Homework 2, Problem 1 (solutions)

Theory/Introduction

In this problem will compare integration using the trapezoid rule, Simpsons's rule, and Gaussian (Gauss-Legendre) quadrature. The function integrated is either cos(x) or exp(-x^2).

For $\cos^2$(x) we can easily compute the exact error; for exp(-x^2) there is no analytic result for the integral, so instead of plotting the error, we plot the normalized difference between the integral with N intervals and 2*N intervals. Plotting the error vs. N, on a log-log plot should show two regions:

1. Decreasing error, with slope of line equal to the power of the approximation error of the algorithm.
2. Increasing error due to round-off. This should have a slope of 1/2 if the round-off error adds randomly.

The method used was standard trapezoid and Simpson's integration rules. For the Gauss-Legendre integration, I used the subroutine supplied by the text to generate the weights and abcissas.

Code

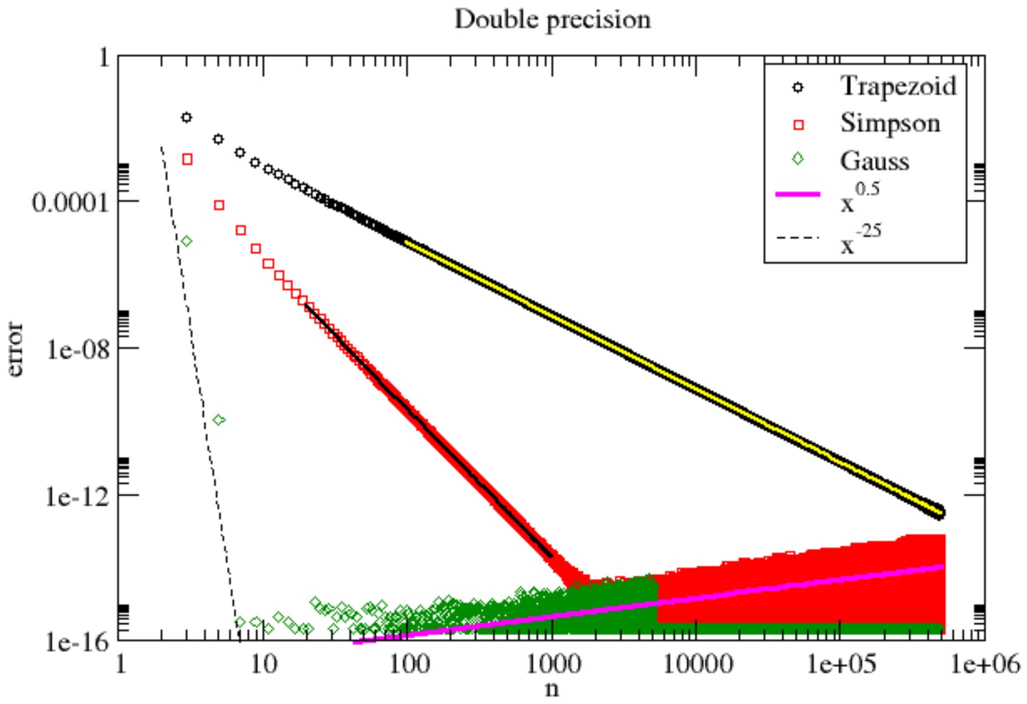
These are only slightly modified from the book sample program.

