ASTR 660 / Class 5

Fortran crash course

Fortran

Fortran (originally short for “Formula Translation”) is one of the oldest programming languages still in common use.

Like C, but unlike Python, it is a **compiled** language. Before code can be run, it has to be converted (**compiled**) into **machine instructions.** A compiled program is called a **binary** or **executable**.

Large programs are built up by combining **libraries** (called **modules** in Fortran) that are compiled individually and **linked** together. Usually compiling and linking are done at the same time.

Functions can also be linked from pre-compiled third-party libraries (the equivalent of `import` in Python).

Fortran

A good resource for learning about Fortran:

https://fortran-lang.org

A tutorial that walks through all the key features of the language:

https://fortran-lang.org/learn/quickstart

A website that allows you to type and run simple Fortran codes online:

https://play.fortran-lang.org/

An extensive guide to "translation" between Fortran and Python:

https://fortran-lang.org/learn/rosetta_stone/

Fortran on CICA

On the CICA cluster, the default Fortran compiler is GNU Fortran (`gfortran`):

```
fomalhaut ~ module load gcc
fomalhaut ~ which gfortran
/cluster/software/gcc/8.3.0/bin/gfortran
fomalhaut ~ gfortran --version
GNU Fortran (GCC) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Write a hello world program:

```
fomalhaut /data/apcooper/astr660 vim hello.f90
```

```
program hello
  ! A hello world program in Fortran.
  print *, "Hello world!"
end program hello
```

```
fomalhaut /data/apcooper/astr660 gfortran -o hello hello.f90
fomalhaut /data/apcooper/astr660 ./hello
 Hello world!
fomalhaut /data/apcooper/astr660
```

Fortran and C

Some important differences between Fortran and C are:

- Fortran has built-in **support for arrays and array operations**, including a **friendly syntax** for writing array operations;

- Fortran has a **more restrictive, higher-level** approach to **memory management**. The user has much less little freedom to manipulate memory directly.

Fortran is a lot less flexible, therefore safe and **easier to write and read**.

Fortran has evolved through many different ‘standards’. You may still find some ancient programs written in Fortran 77, but mostly you will work with Fortran 90.

More recent versions (Fortran 2003/08/…) add support for features found in other modern programming languages (like C++), including better support for object-oriented programming and coroutines.

```
real(8) function circle_area(r)
  implicit none
  !
  ! Computes the area of a circle of radius r
  real :: r

  ! Set PI with the ATAN method
  real(8), parameter :: PI = 4.D0*atan(1.D0)

  circle_area = PI*(r**2.0)

end function circle_area

program compute_circle_area
  implicit none
  !
  ! This is the 'main program'.

  ! An external function
  real(8) :: circle_area
  ! Variables
  real    :: r = 2.0
  real(8) :: area

  ! Call the function
  area = circle_area(r)

  ! Print the result to stdout
  write(*,*)
  write(*,'(a,f6.3,a,g12.4,a)') 'The area of a circle of radius r = ', r, ' is ', area, ' sq. units'
  write(*,*)

end program compute_circle_area
```

Comments with !

```
real(8) function circle_area(r)
  implicit none
  !
  ! Computes the area of a circle of radius r
  real :: r

  ! Set PI with the ATAN method
  real(8), parameter :: PI = 4.D0*atan(1.D0)

  circle_area = PI*(r**2.0)

end function circle_area

program compute_circle_area
  implicit none
  !
  ! This is the 'main program'.

  ! An external function
  real(8) :: circle_area
  ! Variables
  real    :: r = 2.0
  real(8) :: area

  ! Call the function
  area = circle_area(r)

  ! Print the result to stdout
  write(*,*)
  write(*,'(a,f6.3,a,g12.4,a)') 'The area of a circle of radius r = ', r, ' is ', area, ' sq. units'
  write(*,*)

end program compute_circle_area
```

```
real(8) function circle_area(r)
  implicit none
  !
  ! Computes the area of a circle of radius r
  real :: r

  ! Set PI with the ATAN method
  real(8), parameter :: PI = 4.D0*atan(1.D0)

  circle_area = PI*(r**2.0)

end function circle_area
```

Program

```
program compute_circle_area
  implicit none
  !
  ! This is the 'main program'.

