

## Exercises Lecture III

### Numerical analysis: Deterministic methods for numerical Integration

#### 1. 1D integration with equispaced points: trapezoidal & Simpson rules

Consider the definite integral:

$$I = \int_0^1 e^x dx = e - 1 = 1.718282\dots$$

Write a code (e.g. `int.f90`) to calculate the integral using the (1) trapezoidal rule or (2) the Simpson rule. In general, we indicate with  $F_n$  the estimate of the integral from  $x_0$  to  $x_n$  using a discretisation in  $n$  intervals (even for the Simpson algorithm) of width  $h = \frac{x_n - x_0}{n}$ . Therefore:

$$\int_{x_0}^{x_n} f(x)dx = F_n^{trap} + \mathcal{O}(h^2) = F_n^{Simpson} + \mathcal{O}(h^4)$$

where

$$F_n^{trap} = h \left[ \frac{1}{2}f_0 + f_1 + \dots + f_{n-1} + \frac{1}{2}f_n \right]$$

and

$$F_n^{Simpson} = h \left[ \frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \dots + \frac{4}{3}f_{n-3} + \frac{2}{3}f_{n-2} + \frac{4}{3}f_{n-1} + \frac{1}{3}f_n \right]$$

- (a) Which is the dependence on  $n$  of the error  $\Delta_n = F_n - I$ ? You can choose  $n = 2^k$  (with  $k = 2, \dots, 8$ , at least) in order to have equispaced points when doing a log-log plot. You should find  $\Delta_n \approx 1/n^2$  for the *trapezoidal rule* and  $\Delta_n \approx 1/n^4$  for the *Simpson rule*.

## 2. 1D integration: Gaussian Quadrature

The Gauss-Legendre rule for *quadrature* makes use of non equispaced points with specific weights:

$$\int_a^b f(x)dx \approx \sum_{i=1}^n w(i)f(x(i))$$

For integration in  $[-1,1]$ ,  $x_i$  are the roots of the Legendre polynomials: abscissas and weights up to the fourth order, and the degree of the polynomial exactly integrable are listed in the following tables:

$n$	$i$	$x_i$	$w_i$	degree
1	1	0	2	1
2	1	-0.577350269189626	1	3
	2	0.577350269189626	1	
3	1	-0.774596669241483	0.555555555555556	5
	2	0	0.888888888888889	
	3	0.774596669241483	0.555555555555556	
4	1	-0.861136311594053	0.347854845137454	7
	2	-0.339981043584856	0.652145154862546	
	3	0.339981043584856	0.652145154862546	
	4	0.861136311594053	0.347854845137454	

We can transform the special points and weights for integration in an arbitrary interval  $[a, b]$  with the substitution ("new" refers to  $[a, b]$ , "old" to  $[-1, 1]$ ):

$$x_{new} = \frac{b-a}{2}x_{old} + \frac{b+a}{2} \quad \text{and} \quad w_{new} = \frac{b-a}{2}w_{old}$$

- (a) Consider once again the definite integral

$$I = \int_0^1 e^x dx = e - 1$$

whose numerical estimate  $F_N$  has been already calculated using (1) the trapezoidal and (2) the Simpson's rule in the previous exercise. Now we use (3) the Gauss-Legendre quadrature. Here is listed a simple program implementing explicitly the second-order formula (`gauleg-IIorder.f90`). Verify that already at this order, the Gauss-Legendre quadrature gives a very good approximation.

- (b) A more general implementation of Gauss-Legendre is proposed in `gauleg-other.f90` which makes use of the subroutine `gauleg` from "Numerical Recipes" (but the code is self-contained, it can be used without any external routine/module/interface). Estimate the relative error

$$\epsilon = \left| \frac{\text{numeric} - \text{exact}}{\text{exact}} \right|$$

for the 3 different methods, considering e.g.  $N=2, 4, 8, 16, 32, 64$ . Make a log-log plot of  $|\epsilon|$  as a function of  $N$ . What about the dependence of the error on  $N$ ? Can you identify the range of  $N$  where the roundoff errors are dominant? (consider the possibility of increasing the precision).

