# Chapter B18. Integral Equations and Inverse Theory

```
SUBROUTINE fred2(a,b,t,f,w,g,ak)
USE nrtype; USE nrutil, ONLY : assert_eq,unit_matrix
USE nr, ONLY : gauleg,lubksb,ludcmp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(OUT) :: t,f,w
INTERFACE
    FUNCTION g(t)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: t
    REAL(SP), DIMENSION(size(t)) :: g
    END FUNCTION g

    FUNCTION ak(t,s)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
    REAL(SP), DIMENSION(size(t),size(s)) :: ak
    END FUNCTION ak
END INTERFACE
```

Solves a linear Fredholm equation of the second kind by $N$-point Gaussian quadrature. On input, a and b are the limits of integration. g and ak are user-supplied external functions. g returns $g(t)$ as a vector of length $N$ for a vector of $N$ arguments, while ak returns $\lambda K(t,s)$ as an $N \times N$ matrix. The routine returns arrays t and f of length $N$ containing the abscissas $t_i$ of the Gaussian quadrature and the solution $f$ at these abscissas. Also returned is the array w of length $N$ of Gaussian weights for use with the Nystrom interpolation routine fredin.

```
INTEGER(I4B) :: n
INTEGER(I4B), DIMENSION(size(f)) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(size(f),size(f)) :: omk
n=assert_eq(size(f),size(t),size(w),'fred2')
call gauleg(a,b,t,w)                          Replace gauleg with another routine if not
                                                  using Gauss-Legendre quadrature.
call unit_matrix(omk)
omk=omk-ak(t,t)*spread(w,dim=1,ncopies=n)     Form 1 − λK̃.
f=g(t)
call ludcmp(omk,indx,d)                       Solve linear equations.
call lubksb(omk,indx,f)
END SUBROUTINE fred2
```

f90 `call unit_matrix(omk)` The `unit_matrix` routine in `nrutil` does exactly what its name suggests.

`omk=omk-ak(t,t)*spread(w,dim=1,ncopies=n)` By now this idiom should be second nature: the first column of `ak` gets multiplied by the first element of `w`, and so on.

⋆ ⋆ ⋆

```
FUNCTION fredin(x,a,b,t,f,w,g,ak)
USE nrtype; USE nrutil, ONLY : assert_eq
IMPLICIT NONE
REAL(SP), INTENT(IN) :: a,b
REAL(SP), DIMENSION(:), INTENT(IN) :: x,t,f,w
REAL(SP), DIMENSION(size(x)) :: fredin
INTERFACE
    FUNCTION g(t)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: t
    REAL(SP), DIMENSION(size(t)) :: g
    END FUNCTION g

    FUNCTION ak(t,s)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), DIMENSION(:), INTENT(IN) :: t,s
    REAL(SP), DIMENSION(size(t),size(s)) :: ak
    END FUNCTION ak
END INTERFACE
    Input are arrays t and w of length N containing the abscissas and weights of the N-point
    Gaussian quadrature, and the solution array f of length N from fred2. The function
    fredin returns the array of values of f at an array of points x using the Nystrom interpo-
    lation formula. On input, a and b are the limits of integration. g and ak are user-supplied
    external functions. g returns g(t) as a vector of length N for a vector of N arguments,
    while ak returns λK(t,s) as an N × N matrix.
INTEGER(I4B) :: n
n=assert_eq(size(f),size(t),size(w),'fredin')
fredin=g(x)+matmul(ak(x,t),w*f)
END FUNCTION fredin
```

f90 `fredin=g(x)+matmul...` Fortran 90 allows very concise coding here, which also happens to be much closer to the mathematical formulation than the loops required in Fortran 77.

⋆ ⋆ ⋆

```
SUBROUTINE voltra(t0,h,t,f,g,ak)
USE nrtype; USE nrutil, ONLY : array_copy,assert_eq,unit_matrix
USE nr, ONLY : lubksb,ludcmp
IMPLICIT NONE
REAL(SP), INTENT(IN) :: t0,h
REAL(SP), DIMENSION(:), INTENT(OUT) :: t
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: f
INTERFACE
    FUNCTION g(t)
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: t
    REAL(SP), DIMENSION(:), POINTER :: g
    END FUNCTION g

    FUNCTION ak(t,s)
```

```
    USE nrtype
    IMPLICIT NONE
    REAL(SP), INTENT(IN) :: t,s
    REAL(SP), DIMENSION(:,:), POINTER :: ak
    END FUNCTION ak
END INTERFACE
```

