Introduction to High Performance Scientific Computing

Autumn, 2018

Lecture 7

Notes

- **Panopto:**
  - **uploaded lecture 4, part 2**
  - **Lecture 6 did not record, will provide notes on Neural networks early next week**

- **MLC VMs**
  - **Meeting with ICT today, will post announcement afterwards**
  - **Fine to use Windows/Anaconda for coursework, labs 2 and 3 (but talk to me if you're unsure how to use git)**

Working with Python

- **Can be difficult to develop code within functions (can't easily check the values variables are taking on)**

- **Instead, can initially develop small sections of code in the terminal, or in a script – then have access to all variables used (see Newton's method/sqrt demo on Panopto)**

- **When functions become long or unwieldy, can define new functions (inside or outside the long ones)**

- **If using atom, install the** linter-pylint package **(Preferences → Packages within atom)**
  - **Also useful for Fortran:** language-fortran, linter-gfortran
  - **Linters flag errors as you develop code**

Optimization notes

- **Last time: unconstrained optimization:**

$$\text{Find } \mathbf{x} \text{ so that } f(\mathbf{x}) \text{ is minimized}$$

- **Last time: sketched out basic ideas of classical *unconstrained* optimization**

- **Based on Newton's method:** $H|_{\mathbf{x}_0}\mathbf{h} = -\nabla f|_{\mathbf{x}_0}$

- **Assumes smooth cost function, however BFGS often works well with noisy or non-smooth data**

Optimization notes

- **Last time: unconstrained optimization:**

$$\text{Find } \mathbf{x} \text{ so that } f(\mathbf{x}) \text{ is minimized}$$

- **Last time: sketched out basic ideas of classical *unconstrained* optimization**

- **Based on Newton's method:** $H|_{\mathbf{x}_0}\mathbf{h} = -\nabla f|_{\mathbf{x}_0}$

- **Assumes smooth cost function, however BFGS often works well with noisy or non-smooth data**

- scipy.optimize.minimize **provides access to a number of powerful optimization methods**
  - **Have to provide functions defining cost function, and if feasible, the gradient (and for some methods, the Hessian)**

Optimization notes

- **Lab 3: constrained optimization:**

  Find $\mathbf{x}$ so that $f(\mathbf{x})$ is minimized with $\mathbf{x}$ restricted by:
  equality constraints: $g_1(\mathbf{x}) = 0, g_2(\mathbf{x} = 0), ...$
  and/or inequality constraints: $h_1(\mathbf{x}) \geq 0, h_2(\mathbf{x} \geq 0), ...$

- **Can apply simple bounds on parameters using** L-BFGS **and** TNC **methods**

- **Nonlinear constraints require either** COBYLA **or** SLSQP
  - **Classified as either inequality or equality constraints**
  - **Functions must be provided defining each constraint**

Optimization notes

- **Several methods available which do not use/require derivative:** simulated annealing (basin hopping), Powell, Nelder-Mead, genetic algorithms (differential_evolution)

- **Newton-type methods find *local* minima – simulated annealing, genetic algorithms, and "brute" search for global minima – no guarantee of success!**

Python libraries

- **We have looked at** scipy.optimize **and** scikit-learn
- **However, there is *much, much* more out there**
- **Just in** scipy:
- *scipy.special*
- *scipy.integrate*
- *scipy.fftpack*
- *scipy.signal*

- **Try tab completion:** scipy. <tab>
  import scipy. <tab>

Neural networks

- There are several libraries for neural networks (not all Python-specific)

- Scikit-learn is a good place to start, but other packages typically used for large networks
  - Keras, Torch, Tensorflow, …
  - We will take a quick look at Tensorflow towards the end of term

Python notes

- **We have focused on Python as tool for 'simulations'**

- **Also, very useful for data analysis –**
  - **Can easily read most common data formats into Numpy ndarrays (using e.g.** np.loadtxt)
  - **Many useful commands for exploring/analyzing 2-D arrays**
    - min, max, mean, var, argmin, argmax, … **see numpy reference on course webpage**

  - **For datasets with many variables, should consider Pandas package for data analysis**
    - **Provides mix of database and numpy tools**
    - **Not very intuitive or easy to learn though!**

Fortran intro

- Fortran is a *compiled* language (like C++) designed for scientific computing (like Matlab)

- Fortran has evolved substantially from F66 to F77, F90, Fortran 2008.

- Fortran 77 was the dominant standard, but is now outdated, clumsy.
    - *But,* python, matlab and other software rely on fortran 77 libraries (especially *lapack*)

- Fortran 90 is a powerful, completely modern programming language.