  ! An external function
  real(8) :: circle_area
  ! Variables
  real    :: r = 2.0
  real(8) :: area

  ! Call the function
  area = circle_area(r)

  ! Print the result to stdout
  write(*,*)
  write(*,'(a,f6.3,a,g12.4,a)') 'The area of a circle of radius r = ', r, ' is ', area, ' sq. units'
  write(*,*)

end program compute_circle_area
```

```
real(8) function circle_area(r)
  implicit none
  !
  ! Computes the area of a circle of radius r
  real :: r

  ! Set PI with the ATAN method
  real(8), parameter :: PI = 4.D0*atan(1.D0)

  circle_area = PI*(r**2.0)

end function circle_area

program compute_circle_area
  implicit none
  !
  ! This is the 'main program'.
```

Declare variables

```
  ! An external function
  real(8) :: circle_area
  ! Variables
  real    :: r = 2.0
  real(8) :: area
```

Do stuff

```
  ! Call the function
  area = circle_area(r)
```

Output

```
  ! Print the result to stdout
  write(*,*)
  write(*,'(a,f6.3,a,g12.4,a)') 'The area of a circle of radius r = ', r, ' is ', area, ' sq. units'
  write(*,*)

end program compute_circle_area
```

Function

```
real(8) function circle_area(r)
  implicit none
  !
  ! Computes the area of a circle of radius r
  real :: r

  ! Set PI with the ATAN method
  real(8), parameter :: PI = 4.D0*atan(1.D0)

  circle_area = PI*(r**2.0)

end function circle_area
```

Program

```
program compute_circle_area
  implicit none
  !
  ! This is the 'main program'.

  ! An external function
  real(8) :: circle_area
  ! Variables
  real    :: r = 2.0
  real(8) :: area

  ! Call the function
  area = circle_area(r)

  ! Print the result to stdout
  write(*,*)
  write(*,'(a,f6.3,a,g12.4,a)') 'The area of a circle of radius r = ', r, ' is ', area, ' sq. units'
  write(*,*)

end program compute_circle_area
```

Declare the function

Call the function

Type declarations

real
real(8)
real(kind=8)

integer
integer(8)

logical

character

```
real(8) function circle_area(r)
  implicit none
  !
  ! Computes the area of a circle of radius r
  real :: r

  ! Set PI with the ATAN method
  real(8), parameter :: PI = 4.D0*atan(1.D0)

  circle_area = PI*(r**2.0)

end function circle_area

program compute_circle_area
  implicit none
  !
  ! This is the 'main program'.

  ! An external function
  real(8) :: circle_area
  ! Variables
  real    :: r = 2.0
  real(8) :: area

  ! Call the function
  area = circle_area(r)

  ! Print the result to stdout
  write(*,*)
  write(*,'(a,f6.3,a,g12.4,a)') 'The area of a circle of radius r = ', r, ' is ', area, ' sq. units'
  write(*,*)

end program compute_circle_area
```

```
real(8) function circle_area(r)
  implicit none
  !
  ! Computes the area of a circle of radius r
  real :: r

  ! Set PI with the ATAN method
  real(8), parameter :: PI = 4.D0*atan(1.D0)

  circle_area = PI*(r**2.0)

end function circle_area

program compute_circle_area
  implicit none
  !
  ! This is the 'main program'.

  ! An external function
  real(8) :: circle_area
  ! Variables
  real    :: r = 2.0
  real(8) :: area

  ! Call the function
  area = circle_area(r)

  ! Print the result to stdout
  write(*,*)
  write(*,'(a,f6.3,a,g12.4,a)') 'The area of a circle of radius r = ', r, ' is ', area, ' sq. units'
  write(*,*)

end program compute_circle_area
```

Need to define π yourself…

```
real(8) function circle_area(r)
  implicit none
  !
  ! Computes the area of a circle of radius r
  real :: r

  ! Set PI with the ATAN method
  real(8), parameter :: PI = 4.D0*atan(1.D0)

  circle_area = PI*(r**2.0)

end function circle_area

program compute_circle_area
  implicit none
  !
  ! This is the 'main program'.

