
      }
```

The twins array is of the same shape as the spouse and nid arrays: one row per level giving data for the subjects plotted on that row. In this case they are

- 0= nothing
- 1= the sib to my right is a monzygotic twin,
- 2= the sib to my right is a dizygote,
- 3= the sib to my right is a twin, unknown zyogosity.

⟨*align-finish*⟩+≡
```
  if (!is.null(relation) && any(relation[,3] < 4)) {
      twins <- 0* nid
      who  <- (relation[,3] <4)
      ltwin <- relation[who,1]
      rtwin <- relation[who,2]
      ttype <- relation[who,3]

      # find where each of them is plotted (any twin only appears
      #   once with a family id, i.e., under their parents)
      ntemp <- ifelse(rval$fam>0, nid,0) # matix of connected-to-parent ids
```

```
    ltemp <- (1:length(ntemp))[match(ltwin, ntemp, nomatch=0)]
    rtemp <- (1:length(ntemp))[match(rtwin, ntemp, nomatch=0)]
    twins[pmin(ltemp, rtemp)] <- ttype
    }
else twins <- NULL
```

At this point the pedigree has been arranged, with the positions in each row going from 1 to (number of subjects in the row). (For a packed pedigree, which is the usual case). Having everything pushed to the left margin isn't very pretty, now we fix that. Note that alignped4 wants a T/F spouse matrix: it doesn't care about your degree of relationship to the spouse.

⟨*align-finish*⟩+≡

```
if ((is.numeric(align) || align) && max(level) >1)
    pos <- alignped4(rval, spouse>0, level, width, align)
else pos <- rval$pos

if (is.null(twins))
     list(n=rval$n, nid=nid, pos=pos, fam=rval$fam, spouse=spouse)
else list(n=rval$n, nid=nid, pos=pos, fam=rval$fam, spouse=spouse,
              twins=twins)
```

4.3 alignped1

This is the first of the three co-routines. It is called with a single subject, and returns the subtree founded on said subject, as though it were the only tree. We only go down the pedigree, not up. Input arguments are

nid the numeric id of the subject in question

dad points to the row of the father, 0=no father in pedigree

mom points to the row of the mother

level the plotting depth of each subject

horder orders the kids within a sibship

packed if true, everything is slid to the left

spouselist a matrix of spouses

- col 1= pedigree index of the husband
- col 2= pedigree index of the wife
- col 3= 1:plot husband to the left, 2= wife to the left
- col 4= 1:left member is rooted here, 2=right member, 0=either

The return argument is a set of matrices as described in section 4.2, along with the spouselist matrix. The latter has marriages removed as they are processed..

In this routine the `nid` array consists of the final nid array + 1/2 of the final spouse array. The basic algorithm is simple.

1. Find all of the spouses for which `x` is the anchor subject. If there are none then return the trivial tree consisting of `x` alone.

2. For each marriage in the set, call `alignped2` on the children and add this to the result.

Note that the `spouselist` matrix will only contain spouse pairs that are not yet processed. The logic for anchoring is slightly tricky. First, if row 4 of the spouselist matrix is 0, we anchor at the first opportunity, i.e. now.. Also note that if spouselist[,3]==spouselist[,4] it is the husband who is the anchor (just write out the possibilities).

⟨*alignped1*⟩≡
```
alignped1 <- function(x, dad, mom, level, horder, packed, spouselist){
    # Set a few constants
    maxlev <- max(level)
    lev <- level[x]
    n <- integer(maxlev)

    if (length(spouselist)==0)  spouse <- NULL
    else {
        if (any(spouselist[,1]==x)){
            sex <- 1                                # I'm male
            sprows <- (spouselist[,1]==x & (spouselist[,4] ==spouselist[,3] |
                                            spouselist[,4] ==0))
            spouse <- spouselist[sprows, 2] #ids of the spouses
            }
        else {
            sex <- 2
            sprows <- (spouselist[,2]==x & (spouselist[,4]!=spouselist[,3] |
                                            spouselist[,4] ==0))
            spouse <- spouselist[sprows, 1]
            }
        }
    # Marriages that cross levels are plotted at the higher level (lower
    #  on the paper).
    if (length(spouse)) {
        keep <- level[spouse] <= lev
        spouse <- spouse[keep]
        sprows <- (which(sprows))[keep]
        }
    nspouse <- length(spouse)  # Almost always 0, 1 or 2
```

Create the set of 3 return structures, which will be matrices with (1+nspouse) columns. If there are children then other routines will widen the result.

⟨*alignped1*⟩+≡
```
    nid <- fam <- matrix(0L, maxlev, nspouse+1)
    pos <- matrix(0.0, maxlev, nspouse +1)
    n[lev] <- nspouse +1
```

```
pos[lev,] <- 0:nspouse
if (nspouse ==0) {
    # Easy case: the "tree rooted at x" is only x itself
    nid[lev,1] <- x
    return(list(nid=nid, pos=pos, fam=fam, n=n, spouselist=spouselist))
    }
```

Now we have a list of spouses that should be dealt with and the the correponding columns of the spouselist matrix. Create the two complimentary lists lspouse and rspouse to denote those plotted on the left and on the right. For someone with lots of spouses we try to split them evenly. If the number of spouses is odd, then men should have more on the right than on the left, women more on the right. Any hints in the spouselist matrix override. We put the undecided marriages closest to `x`, then add predetermined ones to the left and right. The majority of marriages will be undetermined singletons, for which nleft will be 1 for female (put my husband to the left) and 0 for male.

⟨*alignped1*⟩+≡

```
lspouse <- spouse[spouselist[sprows,3] == 3-sex] # 1-2 or 2-1
rspouse <- spouse[spouselist[sprows,3] == sex]   # 1-1 or 2-2
if (any(spouselist[sprows,3] ==0)) {
    #Not yet decided spouses
    indx <- which(spouselist[sprows,3] ==0)
    nleft <- floor((length(sprows) + (sex==2))/2) #total number to left
    nleft <- nleft - length(lspouse)  #number of undecideds to the left
    if (nleft >0) {
        lspouse <- c(lspouse, spouse[indx[1:nleft]])
        indx <- indx[-(1:nleft)]
      }
    if (length(indx)) rspouse <- c(spouse[indx], rspouse)
  }

nid[lev,] <- c(lspouse, x, rspouse)
nid[lev, 1:nspouse] <- nid[lev, 1:nspouse] + .5  #marriages

spouselist <- spouselist[-sprows,, drop=FALSE]
```

The spouses are in the pedigree, now look below. For each spouse get the list of children. If there are any we call alignped2 to generate their tree and then mark the connection to their parent. If multiple marriages have children we need to join the trees.

⟨*alignped1*⟩+≡

```
nokids <- TRUE   #haven't found any kids yet
spouse <- c(lspouse, rspouse)  #reorder
for (i in 1:nspouse) {
    ispouse <- spouse[i]
    children <- which((dad==x & mom==ispouse) | (dad==ispouse & mom==x))
    if (length(children) > 0) {
        rval1 <- alignped2(children, dad, mom, level, horder,
                           packed, spouselist)
```

```
        spouselist <- rval1$spouselist
        # set the parentage for any kids
        #  a nuisance: it's possible to have a child appear twice, when
        #  via inbreeding two children marry --- makes the "indx" line
        #  below more complicated
        temp <- floor(rval1$nid[lev+1,])  # cut off the .5's for matching
        indx <- (1:length(temp))[match(temp,children, nomatch=0) >0]
        rval1$fam[lev+1,indx] <- i   #set the kids parentage
        if (!packed) {
            # line the kids up below the parents
            # The advantage at this point: we know that there is
            #   nothing to the right that has to be cared for
            kidmean <- mean(rval1$pos[lev+1, indx])
            parmean <- mean(pos[lev, i + 0:1])
            if (kidmean > parmean) {
                # kids to the right of parents: move the parents
                indx <- i:(nspouse+1)
                pos[lev, indx] <- pos[lev, indx] + (kidmean - parmean)
                }
            else {
                # move the kids and their spouses and all below
                shift <- parmean - kidmean
                for (j in (lev+1):maxlev) {
                    jn <- rval1$n[j]
                    if (jn>0)
                        rval1$pos[j, 1:jn] <- rval1$pos[j, 1:jn] +shift
                    }
                }
            }
        if (nokids) {
            rval <- rval1
            nokids <- FALSE
            }
        else {
            rval <- alignped3(rval, rval1, packed)
            }
        }
    }
```

To finish up we need to splice together the tree made up from all the kids, which only has data from lev+1 down, with the data here. There are 3 cases. The first and easiest is when no children were found. The second, and most common, is when the tree below is wider than the tree here, in which case we add the data from this level onto theirs. The third is when below is narrower, for instance an only child.

⟨*alignped1*⟩+≡

```
    if (nokids) {
```

```
        return(list(nid=nid, pos=pos, fam=fam, n=n, spouselist=spouselist))
        }

    if (ncol(rval$nid) >= 1+nspouse) {
        # The rval list has room for me!
        rval$n[lev] <- n[lev]
        indx <- 1:(nspouse+1)
        rval$nid[lev, indx] <- nid[lev,]
        rval$pos[lev, indx] <- pos[lev,]
        }
    else {
        #my structure has room for them
        indx <- 1:ncol(rval$nid)
        rows <- (lev+1):maxlev
        n[rows] <- rval$n[rows]
        nid[rows,indx] <- rval$nid[rows,]
        pos[rows,indx] <- rval$pos[rows,]
        fam[rows,indx] <- rval$fam[rows,]
        rval <- list(nid=nid, pos=pos, fam=fam, n=n)
        }
    rval$spouselist <- spouselist
    rval
    }
```

4.4 alignped2

This routine takes a collection of siblings, grows the tree for each, and appends them side by side into a single tree. The input arguments are the same as those to `alignped1` with the exception that `x` will be a vector. This routine does nothing to the spouselist matrix, but needs to pass it down the tree and back since one of the routines called by `alignped2` might change the matrix.

The code below has one non-obvious special case. Suppose that two sibs marry. When the first sib is processed by `alignped1` then both partners (and any children) will be added to the rval structure below. When the second sib is processed they will come back as a 1 element tree (the marriage will no longer be on the spouselist), which should *not* be added onto rval. The rule thus is to not add any 1 element tree whose value (which must be x[i]) is already in the rval structure for this level. (Where did Curtis O. *find* these families?)

⟨*alignped2*⟩≡

```
alignped2 <- function(x, dad, mom, level, horder, packed,
                      spouselist) {
    x <- x[order(horder[x])]  # Use the hints to order the sibs
    rval <- alignped1(x[1],  dad, mom, level, horder, packed,
                      spouselist)
    spouselist <- rval$spouselist

    if (length(x) >1) {
```

```
    mylev <- level[x[1]]
    for (i in 2:length(x)) {
        rval2 <-  alignped1(x[i], dad, mom, level,
                            horder, packed, spouselist)
        spouselist <- rval2$spouselist

        # Deal with the unusual special case:
        if ((rval2$n[mylev] > 1) ||
                      (is.na(match(x[i], floor(rval$nid[mylev,])))))
            rval <- alignped3(rval, rval2, packed)
        }
    rval$spouselist <- spouselist
    }
rval
}
```

4.5 alignped3

The third co-routine merges two pedigree trees which are side by side into a single object. The primary special case is when the rightmost person in the left tree is the same as the leftmost person in the right tree; we needn't plot two copies of the same person side by side. (When initializing the output structures don't worry about this - there is no harm if they are a column bigger than finally needed.) Beyond that the work is simple bookkeeping.

⟨*alignped3*⟩≡

