Chapter B14. Statistical Description of Data

```
SUBROUTINE moment(data,ave,adev,sdev,var,skew,curt)
USE nrtype; USE nrutil, ONLY : nrerror
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: ave,adev,sdev,var,skew,curt
REAL(SP), DIMENSION(:), INTENT(IN) :: data
```

Given an array of data, this routine returns its mean ave, average deviation adev, standard deviation sdev, variance var, skewness skew, and kurtosis curt.

```
INTEGER(I4B) :: n
REAL(SP) :: ep
REAL(SP), DIMENSION(size(data)) :: p,s
n=size(data)
if (n <= 1) call nrerror('moment: n must be at least 2')
ave=sum(data(:))/n                  First pass to get the mean.
s(:)=data(:)-ave                    Second pass to get the first (absolute), second, third, and
ep=sum(s(:))                            fourth moments of the deviation from the mean.
adev=sum(abs(s(:)))/n
p(:)=s(:)*s(:)
var=sum(p(:))
p(:)=p(:)*s(:)
skew=sum(p(:))
p(:)=p(:)*s(:)
curt=sum(p(:))
var=(var-ep**2/n)/(n-1)             Corrected two-pass formula.
sdev=sqrt(var)
if (var /= 0.0) then
    skew=skew/(n*sdev**3)
    curt=curt/(n*var**2)-3.0_sp
else
    call nrerror('moment: no skew or kurtosis when zero variance')
end if
END SUBROUTINE moment
```

⋆ ⋆ ⋆

```
SUBROUTINE ttest(data1,data2,t,prob)
USE nrtype
USE nr, ONLY : avevar,betai
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob
```

Given the arrays data1 and data2, which need not have the same length, this routine returns Student's t as t, and its significance as prob, small values of prob indicating that

the arrays have significantly different means. The data arrays are assumed to be drawn from populations with the same true variance.

```
INTEGER(I4B) :: n1,n2
REAL(SP) :: ave1,ave2,df,var,var1,var2
n1=size(data1)
n2=size(data2)
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
df=n1+n2-2                                          Degrees of freedom.
var=((n1-1)*var1+(n2-1)*var2)/df                    Pooled variance.
t=(ave1-ave2)/sqrt(var*(1.0_sp/n1+1.0_sp/n2))
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))           See equation (6.4.9).
END SUBROUTINE ttest
```

★ ★ ★

```
SUBROUTINE avevar(data,ave,var)
USE nrtype
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data
REAL(SP), INTENT(OUT) :: ave,var
```

Given array data, returns its mean as ave and its variance as var.

```
INTEGER(I4B) :: n
REAL(SP), DIMENSION(size(data)) :: s
n=size(data)
ave=sum(data(:))/n
s(:)=data(:)-ave
var=dot_product(s,s)
var=(var-sum(s)**2/n)/(n-1)          Corrected two-pass formula (14.1.8).
END SUBROUTINE avevar
```

★ ★ ★

```
SUBROUTINE tutest(data1,data2,t,prob)
USE nrtype
USE nr, ONLY : avevar,betai
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob
```

Given the arrays data1 and data2, which need not have the same length, this routine returns Student's t as t, and its significance as prob, small values of prob indicating that the arrays have significantly different means. The data arrays are allowed to be drawn from populations with unequal variances.

```
INTEGER(I4B) :: n1,n2
REAL(SP) :: ave1,ave2,df,var1,var2
n1=size(data1)
n2=size(data2)
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
t=(ave1-ave2)/sqrt(var1/n1+var2/n2)
df=(var1/n1+var2/n2)**2/((var1/n1)**2/(n1-1)+(var2/n2)**2/(n2-1))
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))
END SUBROUTINE tutest
```

★ ★ ★

```
SUBROUTINE tptest(data1,data2,t,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : avevar,betai
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: t,prob
```

Given the paired arrays data1 and data2 of the same length, this routine returns Student's t for paired data as t, and its significance as prob, small values of prob indicating a significant difference of means.

```
INTEGER(I4B) :: n
REAL(SP) :: ave1,ave2,cov,df,sd,var1,var2
n=assert_eq(size(data1),size(data2),'tptest')
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
cov=dot_product(data1(:)-ave1,data2(:)-ave2)
df=n-1
cov=cov/df
sd=sqrt((var1+var2-2.0_sp*cov)/n)
t=(ave1-ave2)/sd
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))
END SUBROUTINE tptest
```

⋆ ⋆ ⋆

```
SUBROUTINE ftest(data1,data2,f,prob)
USE nrtype
USE nr, ONLY : avevar,betai
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: f,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
```

Given the arrays data1 and data2, which need not have the same length, this routine returns the value of f, and its significance as prob. Small values of prob indicate that the two arrays have significantly different variances.

```
INTEGER(I4B) :: n1,n2
REAL(SP) :: ave1,ave2,df1,df2,var1,var2
n1=size(data1)
n2=size(data2)
call avevar(data1,ave1,var1)
call avevar(data2,ave2,var2)
if (var1 > var2) then          Make F the ratio of the larger variance to the smaller one.
    f=var1/var2
    df1=n1-1
    df2=n2-1
else
    f=var2/var1
    df1=n2-1
    df2=n1-1
end if
prob=2.0_sp*betai(0.5_sp*df2,0.5_sp*df1,df2/(df2+df1*f))
if (prob > 1.0) prob=2.0_sp-prob
END SUBROUTINE ftest
```