  ! An external function
  real(8) :: circle_area
  ! Variables
  real    :: r = 2.0
  real(8) :: area

  ! Call the function
  area = circle_area(r)

  ! Print the result to stdout
  write(*,*)
  write(*,'(a,f6.3,a,g12.4,a)') 'The area of a circle of radius r = ', r, ' is ', area, ' sq. units'
  write(*,*)

end program compute_circle_area
```

Format strings

Formatting output

Python

```python
print("%3d" % 5)
print("%03d" % 5)
print("%s" % "text")
print("%15.7f" % 5.5)
print("%23.16e" % -5.5)
```

Fortran

```fortran
print '(i3)', 5
print '(i3.3)', 5
print '(a)', "text"
print '(f15.7)', 5.5_dp
print '(es23.16)', -5.5_dp
```

Python

```
5
005
text
   5.5000000
-5.5000000000000000e+00
```

Fortran

```
5
005
text
   5.5000000
-5.5000000000000000E+00
```

https://fortran-lang.org/learn/rosetta_stone/

Why do we use Python so much?

Python lets you write, in *minutes*, programs that would take *months* to design, implement and debug in a low-level language like C, Fortran or C++. Often you will run these programs only a few times.

If used well, this is productivity boost far outweighs the "slowness" of pure Python.

Python has an simple and powerful `import` mechanism for combining other people's code with your own.

The 'ecosystem' of optimised scientific packages (built mostly on `numpy`, `scipy` and `matplotlib`) is why Python 'won' over many other interpreted languages for science applications. Python is useful for everything.

Python has left Makefiles, shell scripts, gnuplot/sm, gdb, IDL etc. in the past.

Why learn Fortran (and C/C++) in 2023?

Low level compiled languages are still important:

- For production runs on HPC, efficiency in memory and time is essential. Time is money. Every instruction counts. Fortran and C++ account for almost all production HPC code (e.g. the astrophysics workhorses Gadget and FLASH). You may need to work with these codes.