```
alignped3 <- function(x1, x2, packed, space=1) {
    maxcol <- max(x1$n + x2$n)
    maxlev <- length(x1$n)
    n1 <- max(x1$n)   # These are always >1
    n  <- x1$n + x2$n

    nid <- matrix(0, maxlev, maxcol)
    nid[,1:n1] <- x1$nid

    pos <- matrix(0.0, maxlev, maxcol)
    pos[,1:n1] <- x1$pos

    fam <- matrix(0, maxlev, maxcol)
    fam[,1:n1] <- x1$fam
    fam2 <- x2$fam
    if (!packed) {
        ⟨align3-slide⟩
        }
    ⟨align3-merge⟩

    if (max(n) < maxcol) {
```

```
        maxcol <- max(n)
        nid <- nid[,1:maxcol]
        pos <- pos[,1:maxcol]
        fam <- fam[,1:maxcol]
        }

    list(n=n, nid=nid, pos=pos, fam=fam)
    }
```

For the unpacked case, which is the traditional way to draw a pedigree when we can assume the paper is infinitely wide, all parents are centered over their children. In this case we think if the two trees to be merged as solid blocks. On input they both have a left margin of 0. Compute how far over we have to slide the right tree.

⟨*align3-slide*⟩≡

```
slide <- 0
for (i in 1:maxlev) {
    n1 <- x1$n[i]
    n2 <- x2$n[i]
    if (n1 >0 & n2 >0) {
        if (nid[i,n1] == x2$nid[i,1])
                temp <- pos[i, n1] - x2$pos[i,1]
        else    temp <- space + pos[i, n1] - x2$pos[i,1]
        if (temp > slide) slide <- temp
        }
    }
```

Now merge the two trees. Start at the top level and work down.

1. If n2=0, there is nothing to do
2. Decide if there is a subject overlap, and if so
   - Set the proper parent id. Only one of the two copies will be attached and the other will have fam=0, so max(fam, fam2) preserves the correct one.
   - If not packed, set the position. Choose the one connected to a parent, or midway for a double marriage.
3. If packed=TRUE determine the amount of slide for this row. It will be `space` over from the last element in the left pedigree, less overlap.
4. Move everything over
5. Fix all the children of this level, right hand pedigree, to point to the correct parental position.

⟨*align3-merge*⟩≡

```
for (i in 1:maxlev) {
    n1 <- x1$n[i]
    n2 <- x2$n[i]
```

```
if (n2 >0) {   # If anything needs to be done for this row...
    if (n1>0 && (nid[i,n1] == floor(x2$nid[i,1]))) {
        #two subjects overlap
        overlap <- 1
        fam[i,n1] <- max(fam[i,n1], fam2[i,1])
        nid[i,n1] <- max(nid[i,n1], x2$nid[i,1]) #preserve a ".5"
        if (!packed) {
            if(fam2[i,1]>0)
                if (fam[i,n1]>0)
                    pos[i,n1] <- (x2$pos[i,1] + pos[i,n1] + slide)/2
                else pos[i,n1] <- x2$pos[i,1]+ slide
                }
        n[i] <- n[i] -1
        }
    else overlap <- 0

    if (packed) slide <- if (n1==0) 0 else pos[i,n1] + space - overlap

    zz <- seq(from=overlap+1, length=n2-overlap)
    nid[i, n1 + zz- overlap] <- x2$nid[i, zz]
    fam[i, n1 + zz -overlap] <- fam2[i,zz]
    pos[i, n1 + zz -overlap] <- x2$pos[i,zz] + slide

    if (i<maxlev) {
        # adjust the pointers of any children (look ahead)
        temp <- fam2[i+1,]
        fam2[i+1,] <- ifelse(temp==0, 0, temp + n1 -overlap)
        }
    }
}
```

5 alignped4

The alignped4 routine is the final step of alignment. It attempts to line up children under parents and put spouses and siblings 'close' to each other,to the extent possible within the constraints of page width. This routine used to be the most intricate and complex of the set, until I realized that the task could be cast as constrained quadradic optimization. The current code does necessary setup and then calls the `quadprog` function. At one point I investigated using one of the simpler least-squares routines where β is constrained to be non-negative. However a problem can only be translated into that form if the number of constraints is less than the number of parameters, which is not true in this problem.

There are two important parameters for the function. One is the user specified maximum width. The smallest possible width is the maximum number of subjects on a line, if the user's suggestion is too low it is increased to that 1+ that amount (to give just a little wiggle room). The other is a vector of 2 alignment parameters a and b. For each set of siblings x with parents

at p_1 and p_2 the alignment penalty is

$$(1/k^a) \sum i = 1k(x_i - (p_1 + p_2)^2$$

where k is the number of siblings in the set. Using the fact that $\sum(x_i - c)^2 = \sum(x_i - \mu)^2 + k(c - \mu)^2$, when $a = 1$ then moving a sibship with k sibs one unit to the left or right of optimal will incur the same cost as moving one with only 1 or two sibs out of place. If $a = 0$ then large sibships are harder to move than small ones, with the default value $a = 1.5$ they are slightly easier to move than small ones. The rationale for the default is as long as the parents are somewhere between the first and last siblings the result looks fairly good, so we are more flexible with the spacing of a large family. By tethering all the sibs to a single spot they tend are kept close to each other. The alignment penalty for spouses is $b(x_1 - x_2)^2$, which tends to keep them together. The size of b controls the relative importance of sib-parent and spouse-spouse closeness.

We start by adding in these penalties. The total number of parameters in the alignment problem (what we hand to quadprog) is the set of `sum(n)` positions. A work array myid keeps track of the parameter number for each position so that it is easy to find. There is one extra penalty added at the end. Because the penalty amount would be the same if all the final positions were shifted by a constant, the penalty matrix will not be positive definite; solve.QP doesn't like this. We add a tiny amount of leftward pull to the widest line.

⟨*alignped4*⟩≡

```
alignped4 <- function(rval, spouse, level, width, align) {
    if (is.logical(align)) align <- c(1.5, 2)  #defaults
    maxlev <- nrow(rval$nid)
    width <- max(width, rval$n+.01)   # width must be > the longest row

    n <- sum(rval$n)  # total number of subjects
    myid <- matrix(0, maxlev, ncol(rval$nid))  #number the plotting points
    for (i in 1:maxlev) {
        myid[i, rval$nid[i,]>0] <-  cumsum(c(0, rval$n))[i] + 1:rval$n[i]
        }

    # There will be one penalty for each spouse and one for each child
    npenal <- sum(spouse[rval$nid>0]) + sum(rval$fam >0)
    pmat <- matrix(0., nrow=npenal+1, ncol=n)

    indx <- 0
    # Penalties to keep spouses close
    for (lev in 1:maxlev) {
        if (any(spouse[lev,])) {
            who <- which(spouse[lev,])
            indx <- max(indx) + 1:length(who)
            pmat[cbind(indx, myid[lev,who])] <-  sqrt(align[2])
            pmat[cbind(indx, myid[lev,who+1])] <- -sqrt(align[2])
            }
        }
```

```
# Penalties to keep kids close to parents
for (lev in (1:maxlev)[-1])  { # no parents at the top level
    families <- unique(rval$fam[lev,])
    families <- families[families !=0]  #0 is the 'no parent' marker
    for (i in families) {  #might be none
        who <- which(rval$fam[lev,] == i)
        k <- length(who)
        indx <- max(indx) +1:k   #one penalty per child
        penalty <- sqrt(k^(-align[1]))
        pmat[cbind(indx, myid[lev,who])] <- -penalty
        pmat[cbind(indx, myid[lev-1, rval$fam[lev,who]])] <- penalty/2
        pmat[cbind(indx, myid[lev-1, rval$fam[lev,who]+1])] <- penalty/2
        }
    }
maxrow <- min(which(rval$n==max(rval$n)))
pmat[nrow(pmat), myid[maxrow,1]] <- 1e-5
```

Next come the constraints. If there are k subjects on a line there will be $k+1$ constraints for that line. The first point must be ≥ 0, each subesquent one must be at least 1 unit to the right, and the final point must be $\leq$ the max width.

⟨*alignped4*⟩+≡

```
ncon <- n + maxlev    # number of constraints
cmat <- matrix(0., nrow=ncon, ncol=n)
coff <- 0  # cumulative constraint lines so var
dvec <- rep(1., ncon)
for (lev in 1:maxlev) {
    nn <- rval$n[lev]
    if (nn>1) {
        for (i in 1:(nn-1))
            cmat[coff +i, myid[lev,i + 0:1]] <- c(-1,1)
        }

    cmat[coff+nn,   myid[lev,1]]  <- 1     #first element >=0
    dvec[coff+nn] <- 0
    cmat[coff+nn+1, myid[lev,nn]] <- -1    #last element <= width-1
    dvec[coff+nn+1] <- 1-width
    coff <- coff + nn+ 1
    }

if (exists('solve.QP')) {
     pp <- t(pmat) %*% pmat + 1e-8 * diag(ncol(pmat))
     fit <- solve.QP(pp, rep(0., n), t(cmat), dvec)
     }
else stop("Need the quadprog package")

newpos <- rval$pos
```

```
    #fit <- lsei(pmat, rep(0, nrow(pmat)), G=cmat, H=dvec)
    #newpos[myid>0] <- fit$X[myid]
    newpos[myid>0] <- fit$solution[myid]
    newpos
    }
```

6 Plots

The plotting function for pedigrees has 5 tasks

1. Gather information and check the data. An important step is the call to align.pedigree.
2. Set up the plot region and size the symbols. The program wants to plot circles and squares, so needs to understand the geometry of the paper, pedigree size, and text size to get the right shape and size symbols.
3. Set up the plot and add the symbols for each subject
4. Add connecting lines between spouses, and children with parents
5. Create an invisible return value containing the locations.

Another task, not yet completely understood, is how we might break a plot across multiple pages.

⟨*plot.pedigree*⟩≡

```
plot.pedigree <- function(x, id = x$id, status = x$status,
                          affected = x$affected,
                          cex = 1, col = 1,
                          symbolsize = 1, branch = 0.6,
                          packed = TRUE, align = c(1.5,2), width = 8,
                          density=c(-1, 35,55,25), mar=c(4.1, 1, 4.1, 1),
                          angle=c(90,65,40,0), keep.par=FALSE,
                          subregion, ...)
{
    Call <- match.call()
    n <- length(x$id)
    ⟨pedplot-data⟩
    ⟨pedplot-sizing⟩
    ⟨pedplot-symbols⟩
    ⟨pedplot-lines⟩
    ⟨pedplot-finish⟩
    }
```

6.1 Setup

The dull part is first: check all of the input data for correctness. The `sex` variable is taken from the pedigree so we need not check that. The identifier for each subject is by default the `id` variable from the pedigree, but users often want to add some extra text. The status variable can be used to put a line through the symbol of those who are deceased, it is an optional part of the pedigree.

⟨*pedplot-data*⟩≡

```
if(is.null(status))
  status <- rep(0, n)
else {
    if(!all(status == 0 | status == 1))
      stop("Invalid status code")
    if(length(status) != n)
      stop("Wrong length for status")
}
if(!missing(id)) {
    if(length(id) != n)
      stop("Wrong length for id")
}
```

The "affected status" is a 0/1 matrix of any marker data that the user might want to add. It may be attached to the pedigree or added here. It can be a vector of length `n` or a matrix with `n` rows. If it is not present, the default is to print open symbols without shading or color, which corresponds to a code of 0, while a 1 means to shade the symbol.

If the argment is a matrix, then the shading and/or density value for ith column is taken from the ith element of the angle/density arguments.

(Update by JPS 5/2011) Update to allow missing values (NA) in the "affected" indicators. Missingness of affection status will have a "?" in the midpoint of the portion of the plot symbol rather than blank or shaded. The "?" is in line with standards discussed in Bennet et a. J of Gent Couns., 2008.

For purposes within the plot method, NA values in "affected" are coded to -1.

⟨*pedplot-data*⟩+≡