⋆ ⋆ ⋆

```
SUBROUTINE chsone(bins,ebins,knstrn,df,chsq,prob)
USE nrtype; USE nrutil, ONLY : assert_eq,nrerror
USE nr, ONLY : gammq
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: knstrn
REAL(SP), INTENT(OUT) :: df,chsq,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: bins,ebins
```

Given the same-size arrays bins containing the observed numbers of events, and ebins containing the expected numbers of events, and given the number of constraints knstrn (normally one), this routine returns (trivially) the number of degrees of freedom df, and (nontrivially) the chi-square chsq and the significance prob. A small value of prob indicates a significant difference between the distributions bins and ebins. Note that bins and ebins are both real arrays, although bins will normally contain integer values.

```
INTEGER(I4B) :: ndum
ndum=assert_eq(size(bins),size(ebins),'chsone')
if (any(ebins(:) <= 0.0)) call nrerror('bad expected number in chsone')
df=size(bins)-knstrn
chsq=sum((bins(:)-ebins(:))**2/ebins(:))
prob=gammq(0.5_sp*df,0.5_sp*chsq)          Chi-square probability function. See §6.2.
END SUBROUTINE chsone
```

```
SUBROUTINE chstwo(bins1,bins2,knstrn,df,chsq,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : gammq
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: knstrn
REAL(SP), INTENT(OUT) :: df,chsq,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: bins1,bins2
```

Given the same-size arrays bins1 and bins2, containing two sets of binned data, and given the number of constraints knstrn (normally 1 or 0), this routine returns the number of degrees of freedom df, the chi-square chsq, and the significance prob. A small value of prob indicates a significant difference between the distributions bins1 and bins2. Note that bins1 and bins2 are both real arrays, although they will normally contain integer values.

```
INTEGER(I4B) :: ndum
LOGICAL(LGT), DIMENSION(size(bins1)) :: nzeromask
ndum=assert_eq(size(bins1),size(bins2),'chstwo')
nzeromask = bins1(:) /= 0.0 .or. bins2(:) /= 0.0
chsq=sum((bins1(:)-bins2(:))**2/(bins1(:)+bins2(:)),mask=nzeromask)
df=count(nzeromask)-knstrn                 No data means one less degree of freedom.
prob=gammq(0.5_sp*df,0.5_sp*chsq)          Chi-square probability function. See §6.2.
END SUBROUTINE chstwo
```

f90 nzeromask=...chisq=sum(...mask=nzeromask) We use the optional argument mask in sum to select out the elements to be summed over. In this case, at least one of the elements of bins1 or bins2 is not zero for each term in the sum.

⋆ ⋆ ⋆

```
SUBROUTINE ksone(data,func,d,prob)
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : probks,sort
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: d,prob
REAL(SP), DIMENSION(:), INTENT(INOUT) :: data
INTERFACE
    FUNCTION func(x)
    USE nrtype
    REAL(SP), DIMENSION(:), INTENT(IN) :: x
    REAL(SP), DIMENSION(size(x)) :: func
    END FUNCTION func
END INTERFACE
```

Given an array data, and given a user-supplied function of a single variable func which is a cumulative distribution function ranging from 0 (for smallest values of its argument) to 1 (for largest values of its argument), this routine returns the K–S statistic d, and the significance level prob. Small values of prob show that the cumulative distribution function of data is significantly different from func. The array data is modified by being sorted into ascending order.

```
INTEGER(I4B) :: n
REAL(SP) :: en
REAL(SP), DIMENSION(size(data)) :: fvals
REAL(SP), DIMENSION(size(data)+1) :: temp
call sort(data)                                  If the data are already sorted into ascending order, then this call can be omitted.
n=size(data)
en=n
fvals(:)=func(data(:))
temp=arth(0,1,n+1)/en
d=maxval(max(abs(temp(1:n)-fvals(:)), &          Compute the maximum distance between the data's c.d.f. and the user-supplied function.
    abs(temp(2:n+1)-fvals(:))))
en=sqrt(en)
prob=probks((en+0.12_sp+0.11_sp/en)*d)           Compute significance.
END SUBROUTINE ksone
```

f90 d=maxval(max... Note the difference between max and maxval: max takes two or more arguments and returns the maximum. If the arguments are two arrays, it returns an array each of whose elements is the maximum of the corresponding elements in the two arrays. maxval takes a single array argument and returns its maximum value.

```
SUBROUTINE kstwo(data1,data2,d,prob)
USE nrtype; USE nrutil, ONLY : cumsum
USE nr, ONLY : probks,sort2
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: d,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
```

Given arrays data1 and data2, which can be of different length, this routine returns the K–S statistic d, and the significance level prob for the null hypothesis that the data sets are drawn from the same distribution. Small values of prob show that the cumulative distribution function of data1 is significantly different from that of data2. The arrays data1 and data2 are not modified.

```
INTEGER(I4B) :: n1,n2
REAL(SP) :: en1,en2,en
REAL(SP), DIMENSION(size(data1)+size(data2)) :: dat,org
n1=size(data1)
n2=size(data2)
en1=n1
en2=n2
dat(1:n1)=data1                                  Copy the two data sets into a single array.
dat(n1+1:)=data2
```

```
org(1:n1)=0.0                                   Define an array that contains 0 when the
org(n1+1:)=1.0                                      corresponding element comes from
call sort2(dat,org)                                 data1, 1 from data2.
  Sort the array of 1's and 0's into the order of the merged data sets.
d=maxval(abs(cumsum(org)/en2-cumsum(1.0_sp-org)/en1))
  Now use cumsum to get the c.d.f. corresponding to each set of data.
en=sqrt(en1*en2/(en1+en2))
prob=probks((en+0.12_sp+0.11_sp/en)*d)          Compute significance.
END SUBROUTINE kstwo
```