- Standard python is not well suited to parallel computing across multiple machines (*inter-node parallelism* — not a fundamental problem, solutions are available).

- Much highly optimized legacy code is written in C and Fortran. Numpy makes heavy use of wrapped Fortran code, e.g. from BLAS/LAPACK.

- Standard python itself is written in C. Python can be accelerated by mixing in some C or Fortran (and/or using one of several python libraries to produce machine code on the fly).

Why learn any other language?

All programming languages can solve almost all problems, given enough hard work.

The differences between them are mostly **expressive**. They represent different ways of getting the machine to execute your ideas. They are optimised for different applications.

- Fortran was written to help scientists use computers to solve mathematical problems.
- C is very ‘close to the metal’: you spend a lot of time worrying about how data is stored in the computer’s memory. C is now being “superseded” by a language called Rust.
- C++ was developed to make writing object-oriented programs in C much easier.

By learning different languages, you learn about different ways to approach problems. Learning Fortran might help you to write better Python.

Coding in Fortran 90+

Your task for this week is to learn the basics of Fortran 90. Work through the material linked on slide 3. For the homework you should understand most of the following:

- How to define variables of different types (real, double precision, integers , long integers)
- Conditions: `if … then … else` and loops: `do i=1,N … end do`; `do while … end do`
- How to print text and numbers to stdout.
- How to define and use functions.

Next time we will cover: strings, arrays, subroutines, interfaces, modules, IO.

Coding in Fortran 90+

- Fortran types are strict.
- float32 → real, float64 → double, int32 → integer, int64 → long