```
if(is.null(affected)){
  affected <- matrix(0,nrow=n)
}
else {
    if (is.matrix(affected)){
        if (nrow(affected) != n) stop("Wrong number of rows in affected")
        if (is.logical(affected)) affected <- 1* affected
        if (ncol(affected) > length(angle) || ncol(affected) > length(density))
            stop("More columns in the affected matrix than angle/density values")
        }
    else {
        if (length(affected) != n)
        stop("Wrong length for affected")
```

```
        if (is.logical(affected)) affected <- as.numeric(affected)
        if (is.factor(affected))  affected <- as.numeric(affected) -1
        }
    if(max(affected, na.rm=TRUE) > min(affected, na.rm=TRUE)) {
      affected <- matrix(affected - min(affected, na.rm=TRUE),nrow=n)
      affected[is.na(affected)] <- -1
    } else {
      affected <- matrix(affected,nrow=n)
    }
    if (!all(affected==0 | affected==1 | affected == -1))
        stop("Invalid code for affected status")
}

if (length(col) ==1) col <- rep(col, n)
else if (length(col) != n) stop("Col argument must be of length 1 or n")
```

6.2 Sizing

Now we need to set the sizes. From align.pedigree we will get the maximum width and depth. There is one plotted row for each row of the returned matrices. The number of columns of the matrices is the max width of the pedigree, so there are unused positions in shorter rows, these can be identifed by having an nid value of 0. Horizontal locations for each point go from 0 to xmax, subjects are at least 1 unit apart; a large number will be exactly one unit part. These locations will be at the top center of each plotted symbol.

⟨*pedplot-sizing*⟩≡

```
⟨pedplot-subregion⟩
plist <- align.pedigree(x, packed = packed, width = width, align = align)
if (!missing(subregion)) plist <- subregion2(plist, subregion)
xrange <- range(plist$pos[plist$nid >0])
maxlev <- nrow(plist$pos)
```

We would like to to make the boxes about 2.5 characters wide, which matches most labels, but no more than 0.9 units wide or .5 units high. We also want to vertical room for the labels. Which should have at least 1/2 of stemp2 space above and stemp2 space below. The stemp3 variable is the height of labels: users may use multi-line ones. Our constraints then are

- (box height + label height)*maxlev $\leq$ height: the boxes and labels have to fit vertically
- (box height) * (maxlev + (maxlev-1)/2) $\leq$ height: at least 1/2 a box of space between each row of boxes
- (box width) $\leq$ stemp1 in inches
- (box width) $\leq$ 0.8 unit in user coordinates, otherwise they appear to touch
- User coordinates go from min(xrange)- 1/2 box width to max(xrange) + 1/2 box width.

- the box is square (in inches)

The first 3 of these are easy. The fourth comes into play only for very packed pedigrees. Assume that the box were the maximum size of .8 units, i.e., minimal spacing between them. Then xmin -.45 to xmax + .45 covers the plot region, the scaling between user coordinates and inches is (.8 + xmax-xmin) user = figure region inches, and the box is .8*(figure width)/(.8 + xmax-xmin). The transformation from user units to inches horizontally depends on the box size, since I need to allow for 1/2 a box on the left and right. Vertically the range from 1 to nrow spans the tops of the symbols, which will be the figure region height less (the height of the text for the last row + 1 box); remember that the coordinates point to the top center of the box. We want row 1 to plot at the top, which is done by appropriate setting of the usr parameter.

⟨*pedplot-sizing*⟩+≡
```
frame()
oldpar <- par(mar=mar, xpd=TRUE)
psize <- par('pin')  # plot region in inches
stemp1 <- strwidth("ABC", units='inches', cex=cex)* 2.5/3
stemp2 <- strheight('1g', units='inches', cex=cex)
stemp3 <- max(strheight(id, units='inches', cex=cex))

ht1 <- psize[2]/maxlev - (stemp3 + 1.5*stemp2)
if (ht1 <=0) stop("Labels leave no room for the graph, reduce cex")
ht2 <- psize[2]/(maxlev + (maxlev-1)/2)
wd2 <- .8*psize[1]/(.8 + diff(xrange))

boxsize <- symbolsize* min(ht1, ht2, stemp1, wd2) # box size in inches
hscale <- (psize[1]- boxsize)/diff(xrange)  #horizontal scale from user-> inch
vscale <- (psize[2]-(stemp3 + stemp2/2 + boxsize))/ max(1, maxlev-1)
boxw  <- boxsize/hscale  # box width in user units
boxh  <- boxsize/vscale   # box height in user units
labh  <- stemp2/vscale   # height of a text string
legh  <- min(1/4, boxh  *1.5)  # how tall are the 'legs' up from a child
par(usr=c(xrange[1]- boxw/2, xrange[2]+ boxw/2,
          maxlev+ boxh+ stemp3 + stemp2/2 , 1))
```

6.3 Drawing the tree

Now we draw and label the boxes. Definition of the drawbox function is deferred until later.

⟨*pedplot-symbols*⟩≡
```
⟨pedplot-drawbox⟩

sex <- as.numeric(x$sex)
for (i in 1:maxlev) {
    for (j in 1:plist$n[i]) {
        k <- plist$nid[i,j]
        drawbox(plist$pos[i,j], i, sex[k], affected[k,],
                status[k], col[k], polylist, density, angle,
```

```
                boxw, boxh)
        text(plist$pos[i,j], i + boxh + labh*.7, id[k], cex=cex, adj=c(.5,1))
        }
}
```

Now draw in the connections one by one. First those between spouses.

⟨*pedplot-lines*⟩≡

```
maxcol <- ncol(plist$nid)  #all have the same size
for(i in 1:maxlev) {
    tempy <- i + boxh/2
    if(any(plist$spouse[i,  ]>0)) {
        temp <- (1:maxcol)[plist$spouse[i,  ]>0]
        segments(plist$pos[i, temp] + boxw/2, rep(tempy, length(temp)),
             plist$pos[i, temp + 1] - boxw/2, rep(tempy, length(temp)))

        temp <- (1:maxcol)[plist$spouse[i,  ] ==2]
        if (length(temp)) { #double line for double marriage
            tempy <- tempy + boxh/10
            segments(plist$pos[i, temp] + boxw/2, rep(tempy, length(temp)),
               plist$pos[i, temp + 1] - boxw/2, rep(tempy, length(temp)))
            }
    }
}
```

Now connect the children to the parents. First there are lines up from each child, which would be trivial except for twins, triplets, etc. Then we draw the horizontal bar across siblings and finally the connector from the parent. For twins, the "vertical" lines are angled towards a common point, the variable is called `target` below. The horizontal part is easier if we do things family by family. The `plist$twins` variable is 1/2/3 for a twin on my right, 0 otherwise.

⟨*pedplot-lines*⟩+≡

```
for(i in 2:maxlev) {
    zed <- unique(plist$fam[i,  ])
    zed <- zed[zed > 0]  #list of family ids

    for(fam in zed) {
        xx <- plist$pos[i - 1, fam + 0:1]
        parentx <- mean(xx)   #midpoint of parents


        # Draw the uplines
        who <- (plist$fam[i,] == fam) #The kids of interest
        if (is.null(plist$twins)) target <- plist$pos[i,who]
        else {
            twin.to.left <-(c(0, plist$twins[i,who])[1:sum(who)])
            temp <- cumsum(twin.to.left ==0) #increment if no twin to the left
            # 5 sibs, middle 3 are triplets gives 1,2,2,2,3
            # twin, twin, singleton gives 1,1,2,2,3
```

```
        tcount <- table(temp)
        target <- rep(tapply(plist$pos[i,who], temp, mean), tcount)
        }
    yy <- rep(i, sum(who))
    segments(plist$pos[i,who], yy, target, yy-legh)

    ## draw midpoint MZ twin line
    if (any(plist$twins[i,who] ==1)) {
      who2 <- which(plist$twins[i,who] ==1)
      temp1 <- (plist$pos[i, who][who2] + target[who2])/2
      temp2 <- (plist$pos[i, who][who2+1] + target[who2])/2
        yy <- rep(i, length(who2)) - legh/2
        segments(temp1, yy, temp2, yy)
        }

    # Add a question mark for those of unknown zygosity
    if (any(plist$twins[i,who] ==3)) {
      who2 <- which(plist$twins[i,who] ==3)
      temp1 <- (plist$pos[i, who][who2] + target[who2])/2
      temp2 <- (plist$pos[i, who][who2+1] + target[who2])/2
        yy <- rep(i, length(who2)) - legh/2
        text((temp1+temp2)/2, yy, '?')
        }

    # Add the horizontal line
    segments(min(target), i-legh, max(target), i-legh)

    # Draw line to parents
    x1 <- mean(range(target))
    y1 <- i-legh
    if(branch == 0)
        segments(x1, y1, parentx, (i-1) + boxh/2)
    else {
        y2 <- (i-1) + boxh/2
        x2 <- parentx
        ydelta <- ((y2 - y1) * branch)/2
        segments(c(x1, x1, x2), c(y1, y1 + ydelta, y2 - ydelta),
                 c(x1, x2, x2), c(y1 + ydelta, y2 - ydelta, y2))
        }
    }
}
```

The last set of lines are dotted arcs that connect mulitiple instances of a subject on the same line. These instances may or may not be on the same line. The arrcconect function draws a quadratic arc between locations (x_1, y_1) and (x_2, y_2) whose height is 1/2 unit above a straight line connection.

⟨*pedplot-lines*⟩+≡
```
  arcconnect <- function(x, y) {
      xx <- seq(x[1], x[2], length = 15)
      yy <- seq(y[1], y[2], length = 15) + (seq(-7, 7))^2/98 - .5
      lines(xx, yy, lty = 2)
      }

  uid <- unique(plist$nid)
  for (id in uid[uid>0]) {
      indx <- which(plist$nid == id)
      if (length(indx) >1) {   #subject is a multiple
          tx <- plist$pos[indx]
          ty <- ((row(plist$pos))[indx])[order(tx)]
          tx <- sort(tx)
          for (j in 1:(length(indx) -1))
              arcconnect(tx[j + 0:1], ty[j+  0:1])
          }
      }
```

6.4 Final output

Remind the user of subjects who did not get plotted; these are ususally subjects who are married in but without children. Unless the pedigree contains spousal information the routine does not know who is the spouse. Then restore the plot parameters. This would only not be done if someone wants to further annotate the plot. Last, we give a list of the plot positions for each subject. Someone who is plotted twice will have their first position listed.

⟨*pedplot-finish*⟩≡
```
  ckall <- x$id[is.na(match(x$id,x$id[plist$nid]))]
  if(length(ckall>0)) cat('Did not plot the following people:',ckall,'\n')

  if(!keep.par) par(oldpar)