The problem here is how to compute the cumulative distribution function (c.d.f.) corresponding to each set of data, and then find the corresponding KS statistic, without a serial loop over the data. The trick is to define an array that contains 0 when the corresponding element comes from the first data set and 1 when it's from the second data set. Sort the array of 1's and 0's into the same order as the merged data sets. Now tabulate the partial sums of the array. Every time you encounter a 1, the partial sum increases by 1. So if you normalize the partial sums by dividing by the number of elements in the second data set, you have the c.d.f. of the second data set.

If you subtract the array of 1's and 0's from an array of all 1's, you get an array where 1 corresponds to an element in the first data set, 0 the second data set. So tabulating its partial sums and normalizing gives the c.d.f. of the first data set. As we've seen before, tabulating partial sums can be done with a parallel algorithm (`cumsum` in `nrutil`). The KS statistic is just the maximum absolute difference of the c.d.f.'s, computed in parallel with Fortran 90's `maxval` function.

```
FUNCTION probks(alam)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: alam
REAL(SP) :: probks
REAL(SP), PARAMETER :: EPS1=0.001_sp,EPS2=1.0e-8_sp
INTEGER(I4B), PARAMETER :: NITER=100
   Kolmogorov-Smirnov probability function.
INTEGER(I4B) :: j
REAL(SP) :: a2,fac,term,termbf
a2=-2.0_sp*alam**2
fac=2.0
probks=0.0
termbf=0.0                      Previous term in sum.
do j=1,NITER
    term=fac*exp(a2*j**2)
    probks=probks+term
    if (abs(term) <= EPS1*termbf .or. abs(term) <= EPS2*probks) RETURN
    fac=-fac                    Alternating signs in sum.
    termbf=abs(term)
end do
probks=1.0                      Get here only by failing to converge, which implies the func-
END FUNCTION probks                 tion is very close to 1.
```

⋆ ⋆ ⋆

```
SUBROUTINE cntab1(nn,chisq,df,prob,cramrv,ccc)
USE nrtype; USE nrutil, ONLY : outerprod
USE nr, ONLY : gammq
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
REAL(SP), INTENT(OUT) :: chisq,df,prob,cramrv,ccc
REAL(SP), PARAMETER :: TINY=1.0e-30_sp
```

Given a two-dimensional contingency table in the form of a rectangular integer array nn, this routine returns the chi-square chisq, the number of degrees of freedom df, the significance level prob (small values indicating a significant association), and two measures of association, Cramer's V (cramrv), and the contingency coefficient C (ccc).

```
INTEGER(I4B) :: nni,nnj
REAL(SP) :: sumn
REAL(SP), DIMENSION(size(nn,1)) :: sumi
REAL(SP), DIMENSION(size(nn,2)) :: sumj
REAL(SP), DIMENSION(size(nn,1),size(nn,2)) :: expctd
sumi(:)=sum(nn(:,:),dim=2)                Get the row totals.
sumj(:)=sum(nn(:,:),dim=1)                Get the column totals.
sumn=sum(sumi(:))                         Get the grand total.
nni=size(sumi)-count(sumi(:) == 0.0)
```

Eliminate any zero rows by reducing the number of rows.

```
nnj=size(sumj)-count(sumj(:) == 0.0)      Eliminate any zero columns.
df=nni*nnj-nni-nnj+1                      Corrected number of degrees of freedom.
expctd(:,:)=outerprod(sumi(:),sumj(:))/sumn
chisq=sum((nn(:,:)-expctd(:,:))**2/(expctd(:,:)+TINY))
```

Do the chi-square sum. Here TINY guarantees that any eliminated row or column will not contribute to the sum.

```
prob=gammq(0.5_sp*df,0.5_sp*chisq)        Chi-square probability function.
cramrv=sqrt(chisq/(sumn*min(nni-1,nnj-1)))
ccc=sqrt(chisq/(chisq+sumn))
END SUBROUTINE cntab1
```

f90 `sumi(:)=sum(...dim=2)...sumj(:)=sum(...dim=1)` We use the optional argument dim of sum to sum first over the columns (dim=2) to get the row totals, and then to sum over the rows (dim=1) to get the column totals.

`expctd(:,:)=...` This is a direct implementation of equation (14.4.2) using outerprod from nrutil.