- Fortran is **not** case-sensitive! FORTRAN ≡ fortran ≡ Fortran etc. See https://fortran-lang.org/learn/best_practices/style_guide/
- Fortran array indices by default start at 1, not zero. This leads to frequent **off by one** errors for those used to C or Python.
- Fortran code blocks are explicitly marked at both ends (for example **if … then … else … endif** or **do while … done**). Whitespace and indentation does not matter, but please use it anyway to write your program clearly and neatly!
- Loop iterations (`do i = start, end`) in Fortran **include the end value**, unlike Python.

Good Fortran habits

- Use `write(*,*)` rather than `print` (this is just my personal preference)
- **Always** use `implicit none` (why?).
- Be careful about **initialising** variables (declaring is separate from assigning a value):
  `integer :: n_steps = 0`
  (this does not always have to be done when you declare the variable — but, why not?)
- If something is a fixed parameter (i.e. a constant), declare it as such:
  `integer, parameter :: n_steps = 0`

Coding in the real world

As well as actually writing code, you will need to compile and link it, and probably also debug it. This is a whole separate learning curve.

However, it is probably **at least or more important** for you to learn this “practical” stuff.

You will often have to deal with compiling and running **other people’s code**. The ability to *quickly* diagnose and solve the problems involved in doing that will also make you more productive.

Coding in the real world

We will also try to cover some of these in the next class:

- Makefiles;
- Compiler optimizations;
- Debugging with gdb;
- Environment variables (dynamic linking).

ASTR 660 / Class 5

Numerical integration

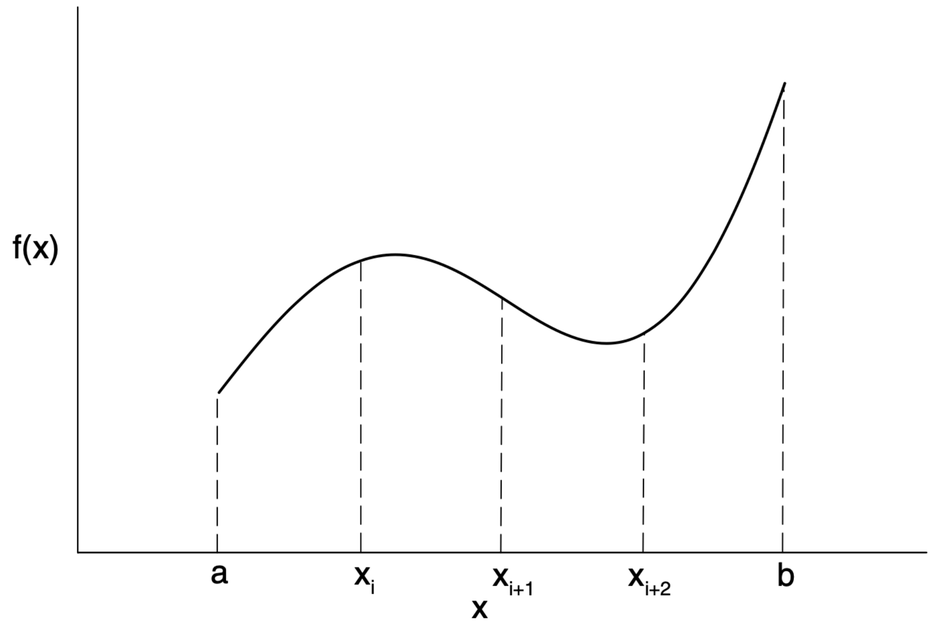
Numerical integration