  tmp <- match(1:length(x$id), plist$nid)
  invisible(list(plist=plist, x=plist$pos[tmp], y= row(plist$pos)[tmp],
                 boxw=boxw, boxh=boxh, call=Call))
```

6.5 Symbols

There are four sumbols corresponding to the four sex codes: square = male, circle = female, diamond= unknown, and triangle = terminated. They are shaded according to the value(s) of affected status for each subject, where 0=unfilled and 1=filled, and filling uses the standard arguments of the `polygon` function. The nuisance is when the affected status is a matrix, in which case the symbol will be divided up into sections, clockwise starting at the lower left. I asked Beth about this (original author) and there was no particular reason to start at 6 o'clock, but it's now established as history.

The first part of the code is to create the collection of polygons that will make up the symbol. These are then used again and again. The collection is kept as a list with the four elements square, circle, diamond and triangle. Each of these is in turn a list with ncol(affected) element, and each of those in turn a list of x and y coordinates. There are 3 cases: the affected matrix has only one column, partitioning a circle for multiple columns, and partitioning the other cases for multiple columns.

⟨*pedplot-drawbox*⟩≡
```
⟨pedplot-circfun⟩
⟨pedplot-polyfun⟩
if (ncol(affected)==1) {
    polylist <- list(
        square = list(list(x=c(-1, -1, 1,1)/2,  y=c(0, 1, 1, 0))),
        circle = list(list(x=.5* cos(seq(0, 2*pi, length=50)),
                           y=.5* sin(seq(0, 2*pi, length=50)) + .5)),
        diamond = list(list(x=c(0, -.5, 0, .5), y=c(0, .5, 1, .5))),
        triangle= list(list(x=c(0, -.56, .56),  y=c(0, 1, 1))))
    }
else {
    nc <- ncol(affected)
    square <- polyfun(nc, list(x=c(-.5, -.5, .5, .5), y=c(-.5, .5, .5, -.5),
                               theta= -c(3,5,7,9)* pi/4))
    circle <- circfun(nc)
    diamond <-  polyfun(nc, list(x=c(0, -.5, 0, .5), y=c(-.5, 0, .5,0),
                               theta= -(1:4) *pi/2))
    triangle <- polyfun(nc, list(x=c(-.56, .0, .56), y=c(-.5, .5, -.5),
                                theta=c(-2, -4, -6) *pi/3))
    polylist <- list(square=square, circle=circle, diamond=diamond,
                     triangle=triangle)
    }
```

The circle function is quite simple. The number of segments is arbitrary, 50 seems to be enough to make the eye happy. We draw the ray from 0 to the edge, then a portion of the arc. The polygon function will connect back to the center.

⟨*pedplot-circfun*⟩≡
```
circfun <- function(nslice, n=50) {
    nseg <- ceiling(n/nslice)  #segments of arc per slice

    theta <- -pi/2 - seq(0, 2*pi, length=nslice +1)
    out <- vector('list', nslice)
    for (i in 1:nslice) {
        theta2 <- seq(theta[i], theta[i+1], length=nseg)
        out[[i]]<- list(x=c(0, cos(theta2)/2),
                        y=c(0, sin(theta2)/2) + .5)
        }
    out
    }
```

Now for the interesting one — dividing a polygon into "pie slices". In computing this we can't use the usual $y = a + bx$ formula for a line, because it doesn't work for vertical ones (like the sides of the square). Instead we use the alternate formulation in terms of a dummy variable z.

$$\begin{aligned} x &= a + bz \\ y &= c + dz \end{aligned}$$

Furthermore, we choose the constants a, b, c, and d so that the side of our polygon correspond to $0 \le z \le 1$. The intersection of a particular ray at angle theta with a particular side will satisfy

$$\begin{aligned} theta &= y/x = \frac{a + bz}{c + dz} \\ z &= \frac{a\theta - c}{b - d\theta} \end{aligned} \tag{1}$$

(2)

Equation 1 will lead to a division by zero if the ray from the origin does not intersect a side, e.g., a vertical divider will be parallel to the sides of a square symbol. The only solutions we want have $0 \le z \le 1$ and are in the 'forward' part of the ray. This latter is true if the inner product $x\cos(\theta) + y\sin(\theta) > 0$. Exactly one of the polygon sides will satisfy both conditions.

⟨*pedplot-polyfun*⟩≡

```
polyfun <- function(nslice, object) {
    # make the indirect segments view
    zmat <- matrix(0,ncol=4, nrow=length(object$x))
    zmat[,1] <- object$x
    zmat[,2] <- c(object$x[-1], object$x[1]) - object$x
    zmat[,3] <- object$y
    zmat[,4] <- c(object$y[-1], object$y[1]) - object$y

    # Find the cutpoint for each angle
    #   Yes we could vectorize the loop, but nslice is never bigger than
    # about 10 (and usually <5), so why be obscure?
    ns1 <- nslice+1
    theta <- -pi/2 - seq(0, 2*pi, length=ns1)
    x <- y <- double(ns1)
    for (i in 1:ns1) {
        z <- (tan(theta[i])*zmat[,1] - zmat[,3])/
            (zmat[,4] - tan(theta[i])*zmat[,2])
        tx <- zmat[,1] + z*zmat[,2]
        ty <- zmat[,3] + z*zmat[,4]
        inner <- tx*cos(theta[i]) + ty*sin(theta[i])
        indx <- which(is.finite(z) & z>=0 &  z<=1 & inner >0)
        x[i] <- tx[indx]
        y[i] <- ty[indx]
        }
```

Now I have the x, y coordinates where each radial slice (the cuts you would make when slicing a pie) intersects the polygon. Add the original vertices of the polygon to the list, sort by angle, and create the output. The radial lines are labeled 1,2, ..., nslice +1 (the original cut from the center to 6 o'clock is repeated at the end), and the inserted vertices with a zero.

⟨*pedplot-polyfun*⟩+≡

```
    nvertex <- length(object$x)
    temp <- data.frame(indx = c(1:ns1, rep(0, nvertex)),
                       theta= c(theta, object$theta),
                       x= c(x, object$x),
                       y= c(y, object$y))
    temp <- temp[order(-temp$theta),]
    out <- vector('list', nslice)
    for (i in 1:nslice) {
        rows <- which(temp$indx==i):which(temp$indx==(i+1))
        out[[i]] <- list(x=c(0, temp$x[rows]), y= c(0, temp$y[rows]) +.5)
        }
    out
    }
```

Finally we get to the drawbox function itself, which is fairly simple. Updates by JPS in 5/2011 to allow missing, and to fix up shadings and borders. For affected=0, don't fill. For affected=1, fill with density-lines and angles. For affected=-1 (missing), fill with "?" in the midpoint of the polygon, with a size adjusted by how many columns in affected. For all shapes drawn, make the border the color for the person.

⟨*pedplot-drawbox*⟩+≡

```
drawbox<- function(x, y, sex, affected, status, col, polylist,
          density, angle, boxw, boxh) {
      for (i in 1:length(affected)) {
          if (affected[i]==0) {
              polygon(x + (polylist[[sex]])[[i]]$x *boxw,
                      y + (polylist[[sex]])[[i]]$y *boxh,
                      col=NA, border=col)
              }

          if(affected[i]==1) {
            ## else {
            polygon(x + (polylist[[sex]])[[i]]$x * boxw,
                    y + (polylist[[sex]])[[i]]$y * boxh,
                    col=col, border=col, density=density[i], angle=angle[i])
          }
          if(affected[i] == -1) {
            polygon(x + (polylist[[sex]])[[i]]$x * boxw,
                    y + (polylist[[sex]])[[i]]$y * boxh,
                    col=NA, border=col)
```

```
            midx <- x + mean(range(polylist[[sex]][[i]]$x*boxw))
            midy <- y + mean(range(polylist[[sex]][[i]]$y*boxh))

            points(midx, midy, pch="?", cex=min(1, cex*2/length(affected)))
          }

        }
      if (status==1) segments(x- .6*boxw, y+1.1*boxh,
                              x+ .6*boxw, y- .1*boxh,)
      ## Do a black slash per Beth, old line was
      ##        x+ .6*boxw, y- .1*boxh, col=col)
    }
```

6.6 Subsetting

This section is still experimental and might change.

Sometimes a pedigree is too large to fit comfortably on one page. The `subregion` argument allows one to plot only a portion of the pedigree based on the plot region. Along with other tools to select portions of the pedigree based on relatedness, such as all the descendents of a particular marriage, it gives a tool for addressing this. This breaks our original goal of completely automatic plots, but users keep asking for more.

The argument is `subregion=c(min x, max x, min depth, max depth)`, and works by editing away portions of the `plist` object returned by align.pedigree. First decide what lines to keep. Then take subjects away from each line, update spouses and twins, and fix up parentage for the line below.

JPS 5/23/2011 note: Found the subregion option to re-scale the y-axis very well, but not the x-axis.

⟨*pedplot-subregion*⟩≡

```
  subregion2 <- function(plist, subreg) {
      if (subreg[3] <1 || subreg[4] > length(plist$n))
          stop("Invalid depth indices in subreg")
      lkeep <- subreg[3]:subreg[4]
      for (i in lkeep) {
          if (!any(plist$pos[i,]>=subreg[1] & plist$pos[i,] <= subreg[2]))
              stop(paste("No subjects retained on level", i))
          }

      nid2 <- plist$nid[lkeep,]
      n2   <- plist$n[lkeep]
      pos2 <- plist$pos[lkeep,]
      spouse2 <- plist$spouse[lkeep,]
      fam2 <- plist$fam[lkeep,]
      if (!is.null(plist$twins)) twin2 <- plist$twins[lkeep,]
```

```
for (i in 1:nrow(nid2)) {
    keep <- which(pos2[i,] >=subreg[1] & pos2[i,] <= subreg[2])
    nkeep <- length(keep)
    n2[i] <- nkeep
    nid2[i, 1:nkeep] <- nid2[i, keep]
    pos2[i, 1:nkeep] <- pos2[i, keep]
    spouse2[i,1:nkeep] <- spouse2[i,keep]
    fam2[i, 1:nkeep] <- fam2[i, keep]
    if (!is.null(plist$twins)) twin2[i, 1:nkeep] <- twin2[i, keep]

    if (i < nrow(nid2)) {  #look ahead
        tfam <- match(fam2[i+1,], keep, nomatch=0)
        fam2[i+1,] <- tfam
        if (any(spouse2[i,tfam] ==0))
            stop("A subregion cannot separate parents")
        }
    }

n <- max(n2)
out <- list(n= n2[1:n], nid=nid2[,1:n, drop=F], pos=pos2[,1:n, drop=F],
            spouse= spouse2[,1:n, drop=F], fam=fam2[,1:n, drop=F])
if (!is.null(plist$twins)) out$twins <- twin2[, 1:n, drop=F]
out
}
```

6.7 Legends

We define a function to draw a legend for the affected matrix. We do so by making use of the pie() function, which will draw a circle that will look like a woman (circle) in the pedigree who has all affected indicators ==1. We do not show what the "?" means, and we do not cover what colors are indicated by the coloring applied to subjects.

We allow the legend to be added to the current pedigree plot by default, and it also works to draw a legend on a separate page. The *new* argument controls this option. When new=TRUE, the default, the plot is added to the current plot (assumed a pedigree plot), and placed in one of the corners of the plot given by *location*, which has options "bottomright", "topright", "topleft", and "bottomleft", with "bottomright" the default.

If new=FALSE, the pie graph is plotted from (-1,1) for both x and y, centered at 0,0 with radius 1. With angle.init=90 and twopi = 2*pi, we control the start to be at the top and the sections are plotted counter-clockwise, respectively, which are some of the settings from the original pie() function.

When we adapted the pie() function to plot in different, non-(0,0) locations on the pedigree, we had these major issues:

1) The Y-axis actually goes from min(y) at the top and max(y) at the bottom. 2) To get the polygon in pie() to not be oblong, we made sure to use asp=1, which re-sets the x- and/or y-axis again. Therefore, we have to manage the placing of the pie in reference to those updated

scalings using par("usr"). 3) We have to choose a center that is not 0,0, and have to add the center x,y coordinates to some of the default settings of pie().

We carry forward from the plot.pedigree the same density and angle defaults for shading sections of each subject's symbol with polygon.

⟨*pedigree.legend*⟩≡