`chisq=...` And here is a direct implementation of equation (14.4.3).

```
SUBROUTINE cntab2(nn,h,hx,hy,hygx,hxgy,uygx,uxgy,uxy)
USE nrtype
IMPLICIT NONE
INTEGER(I4B), DIMENSION(:,:), INTENT(IN) :: nn
REAL(SP), INTENT(OUT) :: h,hx,hy,hygx,hxgy,uygx,uxgy,uxy
REAL(SP), PARAMETER :: TINY=1.0e-30_sp
```

Given a two-dimensional contingency table in the form of a rectangular integer array nn, where the first index labels the x-variable and the second index labels the y variable, this routine returns the entropy h of the whole table, the entropy hx of the x-distribution, the entropy hy of the y-distribution, the entropy hygx of y given x, the entropy hxgy of x given y, the dependency uygx of y on x (eq. 14.4.15), the dependency uxgy of x on y (eq. 14.4.16), and the symmetrical dependency uxy (eq. 14.4.17).

```
REAL(SP) :: sumn
REAL(SP), DIMENSION(size(nn,1)) :: sumi
REAL(SP), DIMENSION(size(nn,2)) :: sumj
sumi(:)=sum(nn(:,:),dim=2)                Get the row totals.
sumj(:)=sum(nn(:,:),dim=1)                Get the column totals.
sumn=sum(sumi(:))
hx=-sum(sumi(:)*log(sumi(:)/sumn), mask=(sumi(:) /= 0.0) )/sumn
```

Entropy of the x distribution,

```
hy=-sum(sumj(:)*log(sumj(:)/sumn), mask=(sumj(:) /= 0.0) )/sumn
```

```
  and of the y distribution.
h=-sum(nn(:,:)*log(nn(:,:)/sumn), mask=(nn(:,:) /= 0) )/sumn
  Total entropy: loop over both x and y.
hygx=h-hx                                   Uses equation (14.4.18),
hxgy=h-hy                                   as does this.
uygx=(hy-hygx)/(hy+TINY)                    Equation (14.4.15).
uxgy=(hx-hxgy)/(hx+TINY)                    Equation (14.4.16).
uxy=2.0_sp*(hx+hy-h)/(hx+hy+TINY)           Equation (14.4.17).
END SUBROUTINE cntab2
```

f90 This code exploits both the `dim` feature of `sum` (see discussion after `cntab1`) and the `mask` feature to restrict the elements to be summed over.

⋆ ⋆ ⋆

```
SUBROUTINE pearsn(x,y,r,prob,z)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : betai
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: r,prob,z
REAL(SP), DIMENSION(:), INTENT(IN) :: x,y
REAL(SP), PARAMETER :: TINY=1.0e-20_sp
    Given two arrays x and y of the same size, this routine computes their correlation coefficient
    r (returned as r), the significance level at which the null hypothesis of zero correlation
    is disproved (prob whose small value indicates a significant correlation), and Fisher's z
    (returned as z), whose value can be used in further statistical tests as described above the
    routine in Volume 1.
    Parameter: TINY will regularize the unusual case of complete correlation.
REAL(SP), DIMENSION(size(x)) :: xt,yt
REAL(SP) :: ax,ay,df,sxx,sxy,syy,t
INTEGER(I4B) :: n
n=assert_eq(size(x),size(y),'pearsn')
ax=sum(x)/n                                          Find the means.
ay=sum(y)/n
xt(:)=x(:)-ax                                        Compute the correlation co-
yt(:)=y(:)-ay                                            efficient.
sxx=dot_product(xt,xt)
syy=dot_product(yt,yt)
sxy=dot_product(xt,yt)
r=sxy/(sqrt(sxx*syy)+TINY)
z=0.5_sp*log(((1.0_sp+r)+TINY)/((1.0_sp-r)+TINY))    Fisher's z transformation.
df=n-2
t=r*sqrt(df/(((1.0_sp-r)+TINY)*((1.0_sp+r)+TINY)))   Equation (14.5.5).
prob=betai(0.5_sp*df,0.5_sp,df/(df+t**2))            Student's t probability.
!prob=erfcc(abs(z*sqrt(n-1.0_sp))/SQRT2)
  For large n, this easier computation of prob, using the short routine erfcc, would give
  approximately the same value.
END SUBROUTINE pearsn
```

⋆ ⋆ ⋆

```
SUBROUTINE spear(data1,data2,d,zd,probd,rs,probrs)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : betai,erfcc,sort2
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
REAL(SP), INTENT(OUT) :: d,zd,probd,rs,probrs
```

Given two data arrays of the same size, `data1` and `data2`, this routine returns their sum-squared difference of ranks as D, the number of standard deviations by which D deviates from its null-hypothesis expected value as `zd`, the two-sided significance level of this deviation as `probd`, Spearman's rank correlation r_s as `rs`, and the two-sided significance level of its deviation from zero as `probrs`. `data1` and `data2` are not modified. A small value of either `probd` or `probrs` indicates a significant correlation (`rs` positive) or anticorrelation (`rs` negative).

```
INTEGER(I4B) :: n
REAL(SP) :: aved,df,en,en3n,fac,sf,sg,t,vard
REAL(SP), DIMENSION(size(data1)) :: wksp1,wksp2
n=assert_eq(size(data1),size(data2),'spear')
wksp1(:)=data1(:)
wksp2(:)=data2(:)
call sort2(wksp1,wksp2)                    Sort each of the data arrays, and convert the
call crank(wksp1,sf)                           entries to ranks. The values sf and sg
call sort2(wksp2,wksp1)                        return the sums \sum(f_k^3-f_k) and \sum(g_m^3-
call crank(wksp2,sg)                           g_m), respectively.
wksp1(:)=wksp1(:)-wksp2(:)
d=dot_product(wksp1,wksp1)                 Sum the squared difference of ranks.
en=n
en3n=en**3-en
aved=en3n/6.0_sp-(sf+sg)/12.0_sp                    Expectation value of D,
fac=(1.0_sp-sf/en3n)*(1.0_sp-sg/en3n)
vard=((en-1.0_sp)*en**2*(en+1.0_sp)**2/36.0_sp)*fac and variance of D give
zd=(d-aved)/sqrt(vard)                              number of standard deviations,
probd=erfcc(abs(zd)/SQRT2)                          and significance.
rs=(1.0_sp-(6.0_sp/en3n)*(d+(sf+sg)/12.0_sp))/sqrt(fac)   Rank correlation coeffi-
fac=(1.0_sp+rs)*(1.0_sp-rs)                                    cient,
if (fac > 0.0) then
    t=rs*sqrt((en-2.0_sp)/fac)                              and its t value,
    df=en-2.0_sp
    probrs=betai(0.5_sp*df,0.5_sp,df/(df+t**2))             give its significance.
else
    probrs=0.0
end if
CONTAINS
```

```
SUBROUTINE crank(w,s)
USE nrtype; USE nrutil, ONLY : arth,array_copy
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: s
REAL(SP), DIMENSION(:), INTENT(INOUT) :: w
```