We have looked at algorithms for solving ODEs, all of which are some variant of Euler's method: extrapolate from what you know and try to control the error in that extrapolation.

This is closely related to the concept of integrating an arbitrary function $f(x)$ numerically. The basic idea is what you learned in calculus 101: the integral is the area under the curve.

For some small step width w_i, we can approximate the area under the curve as a rectangle, and add up all those rectangles to get the total area:

$$\int_a^b f(x)dx \approx \sum_i w_i f(x_i)$$



Numerical integration

$$\int_a^b f(x)dx \approx \sum_i w_i f(x_i)$$

As with the ODE solvers, all we need to decide how to choose the step size w_i. Clearly they can't be "infinitely small" (we have to evaluate the function at each step, which costs time) nor can they be "too big" (inaccurate).

The steps do not all have to be the same.

Once again, the choice is a balance between efficiency, accuracy and stability (working for a wide range of functions and avoiding catastrophic special cases).

Consequently there are several alternatives.

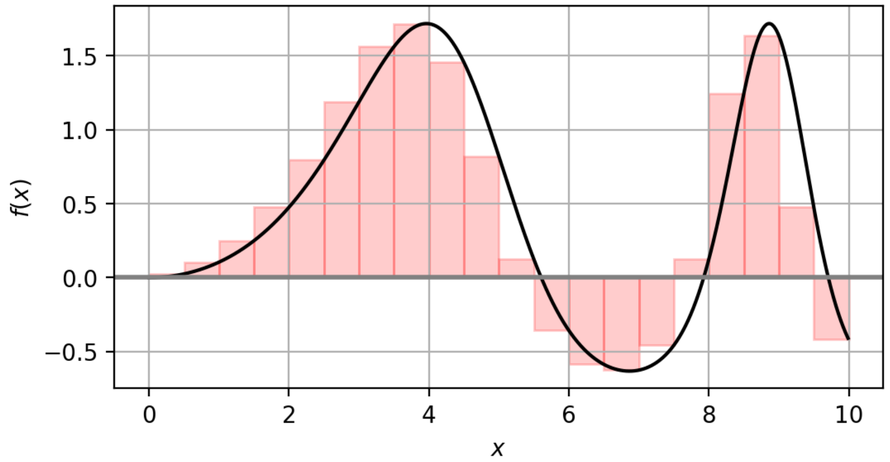
Numerical integration

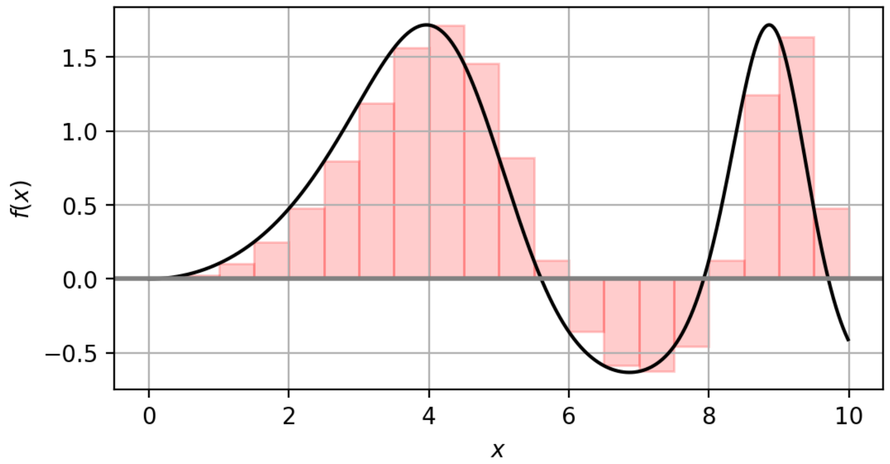
$$\int_a^b f(x)dx \approx \sum_i w_i f(x_i)$$

The most obvious “stability” problem is with **singularities**. Of course, you should not try to integrate over singularities if you know they are in your function — maybe they are not so easy to spot. Maybe they can be removed (by re-writing the function), maybe not.

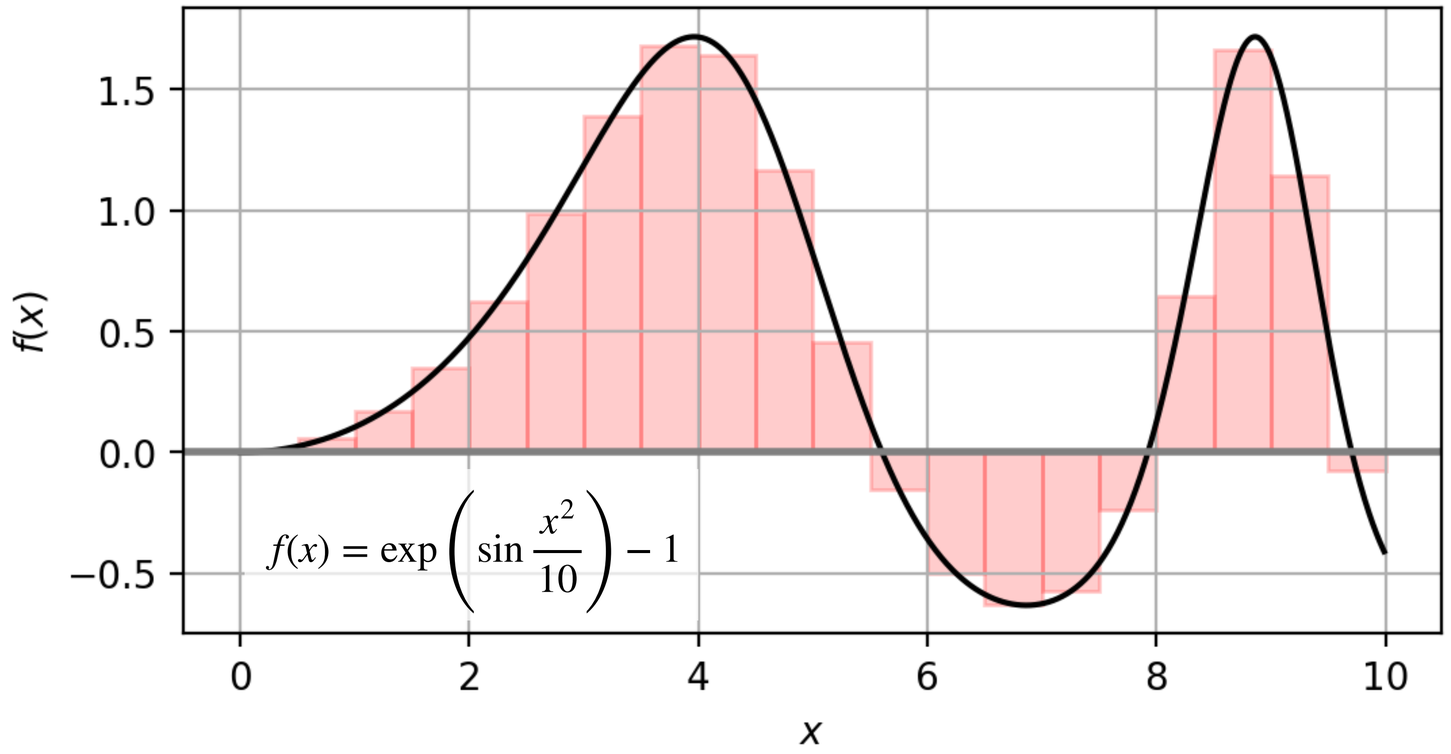
Even if you avoid the actual singularity, how close can you get?

Midpoint



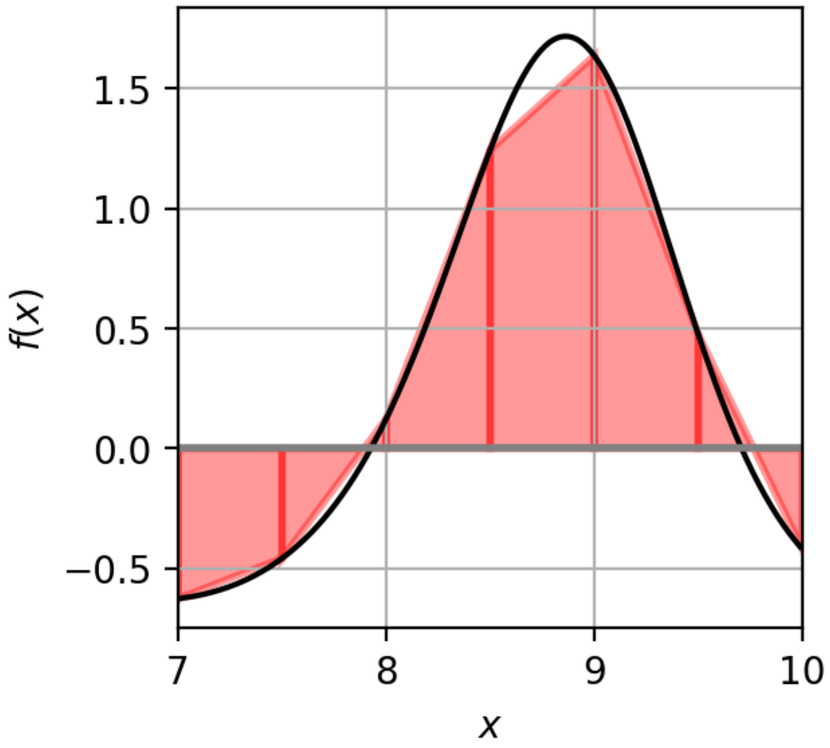


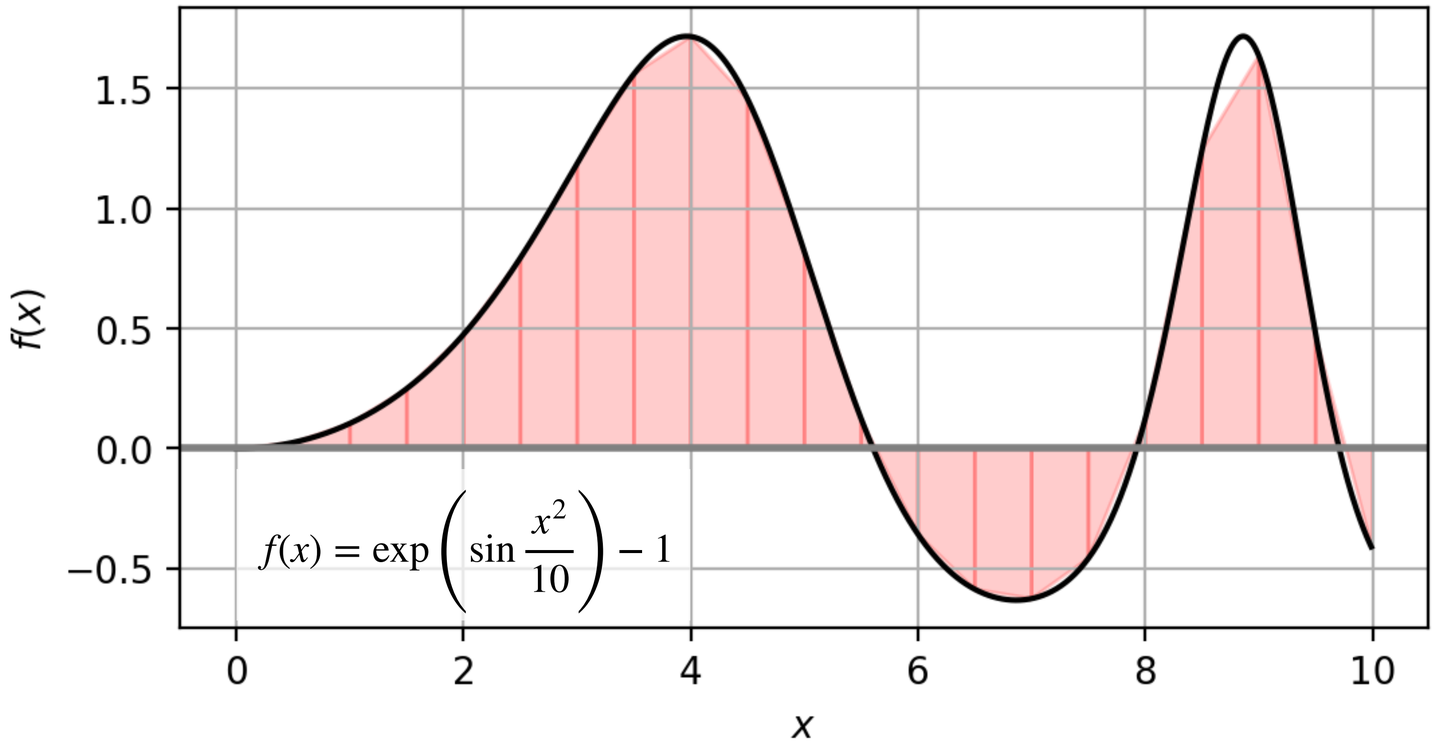
$$\int_a^b f(x)\,dx \approx (a-b)f\left(\frac{a+b}{2}\right)$$



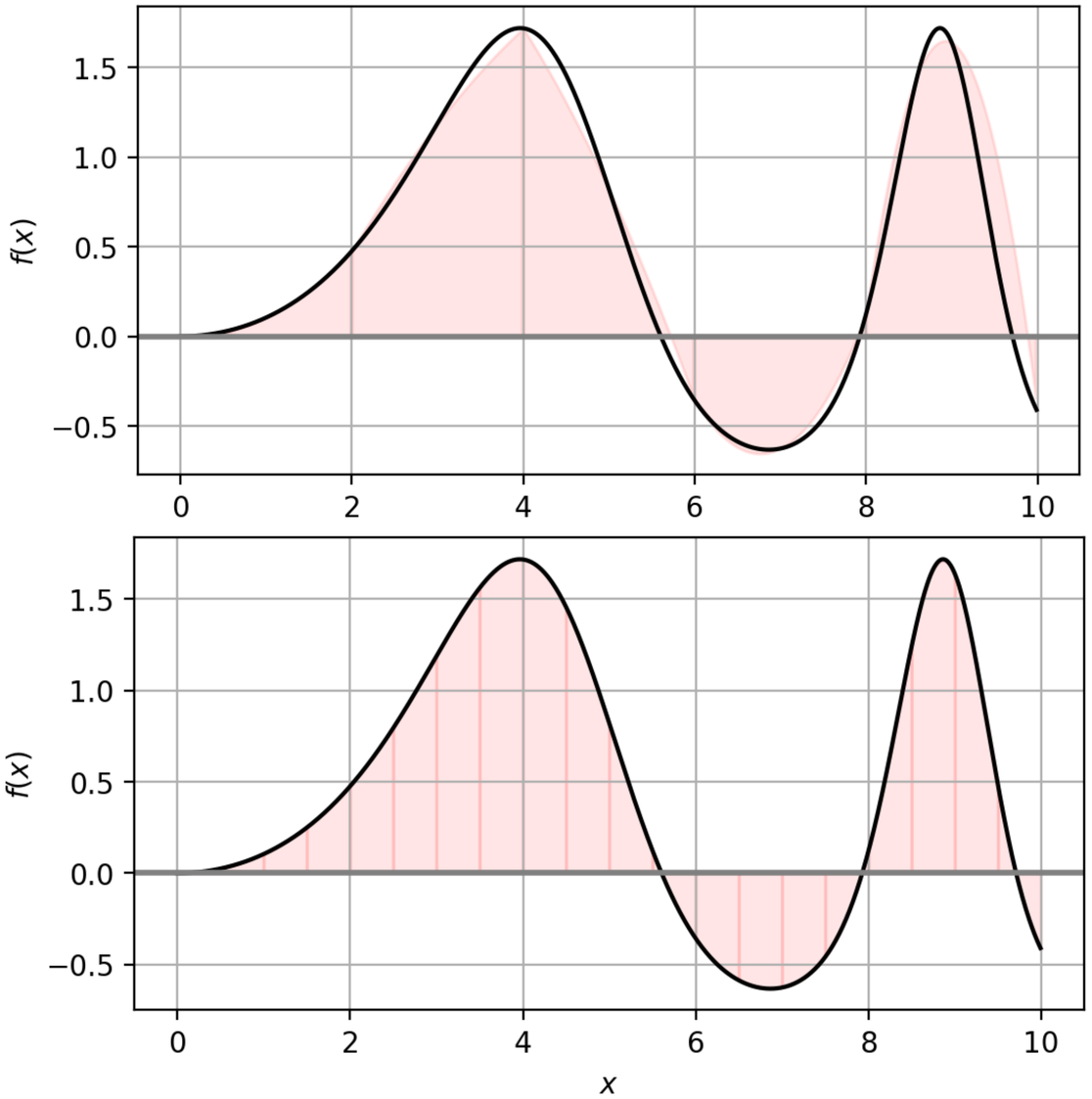
Trapezoid

$$\int_a^b f(x)\,dx \approx (a-b)\frac{f(a)+f(b)}{2}$$





Simpson's (1/3) rule