```
pedigree.legend <- function (ped, labels = dimnames(ped$affected)[[2]],
    edges = 200, radius=NULL, location="bottomright", new=TRUE,
    density = c(-1, 35, 55, 25),  angle = c(90, 65, 40, 0), ...)
{

    naff <- ncol(ped$affected)
    x <- rep(1,naff)

    # Defaults for plotting on separate page:
    ## start at the top, always counter-clockwise, black/white
    init.angle <- 90
    twopi <- 2 * pi
    col <- 1

    default.labels <- paste("affected-", 1:naff, sep='')
    if (is.null(labels)) labels <- default.labels

    ## assign labels to those w/ zero-length label
    whichNoLab <- which(nchar(labels) < 1)
    if(length(whichNoLab))
      labels[whichNoLab] <- paste("affected-", whichNoLab, sep='')


    x <- c(0, cumsum(x)/sum(x))
    dx <- diff(x)
    nx <- length(dx)
    ## settings for plotting on a new page
    if(!new) {
      plot.new()

      pin <- par("pin")
      # radius, xylim, center, line-lengths set to defaults of pie()
      radius <- 1
      xlim <- ylim <- c(-1, 1)
      center <- c(0,0)
      llen <- 0.05

      if (pin[1L] > pin[2L])
        xlim <- (pin[1L]/pin[2L]) * xlim
```

```
  else ylim <- (pin[2L]/pin[1L]) * ylim

  plot.window(xlim, ylim, "", asp = 1)

} else {
  ## Settings to add to pedigree plot
  ## y-axis is flipped, so adjust angle and rotation
  init.angle <- -1*init.angle
  twopi <- -1*twopi

  ## track usr xy limits. With asp=1, it re-scales to have aspect ratio 1:1
  usr.orig <- par("usr")
  plot.window(xlim=usr.orig[1:2], ylim=usr.orig[3:4], "", asp=1)
  usr.asp1 <- par("usr")

  ## set line lengths
  llen <- radius*.15

  ## also decide on good center/radius if not given
  if(is.null(radius))
    radius <- .5

  ## get center of pie chart for coded
  pctusr <- .10*abs(diff(usr.asp1[3:4]))
  center = switch(location,
    "bottomright" = c(max(usr.asp1[1:2])-pctusr,max(usr.asp1[3:4])-pctusr),
    "topright" = c(max(usr.asp1[1:2])-pctusr, min(usr.asp1[3:4]) + pctusr),
    "bottomleft" =c(min(usr.asp1[1:2]) + pctusr, max(usr.asp1[3:4])-pctusr),
    "topleft" = c(min(usr.asp1[1:2]) + pctusr, min(usr.asp1[3:4]) + pctusr))

}

col <- rep(col, length.out = nx)
border <- rep(1, length.out = nx)
lty <- rep(1, length.out = nx)
angle <- rep(angle, length.out = nx)
density <- rep(density, length.out = nx)

t2xy <- function(t) {
    t2p <- twopi * t + init.angle * pi/180
    list(x = radius * cos(t2p), y = radius * sin(t2p))
}
for (i in 1L:nx) {
    n <- max(2, floor(edges * dx[i]))
    P <- t2xy(seq.int(x[i], x[i + 1], length.out = n))
    P$x <- P$x + center[1]
```

```
        P$y <- P$y + center[2]

        polygon(c(P$x, center[1]), c(P$y, center[2]), density = density[i],
                angle = angle[i], border = border[i], col = col[i],
                lty = lty[i])

        P <- t2xy(mean(x[i + 0:1]))
        if(new) {
          ## not centered at 0,0, so added center to x,y
          P$x <- P$x + center[1]
          P$y <- center[2] + ifelse(new, P$y, -1*P$y)
        }

        lab <- as.character(labels[i])
        if (!is.na(lab) && nzchar(lab)) {
          ## put lines
          lines(x=c(P$x, P$x + ifelse(P$x<center[1], -1*llen, llen)),
                y=c(P$y, P$y + ifelse(P$y<center[2], -1*llen, llen)))

          ##  put text just beyond line-length away from pie
          text(x=P$x + ifelse(P$x < center[1], -1.2*llen, 1.2*llen),
               y=P$y + ifelse(P$y < center[2], -1.2*llen, 1.2*llen),
               labels[i], xpd = TRUE,
               adj = ifelse(P$x < center[1], 1, 0), ...)
        }
    }

    invisible(NULL)
}
```

7 Intro to Pedigree Shrink

The pedigree.shrink functions were initially written to deal with a pedigree represented as a data.frame with pedTrim, written by Steve Iturria, to trim the subjects from a pedigree who were less useful for linkage and family association studies. It was later turned into a package called pedShrink by Daniel Schaid's group, still working on a pedigree, but assuming it was just a data.frame. Later, the functions were managed by Jason Sinnwell who worked with the 2010 version of the pedigree object by Terry Therneau in planning to group many of the pedigree functions together into an enhanced kinship package.

This file also contains the pedigree.unrelated function, developed by Dan Schaid and Shannon McDonnell, which uses the kinship matrix to determine relatedness of subjects in a pedigree, and returns the person id of one of the maximal sets of individuals that are not related. Details described below.

8 Pedigree Shrink

The pedigree.shrink function trims an object of class pedigree, and returns a list with information about how the pedigree was shrunk, and the final shrunken pedigree object.

pedigree.shrink. Accepts the following input

ped a pedigree object

avail indicator vector of availability of each person in the pedigree

seed seed to control randomness

maxBits bit size to shrink the pedigree size under

⟨*pedigree.shrink*⟩≡

```
#$Log: pedigree.shrink.q,v $
#Revision 1.5  2010/09/03 21:11:16  sinnwell
#add shrunk "avail" vector to result, keep status and affected in pedObj
#
#Revision 1.4  2010/09/03 19:15:03  sinnwell
#add avail arg which is not part of ped object.  Re-make ped object at the end with status and af
#
#Revision 1.2  2009/11/17 23:08:18  sinnwell
#*** empty log message ***
#
#Revision 1.1  2008/07/16 20:23:07  sinnwell
#Initial revision
#
#$Log: pedigree.shrink.q,v $
#Revision 1.5  2010/09/03 21:11:16  sinnwell
#add shrunk "avail" vector to result, keep status and affected in pedObj
#
#Revision 1.4  2010/09/03 19:15:03  sinnwell
#add avail arg which is not part of ped object.  Re-make ped object at the end with status and af
#
#Revision 1.2  2009/11/17 23:08:18  sinnwell
#*** empty log message ***
#
#Revision 1.1  2008/07/16 20:23:07  sinnwell
#Initial revision
#
pedigree.shrink <- function(ped, avail, affected=NULL, seed=NULL, maxBits = 16){
  if(class(ped) != "pedigree")
    stop("Must be a pegigree object.\n")

  ## set the seed for random selections
  if(is.null(seed))
```

```
  {
    seed <- sample(2^20, size=1)
  }
set.seed(seed)

if(any(is.na(avail)))
  stop("NA values not allowed in avail vector.")

if(is.null(affected))
  affected = if(is.matrix(ped$affected)) ped$affected[,1] else ped$affected

ped$affected = affected


idTrimmed <- numeric()
idList <- list()
nOriginal <- length(ped$id)

bitSizeOriginal <- bitSize(ped)$bitSize

## first find unavailable subjects to remove anyone who is not
## available and does not have an available descendant

idTrimUnavail <- findUnavailable(ped, avail)


if(length(idTrimUnavail)) {

  pedTrimmed <- pedigree.trim(idTrimUnavail, ped)
  avail <- avail[match(pedTrimmed$id, ped$id)]
  idTrimmed <- c(idTrimmed, idTrimUnavail)
  idList$unavail <- paste(idTrimUnavail, collapse=' ')

} else {
  ## no trimming, reset to original ped
  pedTrimmed <- ped
}


## Next trim any available terminal subjects with unknown phenotype
## but only if both parents are available

## added nNew>0 check because no need to trim anymore if empty ped

nChange <- 1
idList$noninform = NULL
```

```
nNew <- length(pedTrimmed$id)

while(nChange > 0 & nNew > 0){
  nOld <- length(pedTrimmed$id)

  ## findAvailNonInform finds non-informative, but after suggesting
  ## their removal, checks for more unavailable subjects before returning
  idTrimNonInform <- findAvailNonInform(pedTrimmed, avail)

  if(length(idTrimNonInform)) {
      pedNew <- pedigree.trim(idTrimNonInform, pedTrimmed)
      avail <- avail[match(pedNew$id, pedTrimmed$id)]
      idTrimmed <- c(idTrimmed, idTrimNonInform)
      idList$noninform = paste(c(idList$noninform,
             idTrimNonInform), collapse=' ')
      pedTrimmed <- pedNew

  }
  nNew <- length(pedTrimmed$id)
  nChange <- nOld - nNew

}

##  Determine number of subjects & bitSize after initial trimming
nIntermed <- length(pedTrimmed$id)

bitSize <- bitSize(pedTrimmed)$bitSize

## Now sequentially shrink to fit bitSize <= maxBits

bitVec <- c(bitSizeOriginal,bitSize)

isTrimmed <- TRUE
idList$affect=NULL

while(isTrimmed & (bitSize > maxBits))
  {

    ## First, try trimming by unknown status
    save <- findAvailAffected(pedTrimmed, avail, affstatus=NA)
    isTrimmed <- save$isTrimmed

    ## Second, try trimming by unaffected status if no unknowns to trim
    if(!isTrimmed)
      {
        save <- findAvailAffected(pedTrimmed, avail, affstatus=0)
```

```
          isTrimmed <- save$isTrimmed

        }


      ## Third, try trimming by affected status if no unknowns & no unaffecteds
      ## to trim
      if(!isTrimmed) {
        save <- findAvailAffected(pedTrimmed, avail, affstatus=1)
        isTrimmed <- save$isTrimmed
      }

      if(isTrimmed)  {
        pedTrimmed <- save$ped
        avail <- save$newAvail
        bitSize <- save$bitSize
        bitVec <- c(bitVec, bitSize)
        idTrimmed <- c(idTrimmed, save$idTrimmed)
        idList$affect = paste(c(idList$affect, save$idTrimmed),
          collapse=' ')
      }


    } # end while (isTrimmed) & (bitSize > maxBits)


  nFinal <- length(pedTrimmed$id)

  obj <- list(pedObj = pedTrimmed,
              idTrimmed = idTrimmed,
              idList = idList,
              bitSize = bitVec,
              avail=avail,
              pedSizeOriginal = nOriginal,
              pedSizeIntermed = nIntermed,
              pedSizeFinal  = nFinal,
              seed = seed)


  oldClass(obj) <- "pedigree.shrink"

  return(obj)
}
```

8.1 Sub-Functions

These next functions were written to support pedigree.shrink. In making the new kinship2 package to include pedigree.shrink, Jason Sinnwell decided to add functionality to removed subjects from a pedigree object given their id. Then within pedigree.shrink, any removal of subjects consists of two steps, identifying who to remove by their ids. Then removing them with a new pedigree.trim function.

The problem with pedigree.trim is that if the removal of any subject causes a marriage to be split and have parentless children, it will cause a problem.

Therefore, when using functions like findAvalNonInform and findAvalAffected for persons to remove, follow them up with a call findUnavailable, after setting the removal candidate's availability to FALSE, so clear up any removals.

This last step was re-written by Jason Sinnwell on 6/1/2011, and his test cases seemed to test against the results before the re-write. He expects there to be bugs to be discovered down the road.

What was previously pedTrim is now split into two functions, pedigree.trim and findUnavail.

pedigree.trim : remove subjects from pedigree object given their id. Update for version 1.2.8 (9/27/11) Allow creation of an empty pedigree if all IDs are removed. This allows bitSize and pedigree.shrink to still complete with an empty pedigree.

findUnavail: identify subjects are not available and who do not have an available descendant. Do this iteratively by successively removing unavailable terminal nodes. Written by Steve Iturria, PhD, modified by Dan Schaid.

⟨*pedigree.trim*⟩≡