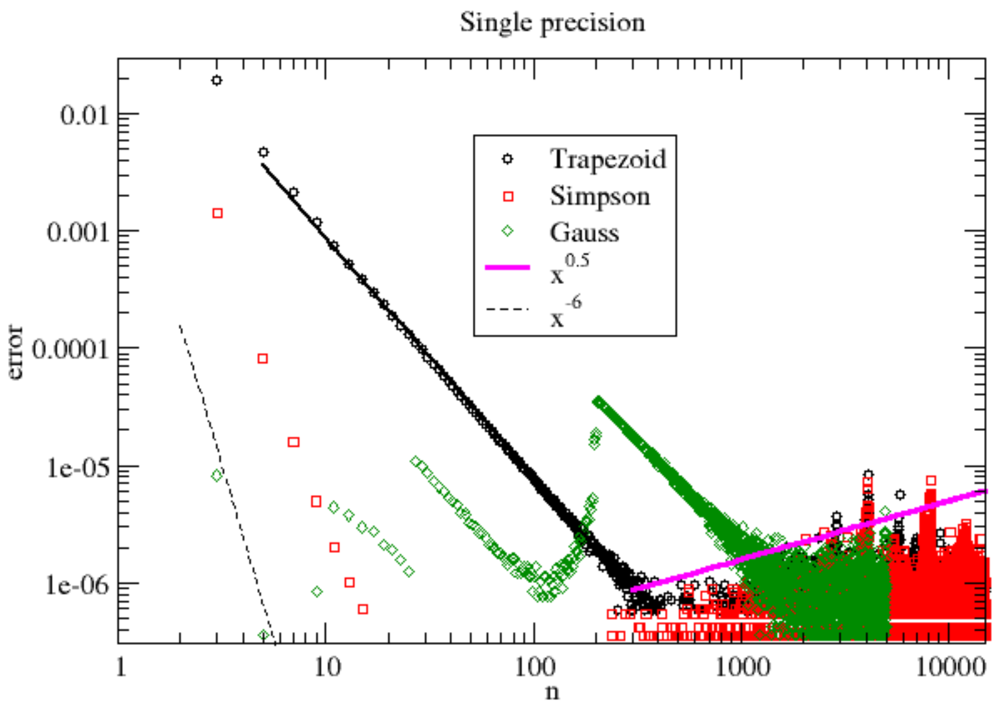
1. integ-4433-double.f95
2. integ-4433-single.f95
3. integ-6433-double.f95
4. integ-6433-single.f95

Results

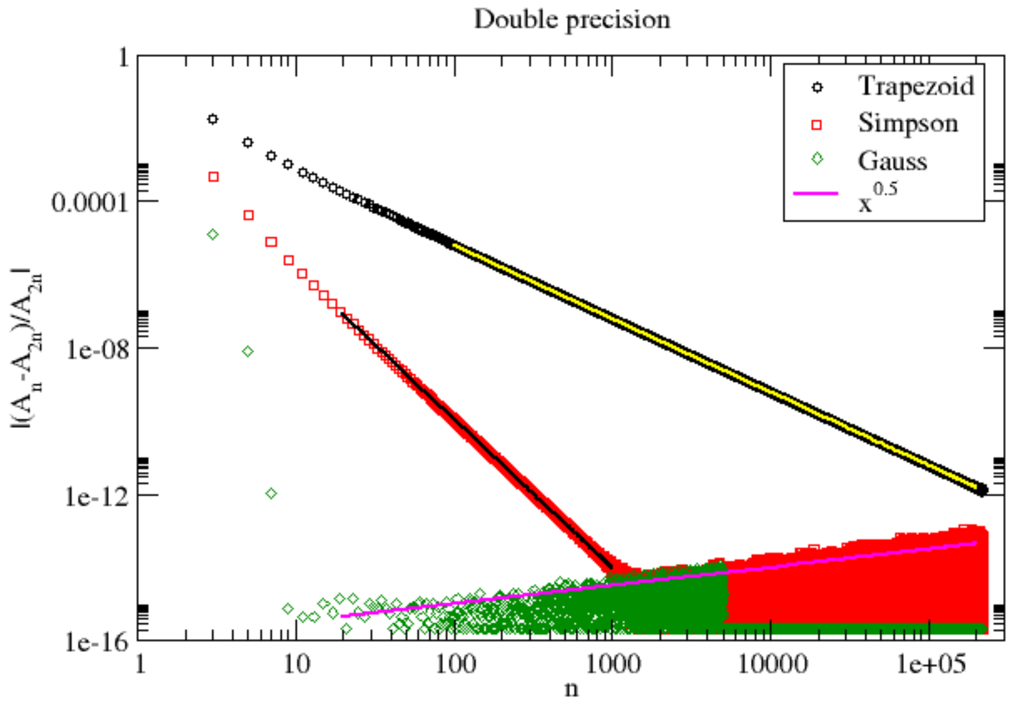
For these plots, I fit the data to a power law for the trapezoid and Simpson's methods. For large N I plotted a curve proportional to $x^{0.5}$.

