Outline:

- Adaptive quadrature, recursive functions
- Load balancing with OpenMP
- nested forking

Code:

- $UWHPSC/codes/adaptive_quadrature
Adaptive quadrature

Problem: Approximate

\[ \int_{-2}^{4} e^{-\beta^2 x^2} + \sin(x) \, dx = \left[ \frac{\sqrt{\pi}}{2\beta} \text{erf}(\beta x) - \cos(x) \right]_{-2}^{4} \]

where \( \text{erf} \) is the error function.

\( \beta = 10: \)
Adaptive quadrature

Idea: Subdivide into subintervals and apply Trapezoid or Simpson’s Rule on each.

Use larger intervals where \( f(x) \) is smoother. Automate!
Adaptive quadrature

Ideas:

\[ \int_{a}^{b} f(x) \, dx = \int_{a}^{(a+b)/2} f(x) \, dx + \int_{(a+b)/2}^{b} f(x) \, dx. \]
Adaptive quadrature

Ideas:

- \[ \int_{a}^{b} f(x) \, dx = \int_{a}^{(a+b)/2} f(x) \, dx + \int_{(a+b)/2}^{b} f(x) \, dx. \]

- If we split the interval in half and the error on each half is less than \( \text{tol}/2 \) then the total error is less than \( \text{tol} \).
Adaptive quadrature

Ideas:

1. $\int_{a}^{b} f(x) \, dx = \int_{a}^{(a+b)/2} f(x) \, dx + \int_{(a+b)/2}^{b} f(x) \, dx.$

2. If we split the interval in half and the error on each half is less than $tol/2$ then the total error is less than $tol$.

3. Simpson’s Rule is much more accurate than Trapezoid so the difference between the two is a good estimate of the error in Trapezoid.
Adaptive quadrature

Ideas:

\[ \int_a^b f(x) \, dx = \int_a^{(a+b)/2} f(x) \, dx + \int_{(a+b)/2}^b f(x) \, dx. \]

• If we split the interval in half and the error on each half is less than \( \text{tol}/2 \) then the total error is less than \( \text{tol} \).

• Simpson’s Rule is much more accurate than Trapezoid so the difference between the two is a good estimate of the error in Trapezoid.

• If the error estimate on either half is greater than \( \text{tol}/2 \), then recursively subdivide that interval in half.
Recursive subroutine example

Compute $m!$ recursively,

Using $m! = m(m-1)(m-2) \cdots 3 \cdot 2 \cdot 1 = m \ (m-1)!$

```fortran
recursive subroutine myfactorial(m,mfact)
    implicit none
    integer, intent(in) :: m
    integer, intent(out) :: mfact
    integer :: m1fact

    if (m <= 1) then
        mfact = 1
    else
        call myfactorial(m-1, m1fact)
        mfact = m * m1fact
    endif
end subroutine myfactorial
```

$UWHPSC/adaptive_quadtrature/factorial_example.f90$

R.J. LeVeque, University of Washington  AMath 483/583, Lecture 20
Adaptive quadrature

Ideas:

• \[ \int_a^b f(x) \, dx = \int_a^{(a+b)/2} f(x) \, dx + \int_{(a+b)/2}^b f(x) \, dx. \]

• If we split the interval in half and the error on each half is less than \( \text{tol}/2 \) then the total error is less than \( \text{tol} \).

• Simpson’s Rule is much more accurate than Trapezoid so the difference between the two is a good estimate of the error in Trapezoid.

• If the error estimate on either half is greater than \( \text{tol}/2 \), then recursively subdivide that interval in half.
Adaptive Quadrature

See codes in $UWHPSC/codes/adaptive_quadrature

..serial: Serial code with recursive subroutine

..openmp1: OpenMP splitting into two pieces

..openmp2: OpenMP with nested forks
Adaptive quadrature — recursion

Selected lines from
$UWHPSC/codes/adaptive_quadrature/serial/adapquad_mod.f90

recursive subroutine adapquad(f,a,b,tol,intest,errest,level,fa,fb)
  implicit none
  real(kind=8), intent(in) :: a,b,tol
  real(kind=8), intent(out) :: intest
  real(kind=8), optional, intent(out) :: errest
  integer, optional, intent(in) :: level
  real(kind=8), optional, intent(in) :: fa,fb
  real(kind=8), external :: f