```
pedigree.trim <- function(removeID, ped){
  ## trim subjects from a pedigree who match the removeID
  ## trim relation matrix as well

  if(class(ped) != "pedigree")
    stop("Must be a pegigree object.\n")


  keep <- is.na(match(ped$id, removeID))
  keep.relate <- is.na(match(ped$id[ped$relation[,1]], removeID)) &
                 is.na(match(ped$id[ped$relation[,2]], removeID))

  nOrig <- length(ped$id)
  nNew <- sum(keep)

  if(nNew > 0) {

    newAffected <- if(is.null(ped$affected)) newAffected <- rep(0, nOrig)

    if(is.matrix(ped$affected)) {
      newAffected <- ped$affected[keep,]
    } else {
```

```
      newAffected <- ped$affected[keep]
    }

    id.new <- ped$id[keep]


    ## step1: update the father and mother indices
    fid.new <- mid.new <- rep(NA, length(id.new))
    fid.new[ped$findex[keep]>0] <- ped$id[ped$findex[keep]]
    mid.new[ped$mindex[keep]>0] <- ped$id[ped$mindex[keep]]

    ## step2: any subject that is to be removed, remove them from being parents
    fid.new[fid.new %in% removeID] <- NA
    mid.new[mid.new %in% removeID] <- NA

    ## make new pedigree object with only essential items
    newPed <- pedigree(id=id.new,
                       dadid=fid.new,
                       momid=mid.new,
                       missid=ped$missid,
                       sex=as.numeric(ped$sex[keep]))

    ## trim non-required objects from ped
    if(!is.null(ped$affected)) newPed$affected <- newAffected
    if(!is.null(ped$status)) newPed$status <-  ped$status[keep]

    if(!is.null(ped$famid)) newPed$famid <- ped$famid[keep]
    if(!is.null(ped$relation))
      newPed$relation <- ped$relation[keep.relate,,drop=FALSE]

  } else {
    ## empty pedigree
    newPed <- list(id=NULL, dadid=NULL, momid=NULL, sex=NULL)
    class(newPed) <- "pedigree"
  }
  return(newPed)
}
```

Place the two exclude functions within the same file as findUnavailable because that is the only place they are used. Pretty self-documenting.

⟨*findUnavailable*⟩≡

```
#$Log: pedTrim.q,v $
```

```
#Revision 1.4  2009/11/19 15:00:31  sinnwell
#*** empty log message ***
#
#Revision 1.3  2009/11/19 14:57:05  sinnwell
#*** empty log message ***
#
#Revision 1.2  2009/11/17 23:11:09  sinnwell
#change for ped object
#
#Revision 1.1  2008/07/16 20:23:29  sinnwell
#Initial revision
#


findUnavailable <-function(ped, avail) {

  ## find id within pedigree anyone who is not available and
  ## does not have an available descendant

  ## avail = TRUE/1 if available, FALSE/0 if not

  ## will do this iteratively by successively removing unavailable
  ## terminal nodes
  ## Steve Iturria, PhD, modified by Dan Schaid

  cont <- TRUE                  # flag for whether to keep iterating

  is.terminal <- (is.parent(ped$id, ped$findex, ped$mindex) == FALSE)

  pedData <- data.frame(id=ped$id, father=ped$findex, mother=ped$mindex,
                        sex=ped$sex, avail, is.terminal)
  iter <- 1

  while(cont)  {
    ##print(paste("Working on iter", iter))

    num.found <- 0
    idx.to.remove <- NULL

    for(i in 1:nrow(pedData))
      {

        if(pedData$is.terminal[i])
          {
            if( pedData$avail[i] == FALSE )   # if not genotyped
              {
```

```
            idx.to.remove <- c(idx.to.remove, i)
            num.found <- num.found + 1

            ## print(paste("  removing", num.found, "of", nrow(pedData)))
          }
        }

    }

  if(num.found > 0) {

    pedData <- pedData[-idx.to.remove, ]
    ## re-index parents, which varies depending on if the removed indx is
    ## prior to parent index
    for(k in 1:nrow(pedData)){
      if(pedData$father[k] > 0) {
        pedData$father[k] <- pedData$father[k] -
            sum(idx.to.remove < pedData$father[k])
      }
      if(pedData$mother[k]+0) {
        pedData$mother[k] <- pedData$mother[k] -
            sum(idx.to.remove < pedData$mother[k])
      }
    }
    pedData$is.terminal <-
      (is.parent(pedData$id, pedData$father, pedData$mother) == FALSE)

  }
  else {
    cont <- FALSE
  }
  iter <- iter + 1

}

## A few more clean up steps

## remove unavailable founders
tmpPed <- excludeUnavailFounders(pedData$id,
                     pedData$father, pedData$mother, pedData$avail)

##
tmpPed <- excludeStrayMarryin(tmpPed$id, tmpPed$father, tmpPed$mother)


id.remove <- ped$id[is.na(match(ped$id, tmpPed$id))]
```

```
  return(id.remove)

}


excludeStrayMarryin <- function(id, father, mother){
  # get rid of founders who are not parents (stray available marryins
  # who are isolated after trimming their unavailable offspring)

  trio <- data.frame(id=id, father=father, mother=mother)
  parent <- is.parent(id, father, mother)
  founder <- is.founder(father, mother)

  exclude <- !parent & founder
  trio <- trio[!exclude,,drop=FALSE]
  return(trio)

}

excludeUnavailFounders <- function(id, father, mother, avail)
  {
    nOriginal <- length(id)
    idOriginal <- id
    zed <- father!=0 & mother !=0
    marriage <- paste(id[father[zed]], id[mother[zed]], sep="-" )

    sibship <- tapply(marriage, marriage, length)
    nm <- names(sibship)

    splitPos <- regexpr("-",nm)
    dad <- substring(nm, 1, splitPos-1)
    mom <- substring(nm, splitPos+1,  nchar(nm))

    ##  Want to look at parents with only one child.
    ##  Look for parents with > 1 marriage.  If any
    ##  marriage has > 1 child then skip this mom/dad pair.

    nmarr.dad <- table(dad)
    nmarr.mom <- table(mom)
    skip <- NULL

    if(any(nmarr.dad > 1)) {
      ## Dads in >1 marriage
      ckdad <- which(as.logical(match(dad,
                      names(nmarr.dad)[which(nmarr.dad > 1)],nomatch=FALSE)))
```

```
  skip <- unique(c(skip, ckdad))
}

if(any(nmarr.mom > 1)) {
  ## Moms in >1 marriage
  ckmom <- which(as.logical(match(mom,
                 names(nmarr.mom)[which(nmarr.mom > 1)],nomatch=FALSE)))
  skip <- unique(c(skip, ckmom))
}

if(length(skip) > 0) {
  dad <- dad[-skip]
  mom <- mom[-skip]
  zed <- (sibship[-skip]==1)
} else {
  zed <- (sibship==1)
}


n <- sum(zed)
idTrimmed <- NULL
if(n>0)
  {

    # dad and mom are the parents of sibships of size 1
    dad <- dad[zed]
    mom <- mom[zed]
    for(i in 1:n){
      ## check if mom and dad are founders (where their parents = 0)
      dad.founder <- (father[id==dad[i]] == 0) & (mother[id==dad[i]] == 0)
      mom.founder <- (father[id==mom[i]] == 0) & (mother[id==mom[i]] == 0)
      both.founder <- dad.founder & mom.founder

      ## check if mom and dad have avail
      dad.avail <- avail[id==dad[i]]
      mom.avail <- avail[id==mom[i]]

      ## define not.avail = T if both mom & dad not avail
      not.avail <- (dad.avail==FALSE & mom.avail==FALSE)

      if(both.founder & not.avail)   {
          ## remove mom and dad from ped, and zero-out parent
          ## ids of their child

        child <- which(father==which(id==dad[i]))
        father[child] <- 0
```

```
            mother[child] <- 0

            idTrimmed <- c(idTrimmed, dad[i], mom[i])

            excludeParents <- (id!=dad[i]) & (id!=mom[i])
            id <- id[excludeParents]
            father <- father[excludeParents]
            mother <- mother[excludeParents]

            ## re-index father and mother, assume len(excludeParents)==2
            father <- father - 1*(father > which(!excludeParents)[1]) -
              1*(father > which(!excludeParents)[2])

            mother <- mother - 1*(mother > which(!excludeParents)[1]) -
              1*(mother > which(!excludeParents)[2])

            avail <- avail[excludeParents]
          }
        }
      }

    nFinal <- length(id)
    nTrimmed = nOriginal - nFinal


    return(list(nTrimmed = nTrimmed, idTrimmed=idTrimmed,
                id=id, father=father, mother=mother))
  }
```

Function to calculate pedigree bit size, which is 2 * n.NonFounder - n.Founder. It is an indicator for how much resources the pedigree will require to be processed by linkage algorithms to calculate the likelihood of the observed genotypes given the pedigree structure.

The Lander-Green handles smaller pedigrees and many markers The Elston-Stewart handles larger pedigrees and fewer markers.

⟨*bitSize*⟩≡

```
  ## renamed from pedBits, part of pedigree.shrink functions

  bitSize <- function(ped) {
    ## calculate bit size of a pedigree

    if(class(ped) != "pedigree")
      stop("Must be a pegigree object.\n")
```

```
  father = ped$findex
  mother = ped$mindex
  id = ped$id

  founder <- father==0 & mother==0
  pedSize <- length(father)
  nFounder <- sum(founder)
  nNonFounder <- pedSize - nFounder
  bitSize <- 2*nNonFounder - nFounder
  return(list(bitSize=bitSize,
              nFounder = nFounder,
              nNonFounder = nNonFounder))
}
```

Two functions to identify subjects to remove by other indicators0 than availability.

findAvailNonInform: id subjects to remove who are available, but not informative. This function was formerly trimAvailNonInform().

findAvailAffected: id subjects to remove who were not removed by findUnavailable(), but who would be best to remove given their affected status. Try trimming one subject by with affected matching affstatus. If there are ties of multiple subjects that reduce bit size equally, randomly choose one of them. This function was formerly named pedTrimOneSubj().

⟨*findAvailNonInform*⟩≡

```
findAvailNonInform <- function(ped, avail){

  ## trim persons who are available but not informative b/c not parent
  ## by setting their availability to FALSE, then call findUnavailable()

  pedData <- data.frame(id=ped$id, father=ped$findex,
                        mother=ped$mindex, avail=avail)

  checkParent <- is.parent(pedData$id, pedData$father, pedData$mother)

  for(i in 1:nrow(pedData)){

    if(checkParent[i]==FALSE & avail[i]==TRUE &
       all(ped$affected[i]==0, na.rm=TRUE)) {