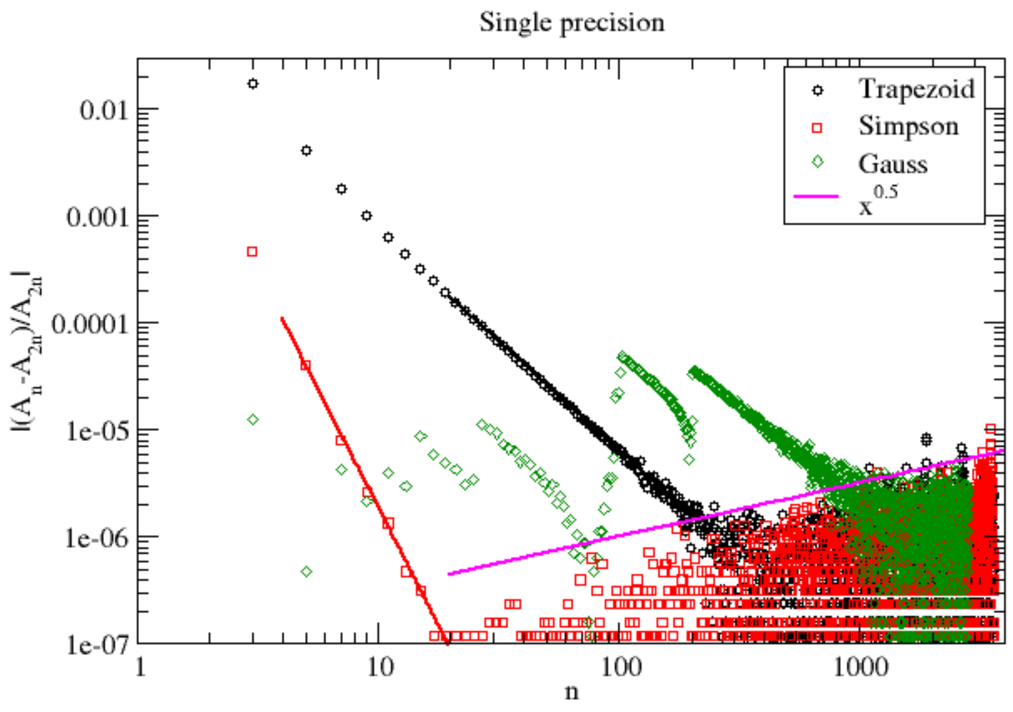
$\cos^2$(x)


Double precision
error
n
Trapezoid
Simpson
Gauss
$x^{0.5}$
x^{-25}
1
0.0001
1e-08
1e-12
1e-16
1
10
100
1000
10000
1e+05
1e+06


Single precision
error
n
Trapezoid
Simpson
Gauss
$x^{0.5}$
x^{-6}
0.01
0.001
0.0001
1e-05
1e-06
1
10
100
1000
10000

$\exp(-x^2)$


Double precision
Trapezoid
Simpson
Gauss
$x^{0.5}$
$|(A_n-A_{2n})/A_{2n}|$
n
1
0.0001
1e-08
1e-12
1e-16
1
10
100
1000
10000
1e+05



Discussion

In some cases the relative error underflows to exactly zero, which cannot be plotted on a log scale. In the code I check for a very small result, and in this case set the value to the machine precision returned by the function epsilon().

To save time I did not compute the Gaussian quadrature for very large N.

It is difficult to fit the roundoff part of the results. Rather than doing a fit, I plotted the function $x^{0.5}$ for large N on the plots. The growth of the roundoff error is consistent with this.

Single Precision

Gaussian quadrature converged by far the fastest, with a minimum error close to machine precision (about 1e-6). In calculating the weights for the Gaussian quadrature there is a tolerance, which I set to 3.0e-06. Possibly by trying different values of this tolerance slightly better Gauss performance could be realized. The minimum error for Simpson's was also close to machine precision. The minimum error for the trapezoid algorithm was slightly larger.

Fitting lines to the initial decay of the error gave these slopes (these are for the $\cos^2(x)$ problem, slopes for $\exp(-x^2)$ were similar)

- Trapezoid: slope=-2.04(3) Optimum value of N (least error) about N=300. In class we predicted 600. Minimum error about 6e-07, similar to our prediction.
- Simpson: slope= -4.3(1) Optimum value of N about N=15; predicted optimum about N=40.
- Gauss: not enough points to fit. There is a very steep decrease of error with N, approximately like x^{-6}. Optimum value of N less than 10. For larger N, the dependence of error on N is quite complicated. Somewhere above 1000 the error seemed to start to follow the $x^{0.5}$ function. The unusual error variation with N depended on the setting of the precision eps in the subroutine calculating the Gauss quadrature weights and abcissas.