Given a sorted array `w`, replaces the elements by their rank, including midranking of ties, and returns as `s` the sum of $f^3 - f$, where f is the number of elements in each tie.

```
INTEGER(I4B) :: i,n,ndum,nties
INTEGER(I4B), DIMENSION(size(w)) :: tstart,tend,tie,idx
n=size(w)
idx(:)=arth(1,1,n)                         Index vector.
tie(:)=merge(1,0,w == eoshift(w,-1))
```

Look for ties: Compare each element to the one before. If it's equal, it's part of a tie, and we put 1 into `tie`. Otherwise we put 0.

```
tie(1)=0                                   Boundary; the first element must be zero.
w(:)=idx(:)                                Assign ranks ignoring possible ties.
if (all(tie == 0)) then                    No ties—we're done.
    s=0.0
    RETURN
end if
call array_copy(pack(idx(:),tie(:)<eoshift(tie(:),1)),tstart,nties,ndum)
```

```
  Look for 0 → 1 transitions in tie, which mean that the 0 element is the start of a tie run.
  Store index of each transition in tstart. nties is the number of ties found.
tend(1:nties)=pack(idx(:),tie(:)>eoshift(tie(:),1))
  Look for 1 → 0 transitions in tie, which mean that the 1 element is the end of a tie run.
do i=1,nties                                          Midrank assignments.
   w(tstart(i):tend(i))=(tstart(i)+tend(i))/2.0_sp
end do
tend(1:nties)=tend(1:nties)-tstart(1:nties)+1         Now calculate s.
s=sum(tend(1:nties)**3-tend(1:nties))
END SUBROUTINE crank
END SUBROUTINE spear
```

To understand how the parallel version of `crank` works, let's consider an example of 9 elements in the array `w`, which is input in sorted order to `crank`. The elements in our example are given in the second line of the following table:

index	1	2	3	4	5	6	7	8	9	
data in `w`	0	0	1	1	1	2	3	4	4	
shift right	0	0	0	1	1	1	2	3	4	
compare	1	1	0	1	1	0	0	0	1	
tie array	0	1	0	1	1	0	0	0	1	
shift left	1	0	1	1	0	0	0	1	0	
$0 \rightarrow 1$	1		3					8		start index
$1 \rightarrow 0$		2			5				9	stop index

We look for ties by comparing this array with itself, right shifted by one element ("shift right" in table). We record a 1 for each element that is the same, a 0 for each element that is different ("compare"). A 1 indicates the element is part of a tie with the *preceding* element, so we always set the first element to 0, even if it was a 1 as in our example. This gives the "tie array." Now wherever the tie array makes a transition $0 \rightarrow 1$ indicates the start of a tie run, while a $1 \rightarrow 0$ transition indicates the end of a tie run. We find these transitions by comparing the tie array to itself left shifted by one ("shift left"). If the tie array element is smaller than the shifted array element, we have a $0 \rightarrow 1$ transition and we record the corresponding index as the start of a tie. Similarly if the tie array element is larger we record the index as the end of a tie. Note that the shifts must be end-off shifts with zeros inserted in the gaps for the boundary conditions to work.

```
call array_copy(pack(idx(:),tie(:)<eoshift(tie(:),1)),
   tstart,nties,ndum)
```

The start indices (1, 3, and 8 in our example above) are here packed into the first few elements of `tstart`. `array_copy` is a useful routine in `nrutil` for copying elements from one array to another, when the number of elements to be copied is not known in advance. This line of code is equivalent to

```
tstart(:)=0
tstart(:)=pack(idx(:), tie(:) < eoshift(tie(:),1),tstart(:))
nties=count(tstart(:) > 0)
```

The point is that we don't know how many elements `pack` is going to select. We have to make sure the dimensions of both sides of the `pack` statement are the same,

so we set the optional third argument of pack to tstart. We then make a separate pass through tstart to count how many elements we copied. Alternatively, we could have used an additional logical array mask and coded this as

```
mask(:)=tie(:) < eoshift(tie(:),1)
nties=count(mask)
tstart(1:nties)=pack(idx(:),mask)
```

But we still need two passes through the mask array. The beauty of the array_copy routine is that nties is determined from the *size* of the first argument, without the necessity for a second pass through the array.