$$\int_a^b f(x)\,dx \approx \frac{(a-b)}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

Based on the **quadratic interpolating polynomial** of $f(x)$ between a and b.

Can also be derived from an average of the midpoint and trapezoid estimates designed to eliminate the lowest-order error terms.

3 evaluations, but much more accurate. Rapid convergence if higher derivatives are smooth.

See e.g. Wikipedia.

Romberg

All these methods are of the “Newton-Cotes” type (see Wikipedia).

The most generally useful method of this type is Romberg’s method. It uses iterative **extrapolation** of the trapezoid method with different step sizes h to estimate the converged solution as $h \to 0$ (Richardson extrapolation; also used in ODE solvers).

See NR for details.

Quadrature methods

(Quadrature is old mathematical jargon for integration.)

The Newton-Cotes type of method assume fixed step sizes (i.e. a regular grid of points at which to evaluate the function) and hence are prone to instability.

A more robust alternative is to choose the evaluation points x_i adaptively, such that the (appropriately weighted) integral approaches the exact solution if the function is a sufficiently low-order polynomial.

The techniques for finding the right evaluation points are (obviously) not straightforward: you can read about these for yourself if you're interested in the mathematics (again, see NR).

Quadrature methods

The Fortran QUADPACK library contains advanced routines for numerical integration using adaptive quadrature. `scipy.integrate.quad` **wraps functions in QUADPACK.** (In C they are wrapped by the GNU Scientific Library, GSL).

These are useful for a wide range of problems with all kinds of "safety features" to deal with difficult integrands. They are black boxes — you are supposed to know what you're trying to integrate and take appropriate care.