Double Precision

Again, Gaussian quadrature converged by far the fastest, with a minimum error close to machine precision (about 1e-16). An N of 500,000 was not high enough for the trapezoid error to be dominated by round off. Our prediction for the optimum N was 1000000, so this was consistent.

Fitting lines to the initial decay of the error gave these slopes:

- Trapezoid: slope=-2.0019(3), no roundoff visible. Optimum N larger than 500000.
- Simpson: slope=-4.032(2) Optimum N about 1100; predicted optimum 2200.
- Gauss: not enough points to fit, eye estimate gives a slope power law of about -25.

Summary

For both single and double precision, the slopes are similar to those predicted: -2 for the trapezoid rule approximation error, and -4 for the Simpson rule error. The Gauss error was quite small, small enough that it was not possible to get a good estimate for the slope. The roundoff error slopes were close to 0.5, although in one case (Gaussian quadrature, single precesion) there was quite complicated error for intermediate n.

```
! integrate.f90: Integrate exp(-x) using trapezoid, Simpson and Gauss rules
!
! From: "A SURVEY OF COMPUTATIONAL PHYSICS"
!       by RH Landau, MJ Paez, and CC BORDEIANU
!       Copyright Princeton University Press, Princeton, 2008.
!       Electronic Materials copyright: R Landau, Oregon State Univ, 2008;
!       MJ Paez, Univ Antioquia, 2008; and CC BORDEIANU, Univ Bucharest, 2008.
!       Supported by the US National Science Foundation
!
! code cleaned up and rewritten in modern fortran style by RT Clay, 2013
!
program integrate
  implicit none

  real(kind=4) :: r1, r2, r3
  real(kind=4) :: theo, vmin, vmax, eps
  integer :: i

  !
  ! theoretical result, integration range
  !
  theo =  0.25*(2.00+sin(2.0))
  vmin=0.0
  vmax=1.0

  eps=epsilon(r1)

  open(10, File='integ-4433-single.dat', Status='Unknown')

  do i= 3, 15001 , 2
     r1=trapez(i, vmin, vmax)
     r1=abs(r1-theo)
     if (r1<eps) r1=eps

     r2=simpson(i,vmin, vmax)
     r2=abs(r2-theo)
     if (r2<eps) r2=eps

     ! only do gauss quad for smaller N
     if (i<5000) then
        r3=quad(i,vmin, vmax)
        r3=abs(r3-theo)
        if (r3<eps) r3=eps
     else
        r3=eps
     endif
      write(10,*) i, r1, r2, r3
   end do
   close(10)

contains

!
! the function we want to integrate
!
  function f(x)
    real(kind=4) :: f, x

    f=cos(x)*cos(x)

  end function f

!
! trapezoid rule
!
  function trapez(i, min, max)
    integer :: i, n
    real(kind=4) :: interval, min, max, trapez, x
    trapez=0
    interval = ((max-min) / (i-1))

    ! sum the midpoints
    do  n=2, (i-1)
       x = interval * (n-1)
       trapez = trapez + f(x)*interval
    end do

    ! add the endpoints
    trapez = trapez+0.5*(f(min)+f(max))*interval

  end function trapez

!
! Simpson's rule
!
  function simpson(i, min, max)
    integer :: i, n
    real(kind=4) :: interval, min, max, simpson, x

    simpson=0.0
    interval = ((max-min) / (i-1))

    ! loop for odd points
    do  n=2, (i-1), 2
       x = interval * (n-1)
       simpson = simpson + 4.0*f(x)
    end do
    ! loop for even points
    do  n=3, (i-1), 2
       x = interval * (n-1)
       simpson = simpson + 2.0*f(x)
    end do
    ! add the endpoints
    simpson = simpson+f(min)+f(max)
    simpson=simpson*interval/3.0

  end function simpson

!
! Gauss' rule
!
  function quad(n, min, max)
    integer :: n
    real(kind=4),dimension(n) :: w,x  ! note use of automatic arrays
    real(kind=4) ::  min, max, quad

    integer :: i, job

    quad=0.0
    job=0.0