Solves a set of $M$ linear Volterra equations of the second kind using the extended trapezoidal rule. On input, t0 is the starting point of the integration. The routine takes $N-1$ steps of size h and returns the abscissas in t, a vector of length $N$. The solution at these points is returned in the $M \times N$ matrix f. g is a user-supplied external function that returns a pointer to the $M$-dimensional vector of functions $g_k(t)$, while ak is another user-supplied external function that returns a pointer to the $M \times M$ matrix $K(t,s)$.

```
INTEGER(I4B) :: i,j,n,ncop,nerr,m
INTEGER(I4B), DIMENSION(size(f,1)) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(size(f,1)) :: b
REAL(SP), DIMENSION(size(f,1),size(f,1)) :: a
n=assert_eq(size(f,2),size(t),'voltra: n')
t(1)=t0                                              Initialize.
call array_copy(g(t(1)),f(:,1),ncop,nerr)
m=assert_eq(size(f,1),ncop,ncop+nerr,'voltra: m')
do i=2,n                                             Take a step h.
    t(i)=t(i-1)+h
    b=g(t(i))+0.5_sp*h*matmul(ak(t(i),t(1)),f(:,1))  Accumulate right-hand side
    do j=2,i-1                                           of linear equations in b.
        b=b+h*matmul(ak(t(i),t(j)),f(:,j))
    end do
    call unit_matrix(a)                              Left-hand side goes in ma-
    a=a-0.5_sp*h*ak(t(i),t(i))                           trix a.
    call ludcmp(a,indx,d)                            Solve linear equations.
    call lubksb(a,indx,b)
    f(:,i)=b(:)
end do
END SUBROUTINE voltra
```

f90 `FUNCTION g(t)...REAL(SP), DIMENSION(:), POINTER :: g` The routine voltra requires an argument that is a function returning a vector, but we don't know the dimension of the vector at compile time. The solution is to make the function return a *pointer* to the vector. This is not the same thing as a pointer to a function, which is not allowed in Fortran 90. When you use the pointer in the routine, Fortran 90 figures out from the context that you want the vector of values, so the code remains highly readable. Similarly, the argument ak is a function returning a pointer to a matrix.

The coding of the user-supplied functions g and ak deserves some comment: functions returning pointers to arrays are potential memory leaks if the arrays are allocated dynamically in the functions. Here the user knows in advance the dimension of the problem, and so there is no need to use dynamical allocation in the functions. For example, in a two-dimensional problem, you can code g as follows:

```
FUNCTION g(t)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: t
REAL(SP), DIMENSION(:), POINTER :: g
REAL(SP), DIMENSION(2), TARGET, SAVE :: gg
g=>gg
g(1)=...
g(2)=...
END FUNCTION g
```

and similarly for ak.

Suppose, however, we coded g with dynamical allocation:

```
FUNCTION g(t)
USE nrtype
IMPLICIT NONE
REAL(SP), INTENT(IN) :: t
REAL(SP), DIMENSION(:), POINTER :: g
allocate(g(2))
g(1)=...
g(2)=...
END FUNCTION g
```

Now g never gets deallocated; each time we call the function fresh memory gets consumed. If you have a problem that really does require dynamical allocation in a pointer function, you have to be sure to deallocate the pointer in the calling routine. In voltra, for example, we would declare pointers gtemp and aktemp. Then instead of writing simply

```
b=g(t(i))+...
```

we would write

```
gtemp=>g(t(i))
b=gtemp+...
deallocate(gtemp)
```

and similarly for each pointer function invocation.

`call array_copy(g(t(1)),f(:,1),ncop,nerr)` The routine would work if we replaced this statement with simply `f(:,1)=g(t(1))`. The purpose of using array_copy from nrutil is that we can check that f and g have consistent dimensions with a call to assert_eq.