- (c) The program `gauleg_nr_test.f90` is another example of the use of the subroutine `gauleg` from "Numerical Recipes"; where the subroutine and other auxiliary routines/module/interface are external and must be compiled and linked. They are extracted from the "Numerical Recipes" library, properly simplified (the original versions contain more and more subroutines) and are listed at the end of these notes.

In order to use the routines of Numerical Recipes, you have to compile and link the main program with:

- the subroutine `gauleg` which gives points and abscissas
- `nrtype.f90` containing type declarations;
- `nrutil.f90` containing modules and utilities;
- `nr.f90` containing (through a module with `interfaces`) the conventions to call the subroutines with the main program

You must compile these files with the option `-c`: this produces `.mod` and `.o` (the objects). In a second step compile the main program. Finally you link all the files `.o` and produce the executable:

```
g95 -c nrtype.f90 nrutil.f90 nr.f90 gauleg.f90
g95 -c gauleg_nr_test.f90
g95 -o a.out gauleg_nr_test.o nrtype.o nrutil.o nr.o gauleg.o
```

### 3. 2D integration

- (a) Consider the classic rectangle integration formula with subdivision of the integration interval into  $2^p$  parts ( $p=2, 3, \dots$  max. 7) in each dimension. Consider for instance the function

$$f(x, y) = x^2 + 6xy + y^2$$

and estimate the integral

$$I = \int \int_{x^2+y^2 \leq 1} f(x, y) dx dy$$

Find the error as a function of the number  $n$  of points. You should see that if an algorithm goes like  $n^{-a}$  in 1D, it goes like  $n^{-a/d}$  in  $d$ -dimensions.

```

!CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
!   int.f90:
!   integrates f(x)=exp(x) in the interval [vmin,vamx]=[0,1]
!   using trapezoidal and Simpson rule
!cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

module intmod
  public :: f, trapez, simpson
contains

  ! function to be integrated
  !
  function f(x)
    implicit none
    real :: f
    real, intent(in) :: x
    f = exp(x)
    return
  end function f

  ! trapezoidal rule
  !
  function trapez(i, min, max)
    implicit none
    real :: trapez
    integer, intent(in) :: i
    real, intent(in) :: min, max
    integer :: n
    real :: x, interval
    trapez = 0.

```

```

        interval= ((max-min) / (i-1))
        ! sum over the internal points (extrema excluded)
        do n = 2, i-1
            x = interval * (n-1)
            trapez = trapez + f(x) * interval
        end do
        ! add extrema
        trapez = trapez + 0.5 * (f(min)+f(max)) * interval
        return
    end function trapez

! Simpson rule
!
function simpson(i, min, max)
    implicit none
    real :: simpson
    integer, intent(in) :: i
    real, intent(in) :: min, max
    integer :: n
    real :: x, interval
    simpson = 0.
    interval = ((max-min) / (i-1))
    ! loop EVEN points
    do n = 2, i-1, 2
        x = interval * (n-1)
        simpson = simpson + 4*f(x)
    end do
    ! loop ODD points
    do n = 3, i-1, 2
        x = interval * (n-1)
        simpson = simpson + 2*f(x)
    end do
    ! add extrema
    simpson = simpson + f(min)+f(max)
    simpson = simpson * interval/3
    return
end function simpson

end module intmod

program int
    use intmod
    !
    ! variable declaration
    !     accuracy limit
    !     min and max in x

```

```

!
implicit none
real :: r1, r2, theo, vmin, vmax, t0, t1
integer :: i, n
! exact value
vmin = 0.0
vmax = 1.0
theo = exp(vmax)-exp(vmin)
print*, ' exact value =', theo
open(unit=7, file='int-tra-sim.dat', status='unknown')
!
write(7,*)"# N, interval, exact, Trap-exact, Simpson-exact"
call cpu_time(t0)
do i = 2,8
  n = 2**i
  r1 = trapez(n+1, vmin, vmax)
  r1 = (r1-theo)
  r2 = simpson(n+1, vmin, vmax)
  r2 = (r2-theo)
  write(7,'(i4,4(2x,f10.6))') n, 1./n, theo, r1, r2
end do
call cpu_time(t1)
print*," total time spent:",t1-t0
close(7)
print*, ' data saved in int-tra-sim.dat (|diff from exact value|)'
stop
end program int