    call gauss(n, job, min, max, x, w)
    do  i=1, n
       quad=quad+f(x(i))*w(i)
    end do

  end function quad

!
!gauss.f90: Points and weights for Gaussian quadrature

!       rescale rescales the gauss-legendre grid points and weights
!
!       npts             number of points
!       job =   0        rescaling uniformly between (a,b)
!               1        for integral (0,b) with 50% points inside (0, ab/(a+b))
!               2        for integral (a,inf) with 50% inside (a,b+2a)
!               x, w             output grid points and weights.
!
  subroutine gauss(npts,job,a,b,x,w)
    integer,intent(in) ::npts,job
    real(kind=4),intent(in) :: a,b
    real(kind=4),intent(out) :: x(npts),w(npts)

    real(kind=4) :: xi,t,t1,pp,p1,p2,p3,aj
    real(kind=4),parameter :: pi=3.14159265358979323846264338328
    real(kind=4),parameter :: eps=3.0e-06
    real(kind=4),parameter :: zero=0.0, one=1.0, two=2.0
    real(kind=4),parameter :: half=0.5, quarter=0.25
    integer :: m,i,j

    m=(npts+1)/2
    do  i=1,m
       t=cos(pi*(i-quarter)/(npts+half))
       do
          p1=one
          p2=zero
          aj=zero
          do j=1,npts
             p3=p2
             p2=p1
             aj=aj+one
             p1=((two*aj-one)*t*p2-(aj-one)*p3)/aj
          end do
          pp=npts*(t*p1-p2)/(t*t-one)
          t1=t
          t=t1-p1/pp
          if (abs(t-t1)<eps) exit
       enddo
       x(i)=-t
       x(npts+1-i)=t
       w(i)=two/((one-t*t)*pp*pp)
       w(npts+1-i)=w(i)
    end do

    !
    ! rescale the grid points
    !
    select case(job)
    case (0)
       ! scale to (a,b) uniformly
       do i=1,npts
          x(i)=x(i)*(b-a)/two+(b+a)/two
          w(i)=w(i)*(b-a)/two
       end do

    case(1)
       ! scale to (0,b) with 50% points inside (0,ab/(a+b))
       do i=1,npts
          xi=x(i)
          x(i)=a*b*(one+xi)/(b+a-(b-a)*xi)
          w(i)=w(i)*two*a*b*b/((b+a-(b-a)*xi)*(b+a-(b-a)*xi))
       end do

    case(2)
       ! scale to (a,inf) with 50% points inside (a,b+2a)
       do i=1,npts
          xi=x(i)
          x(i)=(b*xi+b+a+a)/(one-xi)
          w(i)=w(i)*two*(a+b)/((one-xi)*(one-xi))
       end do
    end select

  end subroutine gauss


end program integrate
```

```fortran
! integrate.f90: Integrate exp(-x) using trapezoid, Simpson and Gauss rules
!
! From: "A SURVEY OF COMPUTATIONAL PHYSICS"
!       by RH Landau, MJ Paez, and CC BORDEIANU
!       Copyright Princeton University Press, Princeton, 2008.
!       Electronic Materials copyright: R Landau, Oregon State Univ, 2008;
!       MJ Paez, Univ Antioquia, 2008; and CC BORDEIANU, Univ Bucharest, 2008.
!       Supported by the US National Science Foundation
!
! code cleaned up and rewritten in modern fortran style by RT Clay, 2015
!
program integrate
  implicit none

  real(kind=8) :: r1, r2, r3, r12, r22, r32, eps
  real(kind=8) :: vmin, vmax
  integer :: i

  open(10, File='integ-6433-double.dat', Status='Unknown')

  vmin=0.0d0
  vmax=1.0d0
  eps=epsilon(r1)

  ! calculate integral using both methods for steps = 3..501
  do i= 3, 3000001 , 2
     r1=trapez(i, vmin, vmax)
     r12=trapez(i*2, vmin, vmax)
     r1=abs((r1-r12)/r12)
     if (r1<eps) r1=eps

     r2=simpson(i,vmin, vmax)
     r22=simpson(i*2+1,vmin, vmax)
     r2=abs((r2-r22)/r22)
     if (r2<eps) r2=eps

     ! only do gauss quad for smaller N
     if (i<5000) then
        r3=quad(i,vmin, vmax)
        r32=quad(i*2,vmin, vmax)
        r3=abs((r3-r32)/r32)
        if (r3<eps) r3=eps
     else
        r3=eps
     endif
     write(10,*) i, r1, r2, r3