      ## could use ped$affected[i,] if keep matrix

        fa <- pedData$id[pedData$father[i]]
        mo <- pedData$id[pedData$mother[i]]
        if(avail[pedData$id==fa] & avail[pedData$id==mo])
          {
```

```
            pedData$avail[i] <- FALSE
          }
      }
  }

  idTrim <- findUnavailable(ped, pedData$avail)
  return(idTrim)
}
```

⟨*findAvailAffected*⟩≡

```
findAvailAffected <- function(ped, avail, affstatus)
  ## Try trimming one subject by affection status indicator
  ## If ties for bits removed, randomly select one of the subjects

  {

    notParent <- !is.parent(ped$id, ped$findex, ped$mindex)

    if(is.na(affstatus)) {
      possiblyTrim <- ped$id[notParent & avail & is.na(ped$affected)]
    } else {
      possiblyTrim <- ped$id[notParent & avail & ped$affected==affstatus]
    }
    nTrim <- length(possiblyTrim)

    if(nTrim == 0)
      {
        return(list(ped=ped,
                    idTrimmed = NA,
                    isTrimmed = FALSE,
                    bitSize = bitSize(ped)$bitSize))
      }

    trimDat <- NULL

  for(idTrim in possiblyTrim) {


      avail.try <- avail
      avail.try[ped$id==idTrim] <- FALSE
      id.rm <- findUnavailable(ped, avail.try)
      newPed <- pedigree.trim(id.rm, ped)
      trimDat <- rbind(trimDat,
                  c(id=idTrim, bitSize=bitSize(newPed)$bitSize))
```

```
    }

    bits <- trimDat[,2]

    # trim by subject with min bits. This trims fewer subject than
    # using max(bits).

    idTrim <- trimDat[bits==min(bits), 1]

    ## break ties by random choice
    if(length(idTrim) > 1)
      {
        rord <- order(runif(length(idTrim)))
        idTrim <- idTrim[rord][1]
      }


    avail[ped$id==idTrim] <- FALSE
    id.rm <- findUnavailable(ped, avail)
    newPed <- pedigree.trim(id.rm, ped)
    pedSize <- bitSize(newPed)$bitSize
    avail <- avail[!(ped$id %in% id.rm)]

    return(list(ped=newPed,
                newAvail = avail,
                idTrimmed = idTrim,
                isTrimmed = TRUE,
                bitSize = pedSize))
  }
```

Group other functions used in the above main functions together as pedigree.shrink.minor.R These functions get indicator vectors of who is a parent, founder, or disconnected

⟨*pedigree.shrink.minor*⟩≡

```
#$Log: pedigree.shrink.minor.q,v $
#Revision 1.5  2009/11/19 18:10:26  sinnwell
#F to FALSE
#
#Revision 1.4  2009/11/19 14:57:13  sinnwell
#*** empty log message ***
#
#Revision 1.3  2009/11/17 23:11:41  sinnwell
#*** empty log message ***
#
#Revision 1.1  2008/07/16 20:22:55  sinnwell
```

```
#Initial revision
#


is.parent <- function(id, findex, mindex){
  # determine subjects who are parents
  # assume input of father/mother indices, not ids

  father <- mother <- rep(0, length(id))
  father[findex>0] <- id[findex]
  mother[mindex>0] <- id[mindex]

  isFather <- !is.na(match(id, unique(father[father!=0])))
  isMother <- !is.na(match(id, unique(mother[mother!=0])))
  isParent <- isFather |isMother
  return(isParent)
}

is.founder <- function(mother, father){
  check <- (father==0) & (mother==0)
  return(check)
}


is.disconnected <- function(id, findex, mindex)
{

  # check to see if any subjects are disconnected in pedigree by checking for
  # kinship = 0 for all subjects excluding self
  father <- id[findex]
  mother <- id[mindex]
  kinMat <- kinship(id, father, mother)
  diag(kinMat) <- 0
  disconnected <- apply(kinMat==0.0, 1, all)

  return(disconnected)
}
```

Print a pedigree.shrink object. Tell the original bit size and the trimmed bit size.

⟨*print.pedigree.shrink*⟩≡

```
#$Log: print.pedigree.shrink.q,v $
#Revision 1.2  2009/11/19 14:35:01  sinnwell
#add ...
#
#Revision 1.1  2009/11/17 14:39:32  sinnwell
```

```
#Initial revision
#
#Revision 1.1  2008/07/16 20:23:14  sinnwell
#Initial revision
#

print.pedigree.shrink <- function(x, ...){

  printBanner(paste("Shrink of Pedigree ", unique(x$pedObj$ped), sep=""))

  cat("Pedigree Size:\n")

  if(length(x$idTrimmed) > 2)
    {
      n <- c(x$pedSizeOriginal, x$pedSizeIntermed, x$pedSizeFinal)
      b <- c(x$bitSize[1], x$bitSize[2], x$bitSize[length(x$bitSize)])
      row.nms <- c("Original","Only Informative","Trimmed")
    } else {
      n <- c(x$pedSizeOriginal, x$pedSizeIntermed)
      b <- c(x$bitSize[1], x$bitSize[2])
      row.nms <- c("Original","Trimmed")
    }

  df <- data.frame(N.subj = n, Bits = b)
  rownames(df) <- row.nms
  print(df, quote=FALSE)


  if(!is.null(x$idList$unavail))
    cat("\n Unavailable subjects trimmed:\n", x$idList$unavail, "\n")

  if(!is.null(x$idList$noninform))
    cat("\n Non-informative subjects trimmed:\n", x$idList$noninform, "\n")

  if(!is.null(x$idList$affect))
    cat("\n Informative subjects trimmed:\n", x$idList$affect, "\n")


  ##cat("\n Pedigree after trimming:", x$bitSize, "\n")

  invisible()
}
```

⟨*printBanner*⟩≡

```
#$Log: printBanner.q,v $
```

```
#Revision 1.4  2007/01/23 21:00:27  sinnwell
#rm ending newline \n.  Users can space if desired.
#
#Revision 1.3  2005/02/04 20:57:18  sinnwell
#banner.width now based on options()$width
#char.perline based on banner.width
#
#Revision 1.2  2004/06/25 15:56:48  sinnwell
#now compatible with R, changed end when a line is done
#
#Revision 1.1  2004/02/26 21:34:55  sinnwell
#Initial revision
#

printBanner <- function(str, banner.width=options()$width, char.perline=.75*banner.width, border

# char.perline was calculated taking the floor of banner.width/3

  vec <- str
  new<-NULL
  onespace<-FALSE
  for(i in 1:nchar(vec)){
    if (substring(vec,i,i)==' ' && onespace==FALSE){
      onespace<-TRUE
      new<-paste(new,substring(vec,i,i),sep="")}
    else if (substring(vec,i,i)==' ' && onespace==TRUE)
      {onespace<-TRUE}
    else{
      onespace<-FALSE
      new<-paste(new,substring(vec,i,i),sep="")}
  }

  where.blank<-NULL
  indx <- 1

  for(i in 1:nchar(new)){
    if((substring(new,i,i)==' ')){
      where.blank[indx]<-i
      indx <- indx+1
    }
  }


# Determine the position in the where.blank vector to insert the Nth character position of "new"
  j<-length(where.blank)+1
```

```
# Add the Nth character position of the "new" string to the where.blank vector.
  where.blank[j]<-nchar(new)

  begin<-1
  end<-max(where.blank[where.blank<=char.perline])

# If end.ok equals NA then the char.perline is less than the position of the 1st blank.
  end.ok <- is.na(end)

# Calculate a new char.perline.
  if (end.ok==TRUE){
    char.perline <- floor(banner.width/2)
    end<-max(where.blank[where.blank<=char.perline])
  }

  cat(paste(rep(border, banner.width), collapse = ""),"\n")

  repeat {
    titleline<-substring(new,begin,end)
    n <- nchar(titleline)
    if(n < banner.width)
      {
        n.remain <- banner.width - n
        n.left <- floor(n.remain/2)
        n.right <- n.remain - n.left
        for(i in 1:n.left) titleline <- paste(" ",titleline,sep="")
        for(i in 1:n.right) titleline <- paste(titleline," ",sep="")
        n <- nchar(titleline)
      }
     cat(titleline,"\n")
    begin<-end+1
    end.old <- end
   # Next line has a problem when used in R.  Use print.banner.R until fixed.
   # Does max with an NA argument
    tmp <- where.blank[(end.old<where.blank) & (where.blank<=end.old+char.perline+1)]
    if(length(tmp)) end <- max(tmp)
    else break

#   end<-max(where.blank[(end.old<where.blank)&(where.blank<=end.old+char.perline+1)])
#   end.ok <- is.na(end)
#   if (end.ok==TRUE)
#      break
  }

  cat(paste(rep(border, banner.width), collapse = ""), "\n")
  invisible()
```

```
}
```

Plot a pedigree.shrink object, which calls the plot.pedigree function on the trimmed pedigree object.

⟨*plot.pedigree.shrink*⟩≡

```
#$Log: plot.pedigree.shrink.q,v $
#Revision 1.4  2010/09/03 21:12:16  sinnwell
#use shrunk "avail" vector for the colored labels
#
#Revision 1.3  2009/11/19 14:57:18  sinnwell
#*** empty log message ***
#
#Revision 1.2  2009/11/17 23:09:51  sinnwell
#updated for ped object
#
#Revision 1.1  2008/07/16 20:23:38  sinnwell
#Initial revision
#

plot.pedigree.shrink <- function(x, bigped=FALSE, title="",
                                 xlegend="topright", ...){

  ##  Plot pedigrees, coloring subjects according
  ##   to availability, shaded by affected status used in shrink

  if(bigped==FALSE){
    tmp <- plot(x$pedObj, col=x$avail+1)
  } else {
    tmp <- plot.pedigree(x$pedObj, align=FALSE, packed=FALSE,
                         col=x$avail+1, cex=0.5,symbolsize=0.5)
  }

  legend(x=xlegend,
         legend=c("Available","UnAvailable"),
         pch=c(1,1), col=c(2,1),bty="n")


  title(paste(title, "\nbits = ", x$bitSize[length(x$bitSize)]))
}
```

/sectionPedigree Unrelated

Purpose: Determine set of maximum number of unrelated available subjects from a pedigree PI: Dan Schaid Author(s): Dan Schaid, Shannon McDonnell Dates: Created: 10/19/2007, Moved to kinship2: 6/2011

In many pedigrees there are multiple sets of subjects that could be of the size of the maximal set of unrelated subjects in a pedigree. The set could contain a married-in uncle and any of a set of siblings from his sister-in-law's family. Therefore, the maximal sets include the uncle and any of the sibship of his wife's sister.

⟨*pedigree.unrelated*⟩≡

```
#$Log: pedigree.unrelated.q,v $
#Revision 1.2  2010/02/11 22:36:48  sinnwell
#require kinship to be loaded before use
#
#Revision 1.1  2009/11/10 19:21:52  sinnwell
#Initial revision
#
#Revision 1.1  2009/11/03 16:42:27  sinnwell
#Initial revision
#
## Authors: Dan Schaid, Shannon McDonnell
## Updated by Jason Sinnwell

pedigree.unrelated <- function(ped, avail) {

  # Requires: kinship function

  # Given vectors id, father, and mother for a pedigree structure,
  # and avail = vector of T/F or 1/0 for whether each subject
  # (corresponding to id vector) is available (e.g.,
  # has DNA available), determine set of maximum number
  # of unrelated available subjects from a pedigree.