⋆ ⋆ ⋆

```
SUBROUTINE kendl1(data1,data2,tau,z,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : erfcc
IMPLICIT NONE
REAL(SP), INTENT(OUT) :: tau,z,prob
REAL(SP), DIMENSION(:), INTENT(IN) :: data1,data2
```

Given same-size data arrays data1 and data2, this program returns Kendall's τ as tau, its number of standard deviations from zero as z, and its two-sided significance level as prob. Small values of prob indicate a significant correlation (tau positive) or anticorrelation (tau negative).

```
INTEGER(I4B) :: is,j,n,n1,n2
REAL(SP) :: var
REAL(SP), DIMENSION(size(data1)) :: a1,a2
n=assert_eq(size(data1),size(data2),'kendl1')
n1=0                                  This will be the argument of one square root in (14.6.8),
n2=0                                  and this the other.
is=0                                  This will be the numerator in (14.6.8).
do j=1,n-1                                          For each first member of pair,
    a1(j+1:n)=data1(j)-data1(j+1:n)                 loop over second member.
    a2(j+1:n)=data2(j)-data2(j+1:n)
    n1=n1+count(a1(j+1:n) /= 0.0)
    n2=n2+count(a2(j+1:n) /= 0.0)
      Now accumulate the numerator in (14.6.8):
    is=is+count((a1(j+1:n) > 0.0 .and. a2(j+1:n) > 0.0) &
        .or. (a1(j+1:n) < 0.0 .and. a2(j+1:n) < 0.0)) - &
        count((a1(j+1:n) > 0.0 .and. a2(j+1:n) < 0.0) &
        .or. (a1(j+1:n) < 0.0 .and. a2(j+1:n) > 0.0))
end do
tau=real(is,sp)/sqrt(real(n1,sp)*real(n2,sp))       Equation (14.6.8).
var=(4.0_sp*n+10.0_sp)/(9.0_sp*n*(n-1.0_sp))        Equation (14.6.9).
z=tau/sqrt(var)
prob=erfcc(abs(z)/SQRT2)                            Significance.
END SUBROUTINE kendl1
```

```
SUBROUTINE kendl2(tab,tau,z,prob)
USE nrtype; USE nrutil, ONLY : cumsum
USE nr, ONLY : erfcc
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(IN) :: tab
REAL(SP), INTENT(OUT) :: tau,z,prob
```

Given a two-dimensional table tab such that tab(k,l) contains the number of events falling in bin k of one variable and bin l of another, this program returns Kendall's τ as tau, its number of standard deviations from zero as z, and its two-sided significance level as prob. Small values of prob indicate a significant correlation (tau positive) or anticorrelation (tau

```
    negative) between the two variables. Although tab is a real array, it will normally contain
    integral values.
REAL(SP), DIMENSION(size(tab,1),size(tab,2)) :: cum,cumt
INTEGER(I4B) :: i,j,ii,jj
REAL(SP) :: sc,sd,en1,en2,points,var
ii=size(tab,1)
jj=size(tab,2)
do i=1,ii                                          Get cumulative sums leftward along
    cumt(i,jj:1:-1)=cumsum(tab(i,jj:1:-1))             rows.
end do
en2=sum(tab(1:ii,1:jj-1)*cumt(1:ii,2:jj))          Tally the extra-y pairs.
do j=1,jj                                          Get counts of points to lower-right
    cum(ii:1:-1,j)=cumsum(cumt(ii:1:-1,j))             of each cell in cum.
end do
points=cum(1,1)                                    Total number of entries in table.
sc=sum(tab(1:ii-1,1:jj-1)*cum(2:ii,2:jj))          Tally the concordant pairs.
do j=1,jj                                          Now get counts of points to upper-
    cum(1:ii,j)=cumsum(cumt(1:ii,j))                   right of each cell in cum,
end do
sd=sum(tab(2:ii,1:jj-1)*cum(1:ii-1,2:jj))          giving tally of discordant points.
do j=1,jj                                          Finally, get cumulative sums upward
    cumt(ii:1:-1,j)=cumsum(tab(ii:1:-1,j))             along columns,
end do
en1=sum(tab(1:ii-1,1:jj)*cumt(2:ii,1:jj))          giving the count of extra-x pairs,
tau=(sc-sd)/sqrt((en1+sc+sd)*(en2+sc+sd))          and compute desired results.
var=(4.0_sp*points+10.0_sp)/(9.0_sp*points*(points-1.0_sp))
z=tau/sqrt(var)
prob=erfcc(abs(z)/SQRT2)
END SUBROUTINE kendl2
```

The underlying algorithm in `kendl2` might seem to require looping over all *pairs* of cells in the two-dimensional table `tab`. Actually, however, clever use of the `cumsum` utility function reduces this to a simple loop over all the cells; moreover this "loop" parallelizes into a simple parallel product and call to the `sum` intrinsic. The basic idea is shown in the following table:

		d	d
	t	y	y
	x	c	c
	x	c	c
	x	c	c

Relative to the cell marked t (which we use to denote the numerical value it contains), the cells marked d contribute to the "discordant" tally in Volume 1's equation (14.6.8),

while the cells marked c contribute to the "concordant" tally. Likewise, the cells marked x and y contribute, respectively, to the "extra-x" and "extra-y" tallies. What about the cells left blank? Since we want to count pairs of cells only *once*, without duplication, these cells will be counted, relative to the location shown as t, when t itself moves into the blank-cell area.