  ! Local variables:
  real(kind=8) :: xmin, fmid, trapezoid, simpson, errest1, errest2, &
    intest1, intest2, tol2, f_a, f_b
  integer :: thislevel, nextlevel
Using optional subroutine parameters in Fortran 90:

```fortran
if (.not. present(level)) then
    ! called from main program, which is level=1:
    thislevel = 1
else
    thislevel = level
endif

write(8,801) a,b,thislevel

if (present(fa)) then
    f_a = fa
else
    f_a = f(a)
endif
```
Main recursion step:

```fortran
if ((errrest > tol) .and. (thislevel < maxlevel)) then
  ! recursively apply this subroutine to each half, with
  ! tolerance tol/2 for each, and nextlevel = thislevel+1:
  tol2 = tol / 2.d0
  nextlevel = thislevel + 1
  call adapquad(f,a,xmid,tol2,intest1,errrest1,nextlevel,f_a,fmid)
  call adapquad(f,xmid,b,tol2,intest2,errrest2,nextlevel,fmid,f_b)
  intest = intest1 + intest2
  errrest = errrest1 + errrest2
else
  ! Use the trapezoid approximation.
  ! Note that simpson would be better,
  ! but we have error estimate for trapezoid
  intest = trapezoid
endif
```
Adaptive quadrature with $\text{tol} = 0.5$

approx = $0.1982448782099E+00$
true = $0.4147421694070E+00$
error = $-0.216E+00$
errest = $-0.415E-01$
Adaptive quadrature with $\text{tol} = 0.1$

approx = $0.4074167985367E+00$
true = $0.4147421694070E+00$
error = $-0.733E-02$
errest = $-0.730E-02$
g was evaluated 53 times
Adaptive quadrature with $tol = 0.02$

approx = $0.4144742980922E+00$
true = $0.4147421694070E+00$
error = $-0.268E-03$
errest = $0.119E-01$
g was evaluated 115 times

R.J. LeVeque, University of Washington | AMath 483/583, Lecture 20
First attempt: split up original interval into 2 pieces in main program...

```fortran
! $UWHPSC/codes/adaptive_quadrature/openmp1/testquad.f90

xmid = 0.5d0*(a+b)
tol2 = tol / 2.d0

 !$omp parallel sections
 !$omp section
    call adapquad(g,a,xmid,tol2,intest1,errest1)
 !$omp section
    call adapquad(g,xmid,b,tol2,intest2,errest2)
 !$omp end parallel sections

int_approx = intest1 + intest2
errest = errest1 + errest2
```

May exhibit poor load balancing if much more work has to be done in one half than the other.
Adaptive quadrature with $\text{tol} = 0.1$

Two threads, with OpenMP applied at top level only.

Thread 0 works only on left half,
Thread 1 works only on right half

Blue: Thread 0
Red: Thread 1
Adaptive quadrature with $\text{tol} = 0.01$

Two threads, with OpenMP applied at top level only.

Note that Thread 1 is done before Thread 0

Poor load balancing if function is much smoother on one half of interval than the other!
Better approach: Allow nested calls to OpenMP.

! $UWHPSC/codes/adaptive_quadrature/openmp2/testquad.f90

! Allow nested OpenMP threading:
!$ call omp_set_nested(.true.)

call adapquad(g, a, b, tol, int_approx, errest)

!============

! $UWHPSC/codes/adaptive_quadrature/openmp2/adapquad_mod.f90

if ((errest > tol) .and. (thislevel < maxlevel)) then
! recursively apply this subroutine to each half, with
! tolerance tol/2 for each, and nextlevel = thislevel+1:
  tol2 = tol / 2.d0
  nextlevel = thislevel + 1

  !$omp parallel sections
  !$omp section
    call adapquad(f,a,xmid,tol2,intest1,errest1,nextlevel,f_a,fmid)
  !$omp section
    call adapquad(f,xmid,b,tol2,intest2,errest2,nextlevel,fmid,f_b)
  !$omp end parallel sections
Adaptive quadrature with $tol = 0.1$

Two threads, with nested OpenMP calls

Next available thread takes each interval to be handled.

Blue: Thread 0
Red: Thread 1
Adaptive quadrature with $\text{tol} = 0.1$.

Running same thing a second time gives different pattern:

Next available thread takes each interval to be handled.

Blue: Thread 0
Red: Thread 1
Adaptive quadrature with $\text{tol} = 0.01$

Two threads, with nested OpenMP calls

Next available thread takes each interval to be handled.

Blue: Thread 0
Red: Thread 1
Software for adaptive quadrature

Much more sophisticated quadrature routines are available...

**QUADPACK:** Fortran 77


**SciPy:** `scipy.integrate.quad` uses QUADPACK:

In [1]: from scipy import integrate as I
In [2]: beta = 10.
In [3]: f = lambda x: exp(-beta**2 * x**2) + sin(x)
In [4]: I.quad(f, -2., 4.)
Out[4]: (0.4147421694070216, 8.440197311887498e-09)

Returns estimate of integral and of error.
Use `I.quad?` or `I?` to learn more.