```



```

!cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
! gauleg-other.f90      P145 Numerical Recipes in Fortran
! adapted from www.cs.umbbc.edu/~squire/download/gauleg.f90
! (everything self-contained!)
! compute x(i) and w(i)  i=1,n Legendre ordinates and weights in (-1,1)
!cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

```

```

subroutine gaulegf(x1, x2, x, w, n)
  implicit none
  integer, intent(in) :: n
  double precision, intent(in) :: x1, x2
  double precision, dimension(n), intent(out) :: x, w
  integer :: i, j, m
  double precision :: p1, p2, p3, pp, xl, xm, z, z1
  double precision, parameter :: eps=3.d-14

```

```

  m = (n+1)/2
  xm = 0.5d0*(x2+x1)
  xl = 0.5d0*(x2-x1)
  do i=1,m
    z = cos(3.141592654d0*(i-0.25d0)/(n+0.5d0))
    z1 = 0.0
    do while(abs(z-z1) > eps)
      p1 = 1.0d0
      p2 = 0.0d0
      do j=1,n
        p3 = p2
        p2 = p1
        p1 = ((2.0d0*j-1.0d0)*z*p2-(j-1.0d0)*p3)/j
      end do
      pp = n*(z*p1-p2)/(z*z-1.0d0)
      z1 = z
      z = z1 - p1/pp
    end do
    x(i) = xm - xl*z
    x(n+1-i) = xm + xl*z
    w(i) = (2.0d0*xl)/((1.0d0-z*z)*pp*pp)
    w(n+1-i) = w(i)
  end do

```

```

end subroutine gaulegf

```

```

program gauleg
  implicit none
  integer :: i, j
  double precision, dimension(100) :: x, w
  double precision :: sum, a, b

```



```

integer, parameter :: debug=0

print *, 'test gauleg.f90 on interval -1.0 to 1.0 ordinates, weights'
do i=1,15
  call gaulegf(-1.0d0, 1.0d0, x, w, i)
  sum = 0.0d0
  do j=1,i
    print *, 'x(' ,j,')=' , x(j), ' w(' ,j,')=' , w(j)
    sum = sum + w(j)
  end do
  print *, '          integrate(1.0, from -1.0 to 1.0)= ', sum
print *, ' '
end do

a = 0.5d0
b = 1.0d0
print *, 'test gauleg on integral(sin(x), from ',a,' to ',b,')'
do i=2,10
  call gaulegf(a, b, x, w, i)
  sum = 0.0d0
  do j=1,i
    if(debug>0)print *, 'x(' ,j,')=' , x(j), ' w(' ,j,')=' , w(j)
    sum = sum +w(j)*sin(x(j))
  end do
  print *, i, ' integral (0.5,1.0) sin(x) dx = ', sum
end do
print *, '-cos(1.0)+cos(0.5) =', -cos(b)+cos(a)
print *, 'exact should be: 0.3372802560'
print *, ' '

a = 0.5d0
b = 5.0d0
print *, 'test gauleg on integral(exp(x), from ',a,' to ',b,')'
do i=2,10
  call gaulegf(a, b, x, w, i)
  sum = 0.0d0
  do j=1,i
    if(debug>0) print *, 'x(' ,j,')=' , x(j), ' w(' ,j,')=' , w(j)
    sum = sum + w(j)*exp(x(j))
  end do
  print *, i, ' integral (0.5,5.0) exp(x) dx = ', sum
end do
print *, 'exp(5.0)-exp(0.5) =', exp(b)-exp(a)
print *, 'exact should be: 146.7644378'
print *, ' '
end program gauleg