⋆ ⋆ ⋆

```
FUNCTION wwghts(n,h,kermom)
USE nrtype; USE nrutil, ONLY : geop
IMPLICIT NONE
INTEGER(I4B), INTENT(IN) :: n
REAL(SP), INTENT(IN) :: h
REAL(SP), DIMENSION(n) :: wwghts
INTERFACE
    FUNCTION kermom(y,m)
    USE nrtype
    IMPLICIT NONE
    REAL(DP), INTENT(IN) :: y
    INTEGER(I4B), INTENT(IN) :: m
    REAL(DP), DIMENSION(m) :: kermom
    END FUNCTION kermom
END INTERFACE
```

Returns in wwghts(1:n) weights for the n-point equal-interval quadrature from 0 to $(n-1)h$ of a function $f(x)$ times an arbitrary (possibly singular) weight function $w(x)$ whose indefinite-integral moments $F_n(y)$ are provided by the user-supplied function kermom.

```
INTEGER(I4B) :: j
REAL(DP) :: hh,hi,c,a,b
REAL(DP), DIMENSION(4) :: wold,wnew,w
hh=h                                  Double precision on internal calculations even though
hi=1.0_dp/hh                              the interface is in single precision.
wwghts(1:n)=0.0                       Zero all the weights so we can sum into them.
wold(1:4)=kermom(0.0_dp,4)            Evaluate indefinite integrals at lower end.
```

```
if (n >= 4) then                        Use highest available order.
    b=0.0                               For another problem, you might change this lower
    do j=1,n-3                              limit.
        c=j-1                           This is called k in equation (18.3.5).
        a=b                             Set upper and lower limits for this step.
        b=a+hh
        if (j == n-3) b=(n-1)*hh        Last interval: go all the way to end.
        wnew(1:4)=kermom(b,4)
        w(1:4)=(wnew(1:4)-wold(1:4))*geop(1.0_dp,hi,4)      Equation (18.3.4).
        wwghts(j:j+3)=wwghts(j:j+3)+(/&                     Equation (18.3.5).
            ((c+1.0_dp)*(c+2.0_dp)*(c+3.0_dp)*w(1)&
            -(11.0_dp+c*(12.0_dp+c*3.0_dp))*w(2)&
                +3.0_dp*(c+2.0_dp)*w(3)-w(4))/6.0_dp,&
            (-c*(c+2.0_dp)*(c+3.0_dp)*w(1)&
            +(6.0_dp+c*(10.0_dp+c*3.0_dp))*w(2)&
                -(3.0_dp*c+5.0_dp)*w(3)+w(4))*0.50_dp,&
            (c*(c+1.0_dp)*(c+3.0_dp)*w(1)&
            -(3.0_dp+c*(8.0_dp+c*3.0_dp))*w(2)&
                +(3.0_dp*c+4.0_dp)*w(3)-w(4))*0.50_dp,&
            (-c*(c+1.0_dp)*(c+2.0_dp)*w(1)&
            +(2.0_dp+c*(6.0_dp+c*3.0_dp))*w(2)&
            -3.0_dp*(c+1.0_dp)*w(3)+w(4))/6.0_dp /)
        wold(1:4)=wnew(1:4)             Reset lower limits for moments.
    end do
else if (n == 3) then                   Lower-order cases; not recommended.
    wnew(1:3)=kermom(hh+hh,3)
    w(1:3)= (/ wnew(1)-wold(1), hi*(wnew(2)-wold(2)),&
        hi**2*(wnew(3)-wold(3)) /)
    wwghts(1:3)= (/ w(1)-1.50_dp*w(2)+0.50_dp*w(3),&
        2.0_dp*w(2)-w(3), 0.50_dp*(w(3)-w(2)) /)
else if (n == 2) then
    wnew(1:2)=kermom(hh,2)
    wwghts(2)=hi*(wnew(2)-wold(2))
    wwghts(1)=wnew(1)-wold(1)-wwghts(2)
end if
END FUNCTION wwghts
```

★ ★ ★

```
MODULE kermom_info
USE nrtype
REAL(DP) :: kermom_x
END MODULE kermom_info
```

```
FUNCTION kermom(y,m)
USE nrtype
USE kermom_info
IMPLICIT NONE
REAL(DP), INTENT(IN) :: y
INTEGER(I4B), INTENT(IN) :: m
REAL(DP), DIMENSION(m) :: kermom
```