  end do
  close(10)

contains

!
! the function we want to integrate
!
  function f(x)
    real(kind=8) :: f, x

    f=exp(-x*x)

  end function f

!
! trapezoid rule
!
  function trapez(i, min, max)
    integer :: i, n
    real(kind=8) :: interval, min, max, trapez, x
    trapez=0
    interval = ((max-min) / (i-1))

    ! sum the midpoints
    do  n=2, (i-1)
       x = interval * (n-1)
       trapez = trapez + f(x)*interval
    end do

    ! add the endpoints
    trapez = trapez+0.5_8*(f(min)+f(max))*interval

  end function trapez

!
! Simpson's rule
!
  function simpson(i, min, max)
    integer :: i, n
    real(kind=8) :: interval, min, max, simpson, x

    simpson=0.0_8
    interval = ((max-min) / (i-1))

    ! loop for odd points
    do  n=2, (i-1), 2
       x = interval * (n-1)
       simpson = simpson + 4.0_8*f(x)
    end do
    ! loop for even points
    do  n=3, (i-1), 2
       x = interval * (n-1)
       simpson = simpson + 2.0_8*f(x)
    end do
    ! add the endpoints
    simpson = simpson+f(min)+f(max)
    simpson=simpson*interval/3.0_8

  end function simpson

!
! Gauss' rule
!
  function quad(n, min, max)
    integer :: n
    real(kind=8),dimension(n) :: w,x  ! note use of automatic arrays
    real(kind=8) ::  min, max, quad

    integer :: i, job

    quad=0.0_8
    job=0.0_8
    call gauss(n, job, min, max, x, w)
    do  i=1, n
       quad=quad+f(x(i))*w(i)
    end do

  end function quad

!
!gauss.f90: Points and weights for Gaussian quadrature

!       rescale rescales the gauss-legendre grid points and weights
!
!       npts             number of points
!       job =   0        rescaling uniformly between (a,b)
!               1        for integral (0,b) with 50% points inside (0, ab/(a+b))
!               2        for integral (a,inf) with 50% inside (a,b+2a)
!               x, w             output grid points and weights.
!
  subroutine gauss(npts,job,a,b,x,w)
    integer,intent(in) ::npts,job
    real(kind=8),intent(in) :: a,b
    real(kind=8),intent(out) :: x(npts),w(npts)

    real(kind=8) :: xi,t,t1,pp,p1,p2,p3,aj
    real(kind=8),parameter :: pi=3.14159265358979323846264338328_8
    real(kind=8),parameter :: eps=3.0e-16_8
    real(kind=8),parameter :: zero=0.0_8, one=1.0_8, two=2.0_8
    real(kind=8),parameter :: half=0.5_8, quarter=0.25_8
    integer :: m,i,j
    m=(npts+1)/2
    do  i=1,m
       t=cos(pi*(i-quarter)/(npts+half))
       do
          p1=one
          p2=zero
          aj=zero
          do j=1,npts
             p3=p2
             p2=p1
             aj=aj+one
             p1=((two*aj-one)*t*p2-(aj-one)*p3)/aj
          end do
          pp=npts*(t*p1-p2)/(t*t-one)
          t1=t
          t=t1-p1/pp
          if (abs(t-t1)<eps) exit
       enddo
       x(i)=-t
       x(npts+1-i)=t
       w(i)=two/((one-t*t)*pp*pp)
       w(npts+1-i)=w(i)
    end do

    !
    ! rescale the grid points
    !
    select case(job)
    case (0)
       ! scale to (a,b) uniformly
       do i=1,npts
          x(i)=x(i)*(b-a)/two+(b+a)/two
          w(i)=w(i)*(b-a)/two
       end do

    case(1)
       ! scale to (0,b) with 50% points inside (0,ab/(a+b))
       do i=1,npts
          xi=x(i)
          x(i)=a*b*(one+xi)/(b+a-(b-a)*xi)
          w(i)=w(i)*two*a*b*b/((b+a-(b-a)*xi)*(b+a-(b-a)*xi))
       end do

    case(2)
       ! scale to (a,inf) with 50% points inside (a,b+2a)
       do i=1,npts
          xi=x(i)
          x(i)=(b*xi+b+a+a)/(one-xi)
          w(i)=w(i)*two*(a+b)/((one-xi)*(one-xi))
       end do
    end select

  end subroutine gauss


end program integrate
```