Symbolically we have

$$\begin{aligned} \text{concordant} &= \sum_n t_n \left(\sum_{\text{lower right}} c_m \right) \\ \text{discordant} &= \sum_n t_n \left(\sum_{\text{upper right}} d_m \right) \\ \text{extra-}x &= \sum_n t_n \left(\sum_{\text{below}} x_m \right) \\ \text{extra-}y &= \sum_n t_n \left(\sum_{\text{to the right}} y_m \right) \end{aligned} \tag{B14.1}$$

Here n varies over all the positions in the table, while the limits of the inner sums are relative to the position of n. (The letters t_n, c_m, d_m, x_m, y_m all represent the value in a cell; we use different letters only to make the relation with the above table clear.) Now the final trick is to recognize that the inner sums, over cells to the lower- or upper-right, below, and to the right can be done in parallel by cumulative sums (`cumsum`) sweeping to the right and up. The routine does these in a nonintuitive order merely to be able to reuse maximally the scratch spaces `cum` and `cumt`.

⋆ ⋆ ⋆

```
SUBROUTINE ks2d1s(x1,y1,quadvl,d1,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : pearsn,probks,quadct
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1
REAL(SP), INTENT(OUT) :: d1,prob
INTERFACE
    SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: x,y
    REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
    END SUBROUTINE quadvl
END INTERFACE
```

Two-dimensional Kolmogorov-Smirnov test of one sample against a model. Given the x- and y-coordinates of a set of data points in arrays `x1` and `y1` of the same length, and given a user-supplied function `quadvl` that exemplifies the model, this routine returns the two-dimensional K-S statistic as `d1`, and its significance level as `prob`. Small values of `prob` show that the sample is significantly different from the model. Note that the test is slightly distribution-dependent, so `prob` is only an estimate.

```
INTEGER(I4B) :: j,n1
REAL(SP) :: dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,rr,sqen
n1=assert_eq(size(x1),size(y1),'ks2d1s')
d1=0.0
```

```
do j=1,n1                                  Loop over the data points.
    call quadct(x1(j),y1(j),x1,y1,fa,fb,fc,fd)
    call quadvl(x1(j),y1(j),ga,gb,gc,gd)
    d1=max(d1,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
      For both the sample and the model, the distribution is integrated in each of four quad-
      rants, and the maximum difference is saved.
end do
call pearsn(x1,y1,r1,dum,dumm)             Get the linear correlation coefficient r1.
sqen=sqrt(real(n1,sp))
rr=sqrt(1.0_sp-r1**2)
  Estimate the probability using the K-S probability function probks.
prob=probks(d1*sqen/(1.0_sp+rr*(0.25_sp-0.75_sp/sqen)))
END SUBROUTINE ks2d1s
```

```
SUBROUTINE quadct(x,y,xx,yy,fa,fb,fc,fd)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y
REAL(SP), DIMENSION(:), INTENT(IN) :: xx,yy
REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
```

Given an origin (x, y), and an array of points with coordinates xx and yy, count how many of them are in each quadrant around the origin, and return the normalized fractions. Quadrants are labeled alphabetically, counterclockwise from the upper right. Used by ks2d1s and ks2d2s.

```
INTEGER(I4B) :: na,nb,nc,nd,nn
REAL(SP) :: ff
nn=assert_eq(size(xx),size(yy),'quadct')
na=count(yy(:) > y .and. xx(:) > x)
nb=count(yy(:) > y .and. xx(:) <= x)
nc=count(yy(:) <= y .and. xx(:) <= x)
nd=nn-na-nb-nc
ff=1.0_sp/nn
fa=ff*na
fb=ff*nb
fc=ff*nc
fd=ff*nd
END SUBROUTINE quadct
```

```
SUBROUTINE quadvl(x,y,fa,fb,fc,fd)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: x,y
REAL(SP), INTENT(OUT) :: fa,fb,fc,fd
```

This is a sample of a user-supplied routine to be used with ks2d1s. In this case, the model distribution is uniform inside the square $-1 < x < 1$, $-1 < y < 1$. In general this routine should return, for any point (x, y), the fraction of the total distribution in each of the four quadrants around that point. The fractions, fa, fb, fc, and fd, must add up to 1. Quadrants are alphabetical, counterclockwise from the upper right.

```
REAL(SP) :: qa,qb,qc,qd
qa=min(2.0_sp,max(0.0_sp,1.0_sp-x))
qb=min(2.0_sp,max(0.0_sp,1.0_sp-y))
qc=min(2.0_sp,max(0.0_sp,x+1.0_sp))
qd=min(2.0_sp,max(0.0_sp,y+1.0_sp))
fa=0.25_sp*qa*qb
fb=0.25_sp*qb*qc
fc=0.25_sp*qc*qd
fd=0.25_sp*qd*qa
END SUBROUTINE quadvl
```

```
SUBROUTINE ks2d2s(x1,y1,x2,y2,d,prob)
USE nrtype; USE nrutil, ONLY : assert_eq
USE nr, ONLY : pearsn,probks,quadct
IMPLICIT NONE
REAL(SP), DIMENSION(:), INTENT(IN) :: x1,y1,x2,y2
REAL(SP), INTENT(OUT) :: d,prob
```