Returns in kermom(1:m) the first m indefinite-integral moments of one row of the singular part of the kernel. (For this example, m is hard-wired to be 4.) The input variable y labels the column, while kermom_x (in the module kermom_info) is the row.

```
REAL(DP) :: x,d,df,clog,x2,x3,x4
x=kermom_x                      We can take x as the lower limit of integration. Thus, we
if (y >= x) then                    return the moment integrals either purely to the left or
    d=y-x                           purely to the right of the diagonal.
    df=2.0_dp*sqrt(d)*d
    kermom(1:4) = (/ df/3.0_dp, df*(x/3.0_dp+d/5.0_dp),&
```

```
        df*((x/3.0_dp + 0.4_dp*d)*x + d**2/7.0_dp),&
        df*(((x/3.0_dp + 0.6_dp*d)*x + 3.0_dp*d**2/7.0_dp)*x&
            + d**3/9.0_dp) /)
else
    x2=x**2
    x3=x2*x
    x4=x2*x2
    d=x-y
    clog=log(d)
    kermom(1:4) = (/ d*(clog-1.0_dp),&
        -0.25_dp*(3.0_dp*x+y-2.0_dp*clog*(x+y))*d,&
        (-11.0_dp*x3+y*(6.0_dp*x2+y*(3.0_dp*x+2.0_dp*y))&
            +6.0_dp*clog*(x3-y**3))/18.0_dp,&
        (-25.0_dp*x4+y*(12.0_dp*x3+y*(6.0_dp*x2+y*&
            (4.0_dp*x+3.0_dp*y)))+12.0_dp*clog*(x4-y**4))/48.0_dp /)
end if
END FUNCTION kermom
```

f90 `MODULE kermom_info` This module functions just like a common block to share the variable `kermom_x` with the routine `quadmx`.

⋆ ⋆ ⋆

```
SUBROUTINE quadmx(a)
USE nrtype; USE nrutil, ONLY : arth,assert_eq,diagadd,outerprod
USE nr, ONLY : wwghts,kermom
USE kermom_info
IMPLICIT NONE
REAL(SP), DIMENSION(:,:), INTENT(OUT) :: a
```

Constructs in the $N \times N$ array `a` the quadrature matrix for an example Fredholm equation of the second kind. The nonsingular part of the kernel is computed within this routine, while the quadrature weights that integrate the singular part of the kernel are obtained via calls to `wwghts`. An external routine `kermom`, which supplies indefinite-integral moments of the singular part of the kernel, is passed to `wwghts`.

```
INTEGER(I4B) :: j,n
REAL(SP) :: h,x
REAL(SP), DIMENSION(size(a,1)) :: wt
n=assert_eq(size(a,1),size(a,2),'quadmx')
h=PI/(n-1)
do j=1,n
    x=(j-1)*h
    kermom_x=x                      Put x in the module kermom_info for use by kermom.
    wt(:)=wwghts(n,h,kermom)        Part of nonsingular kernel.
    a(j,:)=wt(:)                    Put together all the pieces of the kernel.
end do
wt(:)=cos(arth(0,1,n)*h)
a(:,:)=a(:,:)*outerprod(wt(:),wt(:))
call diagadd(a,1.0_sp)              Since equation of the second kind, there is diagonal
END SUBROUTINE quadmx                   piece independent of h.
```

f90 `call diagadd...` See discussion of `diagadd` after `hqr` on p. 1234.

⋆ ⋆ ⋆

```
PROGRAM fredex
USE nrtype; USE nrutil, ONLY : arth
USE nr, ONLY : quadmx,ludcmp,lubksb
IMPLICIT NONE
INTEGER(I4B), PARAMETER :: N=40
INTEGER(I4B) :: j
INTEGER(I4B), DIMENSION(N) :: indx
REAL(SP) :: d
REAL(SP), DIMENSION(N) :: g,x
REAL(SP), DIMENSION(N,N) :: a
```

This sample program shows how to solve a Fredholm equation of the second kind using the product Nystrom method and a quadrature rule especially constructed for a particular, singular, kernel.

Parameter: N is the size of the grid.

```
call quadmx(a)                Make the quadrature matrix; all the action is here.
call ludcmp(a,indx,d)         Decompose the matrix.
x(:)=arth(0,1,n)*PI/(n-1)
g(:)=sin(x(:))                Construct the right-hand side, here sin x.
call lubksb(a,indx,g)         Backsubstitute.
do j=1,n                      Write out the solution.
    write (*,*) j,x(j),g(j)
end do
write (*,*) 'normal completion'
END PROGRAM fredex
```