  # This is a greedy algorithm that uses the kinship
  # matrix, sequentially removing rows/cols that
  # are non-zero for subjects that have the most
  # number of zero kinship coefficients (greedy
  # by choosing a row of kinship matrix that has
  # the most number of zeros, and then remove any
  # cols and their corresponding rows that are non-zero.
  # To account for ties of the count of zeros for rows,
  # a random choice is made. Hence, running this function
  # multiple times can return different sets of unrelated
  # subjects.
```

```
  id <- ped$id
  avail <- as.integer(avail)


  kin <- kinship(ped)

  ord <- order(id)
  id <- id[ord]
  avail <- as.logical(avail[ord])
  kin <- kin[ord,][,ord]

  rord <- order(runif(nrow(kin)))

  id <- id[rord]
  avail <- avail[rord]
  kin <- kin[rord,][,rord]

  id.avail <- id[avail]
  kin.avail <- kin[avail,,drop=FALSE][,avail,drop=FALSE]

  diag(kin.avail) <- 0

  while(any(kin.avail > 0))
    {
      nr <- nrow(kin.avail)
      indx <- 1:nrow(kin.avail)
      zero.count <- apply(kin.avail==0, 1, sum)

      mx <- max(zero.count[zero.count < nr])
      mx.zero <- indx[zero.count == mx][1]

      exclude <- indx[kin.avail[, mx.zero] > 0]

      kin.avail <- kin.avail[- exclude, , drop=FALSE][, -exclude, drop=FALSE]

    }

  choice <- sort(dimnames(kin.avail)[[1]])

  return(choice)
}
```

9 Checks

Last are various helper routines and data checks.

9.1 kindepth

One helper function used throughout computes the depth of each subject in the pedigree. For each subject this is defined as the maximal number of generations of ancestors: how far to the farthest founder. This can be called with a pedigree object, or with the full argument list. In the former case we can simply skip a step.

⟨*kindepth*⟩≡

```
kindepth <- function(id, dad.id, mom.id, align=FALSE) {
    if (class(id)=='pedigree' || class(id)=='pedigreeList') {
        didx <- id$findex
        midx <- id$mindex
        n <- length(didx)
        }
    else {
        n <- length(id)
        if (missing(dad.id) || length(dad.id) !=n)
            stop("Invalid father id")
        if (missing(mom.id) || length(mom.id) !=n)
            stop("Invalid mother id")
        midx <- match(mom.id, id, nomatch=0) # row number of my mom
        didx <- match(dad.id, id, nomatch=0) # row number of my dad
        }
    if (n==1) return (0)  # special case of a single subject
    parents <- which(midx==0 & didx==0)  #founders

    depth <- rep(0,n)
    # At each iteration below, all children of the current "parents" are
    #    labeled with depth 'i', and become the parents of the next iteration
    for (i in 1:n) {
        child  <- match(midx, parents, nomatch=0) +
                  match(didx, parents, nomatch=0)

        if (all(child==0)) break
        if (i==n)
            stop("Impossible pedegree: someone is their own ancestor")

        parents <- which(child>0) #next generation of parents
        depth[parents] <- i
        }
    if (!align) return(depth)
```

The align argument is used only by the plotting routines. It makes the plotted result prettier in the following (fairly common) case. Assume that subjects A and B marry, we have some ancestry information for both, and that A's ancestors go back 3 generations, B's for only two. If we add +1 to the depth of B and all her ancestors, then A and B will be the same depth, and will plot on the same line. A marry-in to the pedigree with no ancestry is also handled nicely by the algorithm. However, if we have an inbred pedigree, there may not be a simple fix of this sort.

The algorithm is

1. Find any mother-father pairs that are mismatched in depth. We think that aligning the top of a pedigree is more important than aligning at the bottom, so choose a mismatch pair of minimal depth.

2. The children's depth is max(father, mother) +1. Call the parent closest to the children "good" and the other "bad".

3. Chase up the good side, and get a list of all subjects connected to "good", including in-laws (spouse connections) and sibs that are at this level or above. Call this agood (ancestors of good). We do not follow any connections at a depth lower than the marriage in question, to get the highest marriages right. For the bad side, just get ancestors.

4. Avoid pedigree loops! If the agood list contains anyone in abad, then don't try to fix the alignment, otherwise: Push abad down, then run the pushdown algorithm to repair any descendents — you may have pulled down a grandparent but not the sibs of that grandparent.

It may be possible to do better alignment when the pedigree has loops, but it is definitely beyond this program's abilities. This could be an addition to authint one day. One particular case that we've seen was a pair of brothers that married a pair of sisters. Pulling one brother down fixes the other at the same time. The code below, however, says "loop! stay away!".

⟨*kindepth*⟩+≡

```
chaseup <- function(x, midx, didx) {
    new <- c(midx[x], didx[x])  # mother and father
    new <- new[new>0]
    while (length(new) >1) {
        x <- unique(c(x, new))
        new <- c(midx[new], didx[new])
        new <- new[new>0]
        }
    x
    }

dads <- didx[midx>0 & didx>0]   # the father side of all spouse pairs
moms <- midx[midx>0 & didx>0]
# Get rid of duplicate pairs
dups <- duplicated(dads + moms*n)
```

```
if (any(dups)) {
    dads <- dads[!dups]
    moms <- moms[!dups]
    }
npair<- length(dads)
done <- rep(FALSE, npair)  #couples that are taken care of
while (TRUE) {
    pairs.to.fix <- (1:npair)[(depth[dads] != depth[moms]) & !done]
    if (length(pairs.to.fix) ==0) break
    temp <- pmax(depth[dads], depth[moms])[pairs.to.fix]
    who <- min(pairs.to.fix[temp==min(temp)])  # the chosen couple

    good <- moms[who]; bad <- dads[who]
    if (depth[dads[who]] > depth[moms[who]]) {
        good <- dads[who]; bad <- moms[who]
        }
    abad  <- chaseup(bad,  midx, didx)
    if (length(abad) ==1 && sum(c(dads,moms)==bad)==1) {
        # simple case, a solitary marry-in
        depth[bad] <- depth[good]
        }
    else {
        agood <- chaseup(good, midx, didx)  #ancestors of the "good" side
        # For spouse chasing, I need to exclude the given pair
        tdad <- dads[-who]
        tmom <- moms[-who]
        while (1) {
            # spouses of any on agood list
            spouse <- c(tmom[!is.na(match(tdad, agood))],
                        tdad[!is.na(match(tmom, agood))])
            temp <- unique(c(agood, spouse))
            temp <- unique(chaseup(temp, midx, didx)) #parents
            kids <- (!is.na(match(midx, temp)) | !is.na(match(didx, temp)))
            temp <- unique(c(temp, (1:n)[kids & depth <= depth[good]]))
            if (length(temp) == length(agood)) break
            else agood <- temp
            }

        if (all(match(abad, agood, nomatch=0) ==0)) {
            # shift it down
            depth[abad] <- depth[abad] + (depth[good] - depth[bad])
            #
            # Siblings may have had children: make sure all kids are
            #   below their parents.  It's easiest to run through the
            #   whole tree
            for (i in 0:n) {
```

```
                parents <- which(depth==i)
                child <- match(midx, parents, nomatch=0) +
                         match(didx, parents, nomatch=0)
                if (all(child==0)) break
                depth[child>0] <- pmax(i+1, depth[child>0])
                }
            }
        }
    done[who] <- TRUE
    }
if (all(depth>0)) stop("You found a bug in kindepth's alignment code!")
depth
}
```

9.2 familycheck

The familycheck routine checks out a family id, by trying to construct its own and comparing the results. The input argument "newfam" is optional: if you've already created this vector for other reasons, then putting the arg in saves time.

If there are any joins, then an attribute "join" is attached. It will be a matrix with famid as row labels, new-family-id as the columns, and the number of subjects as entries.

⟨*familycheck*⟩≡

```
# This routine checks out a family id, by trying to construct its own
#  and comparing the results
#
# The input argument "newfam" is optional: if you've already created this
#   vector for other reasons, then putting the arg in saves time.
#
# Output is a dataframe with columns:
#   famid: the family id, as entered into the data set
#   n    : number of subjects in the family
#   unrelated: number of them that appear to be unrelated to anyone else
#          in the entire pedigree set.  This is usually marry-ins with no
#          children (in the pedigree), and if so are not a problem.
#   split : number of unique "new" family ids.
#            if this is 0, it means that no one in this "family" is related to
#                   anyone else (not good)
#            1 = everythings is fine
#            2+= the family appears to be a set of disjoint trees.  Are you
#                 missing some of the people?
#   join : number of other families that had a unique famid, but are actually
#            joined to this one.  0 is the hope.
#
#  If there are any joins, then an attribute "join" is attached.  It will be
#   a matrix with famid as row labels, new-family-id as the columns, and
```

```
#   the number of subjects as entries.
#
familycheck <- function(famid, id, father.id, mother.id, newfam) {
    if (is.numeric(famid) && any(is.na(famid)))
        stop ("Family id of missing not allowed")
    nfam <- length(unique(famid))

    if (missing(newfam)) newfam <- makefamid(id, father.id, mother.id)
    else if (length(newfam) != length(famid))
        stop("Invalid length for newfam")

    xtab <- table(famid, newfam)
    if (any(newfam==0)) {
        unrelated <- xtab[,1]
        xtab <- xtab[,-1, drop=FALSE]
        ## bug fix suggested by Amanda Blackford 6/2011
      }
    else unrelated <-  rep(0, nfam)

    splits <- apply(xtab>0, 1, sum)
    joins  <- apply(xtab>0, 2, sum)

    temp <- apply((xtab>0) * outer(rep(1,nfam), joins-1), 1, sum)

    out <- data.frame(famid = dimnames(xtab)[[1]],
                      n = as.vector(table(famid)),
                      unrelated = as.vector(unrelated),
                      split = as.vector(splits),
                      join = temp,
                      row.names=1:nfam)
    if (any(joins >1)) {
      tab1 <- xtab[temp>0,]  #families with multiple outcomes
      tab1 <- tab1[,apply(tab1>0,2,sum) >0] #only keep non-zero columns
      dimnames(tab1) <- list(dimnames(tab1)[[1]], NULL)
      attr(out, 'join') <- tab1
    }

    out
  }
```

9.3 check.hint

This routine tries to remove inconsistencies in spousal hints. These and arise in autohint with complex pedigrees. One can have ABA (subject A is on both the left and the right of B), cycles, etc. Actually, these used to arise in autohint, I don't know if it's so after the recent rewrite. Users can introduce problems as well if they modify the hints.

⟨*check.hint*⟩≡

```
check.hint <- function(hints, sex) {
    if (is.null(hints$order)) stop("Missing order component")
    if (!is.numeric(hints$order)) stop("Invalid order component")
    n <- length(sex)
    if (length(hints$order) != n) stop("Wrong length for order component")

    spouse <- hints$spouse
    if (is.null(spouse)) hints
    else {
        lspouse <- spouse[,1]
        rspouse <- spouse[,2]
        if (any(lspouse <1 | lspouse >n | rspouse <1 | rspouse > n))
            stop("Invalid spouse value")

        temp1 <- (sex[lspouse]== 'female' & sex[rspouse]=='male')
        temp2 <- (sex[rspouse]== 'female' & sex[lspouse]=='male')
        if (!all(temp1 | temp2))
            stop("A marriage is not male/female")

        hash <- n*pmax(lspouse, rspouse) + pmin(lspouse, rspouse)
        #Turn off this check for now - is set off if someone is married to two siblings
        #if (any(duplicated(hash))) stop("Duplicate marriage")

        # Break any loops: A left of B, B left of C, C left of A.
        #  Not yet done
      }
    hints
  }
```