Compute two-dimensional Kolmogorov-Smirnov test on two samples. Input are the x- and y-coordinates of the first sample in arrays x1 and y1 of the same length, and of the second sample in arrays x2 and y2 of the same length (possibly different from the length of the first sample). The routine returns the two-dimensional, two-sample K-S statistic as d, and its significance level as prob. Small values of prob show that the two samples are significantly different. Note that the test is slightly distribution-dependent, so prob is only an estimate.

```
INTEGER(I4B) :: j,n1,n2
REAL(SP) :: d1,d2,dum,dumm,fa,fb,fc,fd,ga,gb,gc,gd,r1,r2,rr,sqen
n1=assert_eq(size(x1),size(y1),'ks2d2s: n1')
n2=assert_eq(size(x2),size(y2),'ks2d2s: n2')
d1=0.0
do j=1,n1                                   First, use points in the first sample as origins.
    call quadct(x1(j),y1(j),x1,y1,fa,fb,fc,fd)
    call quadct(x1(j),y1(j),x2,y2,ga,gb,gc,gd)
    d1=max(d1,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
end do
d2=0.0
do j=1,n2                                   Then, use points in the second sample as ori-
    call quadct(x2(j),y2(j),x1,y1,fa,fb,fc,fd)      gins.
    call quadct(x2(j),y2(j),x2,y2,ga,gb,gc,gd)
    d2=max(d2,abs(fa-ga),abs(fb-gb),abs(fc-gc),abs(fd-gd))
end do
d=0.5_sp*(d1+d2)                            Average the K-S statistics.
sqen=sqrt(real(n1,sp)*real(n2,sp)/real(n1+n2,sp))
call pearsn(x1,y1,r1,dum,dumm)              Get the linear correlation coefficient for each sam-
call pearsn(x2,y2,r2,dum,dumm)                  ple.
rr=sqrt(1.0_sp-0.5_sp*(r1**2+r2**2))
```

Estimate the probability using the K-S probability function probks.

```
prob=probks(d*sqen/(1.0_sp+rr*(0.25_sp-0.75_sp/sqen)))
END SUBROUTINE ks2d2s
```

★ ★ ★

```
FUNCTION savgol(nl,nrr,ld,m)
USE nrtype; USE nrutil, ONLY : arth,assert,poly
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: nl,nrr,ld,m
```

Returns in array c, in wrap-around order (N.B.!) consistent with the argument respns in routine convlv, a set of Savitzky-Golay filter coefficients. nl is the number of leftward (past) data points used, while nrr is the number of rightward (future) data points, making the total number of data points used $\texttt{nl}+\texttt{nrr}+1$. ld is the order of the derivative desired (e.g., $\texttt{ld}=0$ for smoothed function). m is the order of the smoothing polynomial, also equal to the highest conserved moment; usual value is $\texttt{m}=2$ or $\texttt{m}=4$.

```
REAL(SP), DIMENSION(nl+nrr+1) :: savgol
INTEGER(I4B) :: imj,ipj,mm,np
INTEGER(I4B), DIMENSION(m+1) :: indx
REAL(SP) :: d,sm
REAL(SP), DIMENSION(m+1) :: b
REAL(SP), DIMENSION(m+1,m+1) :: a
INTEGER(I4B) :: irng(nl+nrr+1)
call assert(nl >= 0, nrr >= 0, ld <= m, nl+nrr >= m, 'savgol args')
do ipj=0,2*m                                Set up the normal equations of the desired least
    sm=sum(arth(1.0_sp,1.0_sp,nrr)**ipj)+&      squares fit.
        sum(arth(-1.0_sp,-1.0_sp,nl)**ipj)
```

```
    if (ipj == 0) sm=sm+1.0_sp
    mm=min(ipj,2*m-ipj)
    do imj=-mm,mm,2
        a(1+(ipj+imj)/2,1+(ipj-imj)/2)=sm
    end do
end do
call ludcmp(a(:,:),indx(:),d)          Solve them: LU decomposition.
b(:)=0.0
b(ld+1)=1.0                            Right-hand-side vector is unit vector, depending
call lubksb(a(:,:),indx(:),b(:))           on which derivative we want.
  Backsubstitute, giving one row of the inverse matrix.
savgol(:)=0.0                          Zero the output array (it may be bigger than
irng(:)=arth(-nl,1,nrr+nl+1)               number of coefficients).
np=nl+nrr+1
savgol(mod(np-irng(:),np)+1)=poly(real(irng(:),sp),b(:))
  Each Savitzky-Golay coefficient is the value of the polynomial in (14.8.6) at the corresponding
  integer. The polynomial coefficients are a row of the inverse matrix. The mod function takes
  care of the wrap-around order.
END FUNCTION savgol
```

f90 `do imj=-mm,mm,2` Here is an example of a loop that cannot be parallelized in the framework of Fortran 90: We need to access "skew" sections of the matrix `a`.

`savgol...=poly(real(irng(:),sp),b(:)))` The `poly` function in `nrutil` returns the value of a polynomial, here the one in equation (14.8.6). We need the explicit kind type parameter `sp` in the `real` function, otherwise it would return type default real for the integer argument and would not automatically convert to double precision if desired.