```
>>> help(integrate)
 Methods for Integrating Functions given function object.

   quad           -- General purpose integration.
   dblquad        -- General purpose double integration.
   tplquad        -- General purpose triple integration.
   fixed_quad     -- Integrate func(x) using Gaussian quadrature of order n.
   quadrature     -- Integrate with given tolerance using Gaussian quadrature.
   romberg        -- Integrate func using Romberg integration.

 Methods for Integrating Functions given fixed samples.

   trapezoid             -- Use trapezoidal rule to compute integral.
   cumulative_trapezoid -- Use trapezoidal rule to cumulatively compute integral.
   simpson               -- Use Simpson's rule to compute integral from samples.
   romb                  -- Use Romberg Integration to compute integral from
                         -- (2**k + 1) evenly-spaced samples.

   See the special module's orthogonal polynomials (special) for Gaussian
      quadrature roots and weights for other weighting factors and regions.

 Interface to numerical integrators of ODE systems.

   odeint         -- General integration of ordinary differential equations.
   ode            -- Integrate ODE using VODE and ZVODE routines.
```

Bonus slide: timing python

```
def time_func(f,nloop,ncalls):
    """
    """
    from time import time_ns
    tot_time = 0

    for j in range(0,nloop):
        t0 = time_ns()
        for i in range(0,ncalls):
            f() # Call the function
        t1 = time_ns()
        tot_time += (t1-t0)
    tot_time = tot_time / ncalls / nloop # In ns
    print('Mean for {:d}x{:d} loops: {:6.3f} ms'.format(nloop,ncalls,tot_time/1e6))
```