```









```

    n_copied=min(size(src),size(dest))
    n_not_copied=size(src)-n_copied
    dest(1:n_copied)=src(1:n_copied)
END SUBROUTINE array_copy_i

FUNCTION assert_eq2(n1,n2,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: n1,n2
  INTEGER :: assert_eq2
  if (n1 == n2) then
    assert_eq2=n1
  else
    write (*,*) 'nrerror: an assert_eq failed with this tag:',string
    STOP 'program terminated by assert_eq2'
  end if
END FUNCTION assert_eq2

FUNCTION assert_eq3(n1,n2,n3,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: n1,n2,n3
  INTEGER :: assert_eq3
  if (n1 == n2 .and. n2 == n3) then
    assert_eq3=n1
  else
    write (*,*) 'nrerror: an assert_eq failed with this tag:',string
    STOP 'program terminated by assert_eq3'
  end if
END FUNCTION assert_eq3

FUNCTION assert_eq4(n1,n2,n3,n4,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  INTEGER, INTENT(IN) :: n1,n2,n3,n4
  INTEGER :: assert_eq4
  if (n1 == n2 .and. n2 == n3 .and. n3 == n4) then
    assert_eq4=n1
  else
    write (*,*) 'nrerror: an assert_eq failed with this tag:',string
    STOP 'program terminated by assert_eq4'
  end if
END FUNCTION assert_eq4

FUNCTION assert_eqn(nn,string)
  CHARACTER(LEN=*), INTENT(IN) :: string
  INTEGER, DIMENSION(:), INTENT(IN) :: nn
  INTEGER :: assert_eqn
  if (all(nn(2:) == nn(1))) then

```

```

        assert_eqn=nn(1)
    else
        write (*,*) 'nerror: an assert_eq failed with this tag:',string
        STOP 'program terminated by assert_eqn'
    end if
END FUNCTION assert_eqn

SUBROUTINE nrerror(string)
    CHARACTER(LEN=*), INTENT(IN) :: string
    write (*,*) 'nerror: ',string
    STOP 'program terminated by nrerror'
END SUBROUTINE nrerror

FUNCTION arth_r(first,increment,n)
    REAL(SP), INTENT(IN) :: first,increment
    INTEGER(I4B), INTENT(IN) :: n
    REAL(SP), DIMENSION(n) :: arth_r
    INTEGER(I4B) :: k,k2
    REAL(SP) :: temp
    if (n > 0) arth_r(1)=first
    if (n <= NPAR_ARTH) then
        do k=2,n
            arth_r(k)=arth_r(k-1)+increment
        end do
    else
        do k=2,NPAR2_ARTH
            arth_r(k)=arth_r(k-1)+increment
        end do
        temp=increment*NPAR2_ARTH
        k=NPAR2_ARTH
        do
            if (k >= n) exit
            k2=k+k
            arth_r(k+1:min(k2,n))=temp+arth_r(1:min(k,n-k))
            temp=temp+temp
            k=k2
        end do
    end if
END FUNCTION arth_r

FUNCTION arth_d(first,increment,n)
    REAL(DP), INTENT(IN) :: first,increment
    INTEGER(I4B), INTENT(IN) :: n
    REAL(DP), DIMENSION(n) :: arth_d
    INTEGER(I4B) :: k,k2
    REAL(DP) :: temp

```

```

if (n > 0) arth_d(1)=first
if (n <= NPAR_ARTH) then
  do k=2,n
    arth_d(k)=arth_d(k-1)+increment
  end do
else
  do k=2,NPAR2_ARTH
    arth_d(k)=arth_d(k-1)+increment
  end do
  temp=increment*NPAR2_ARTH
  k=NPAR2_ARTH
  do
    if (k >= n) exit
    k2=k+k
    arth_d(k+1:min(k2,n))=temp+arth_d(1:min(k,n-k))
    temp=temp+temp
    k=k2
  end do
end if
END FUNCTION arth_d

FUNCTION arth_i(first,increment,n)
INTEGER(I4B), INTENT(IN) :: first,increment,n
INTEGER(I4B), DIMENSION(n) :: arth_i
INTEGER(I4B) :: k,k2,temp
if (n > 0) arth_i(1)=first
if (n <= NPAR_ARTH) then
  do k=2,n
    arth_i(k)=arth_i(k-1)+increment
  end do
else
  do k=2,NPAR2_ARTH
    arth_i(k)=arth_i(k-1)+increment
  end do
  temp=increment*NPAR2_ARTH
  k=NPAR2_ARTH
  do
    if (k >= n) exit
    k2=k+k
    arth_i(k+1:min(k2,n))=temp+arth_i(1:min(k,n-k))
    temp=temp+temp
    k=k2
  end do
end if
END FUNCTION arth_i ! .... and many other FUNCTIONS and SUBROUTINES ....
END MODULE nrutil

```