- Typically, F77 codes have *.f* extension, F90 codes use *.f90*

Interpreted vs compiled languages

Determines how code is converted into machine instructions

Interpreter:

- **Goes through code line-by-line, translates into machine language, and executes**

- **Allows for "interactive" programming as in Matlab and Python**

- **However, cannot optimize over blocks of code (*e.g.* a for loop)**

Interpreted vs compiled languages

Determines how code is converted into machine instructions

Interpreter:

- **Goes through code line-by-line, translates into machine language, and executes**
- **Allows for "interactive" programming as in Matlab and Python**
- **However, cannot optimize over blocks of code (*e.g.* a for loop)**

Compiler:

- **Programs stored in file(s) called source code**
- **Compiler analyzes the source code, optimizes where possible, and generates *object* files**
- **A *linker* converts object files into an *executable* file**
- **Interactive programming is not possible, but code may run much faster.**

Basic code structure

```
! Basic Fortran 90 code structure

!1. Header
program template

    !2. Variable declarations (e.g. integers, real numbers,...)

    !3. basic code: input, loops, if-statements, subroutine calls
    print *, 'template code'


!4. End program
end program template

! To compile this code:
! $ gfortran -o template.exe template.f90
! To run the resulting executable: $ ./template.exe
```

Note: Indentation is optional, but *highly* recommended (makes code readable).

See template.f90

Fortran code example

Compute *sin(i)*, i=1,2,3, …, N

Declare a few variables:

```
!1. Header:
Program example1

    !2. Variable declarations:
    implicit none !means all variables in code must be declared
    integer :: i1,j1,N
    real(kind=8) :: var1, var2
    real(kind=8), dimension(10) :: array1
```

Fortran code example

Compute *sin(i)*, i=1,2,3, …, N

Read data:

```
!3. basic code: input, loops, if-statements, subroutine call

    !read data from data.in
    open(unit=10, file='data.in')
         read(10,*) N
    close(10)
```

Fortran code example

Compute *sin(i)*, i=1,2,3, …, N

Main code:

```
!check that N is smaller than size of array1:
   if (N <= size(array1)) then
```

Fortran code example

Compute *sin(i)*, i=1,2,3, …, N

Main code:

```
!check that N is smaller than size of array1:
   if (N <= size(array1)) then

      !compute sin(x) where x = 1,2,3,...,N
      do i1 = 1,N !loop from 1 to N
         var1 = dble(i1) !convert integer to double-prec number
         array1(i1) = sin(var1)
      end do
```

Fortran code example

Compute *sin(i)*, i=1,2,3, …, N

Main code:

```
!check that N is smaller than size of array1:
   if (N <= size(array1)) then

      !compute sin(x) where x = 1,2,3,...,N
      do i1 = 1,N !loop from 1 to N
         var1 = dble(i1) !convert integer to double-prec number
         array1(i1) = sin(var1)
      end do

      !print 1st N elements of array
      print *, 'array1=',array1(1:N)
   else
      print *, 'N must be smaller than', size(array1)
       STOP
   end if
```

Fortran code example

Compute *sin(i)*, i=1,2,3, …, N

Compile and run:

```
$ gfortran —o example1.exe example1.f90

$ ./example1.exe
 array1=  0.84147098480789650       0.90929742682568171
0.14112000805986721      -0.75680249530792820      -0.95892427466313845
```

Fortran code example

‘Cleaner’ code: move loop to a subroutine:

3. Main code:

```
!check that N is smaller than size of array1:
   if (N <= size(array1)) then
      !compute sin(x) where x = 1,2,3,...,N
       call calculations(N,array1)

      !print 1st N elements of array
      print *, 'array1=',array1(1:N)
   else
      print *, 'N must be smaller than', size(array1)
   end if
```

Need subroutine, *calculations*, which take N as input and returns array1

Fortran code example

```
!----------------------
!subroutine calculations
!----------------------
subroutine calculations(N,array)
    implicit none
    integer, intent(in) :: N
    real(kind=8), dimension(10), intent(out) :: array
    integer :: i1
    real(kind=8) :: var1

    do i1 = 1,N !loop from 1 to N
          var1 = dble(i1) !convert integer to real number
          array(i1) = sin(var1)
    end do
end subroutine calculations
```

See f90example2.f90

Floating point numbers in Fortran

- **Single precision: 7 significant figures (4 bytes), not often used**

```
!single precision
real :: var1
real(kind=4) :: var2
real*4 :: var3
```

- **Double precision: 15 significant figures (8 bytes), almost always want double precision in scientific computing**

```
!double precision
real(kind=8) :: dvar1
real*8 :: dvar2
double precision :: dvar3
```

- **All three double precision variable declarations are equivalent, but the *real(kind=)* syntax is "more standard" than the others**

Floating point numbers in Fortran

- **Use *dble* to convert integer to ensure double precision**

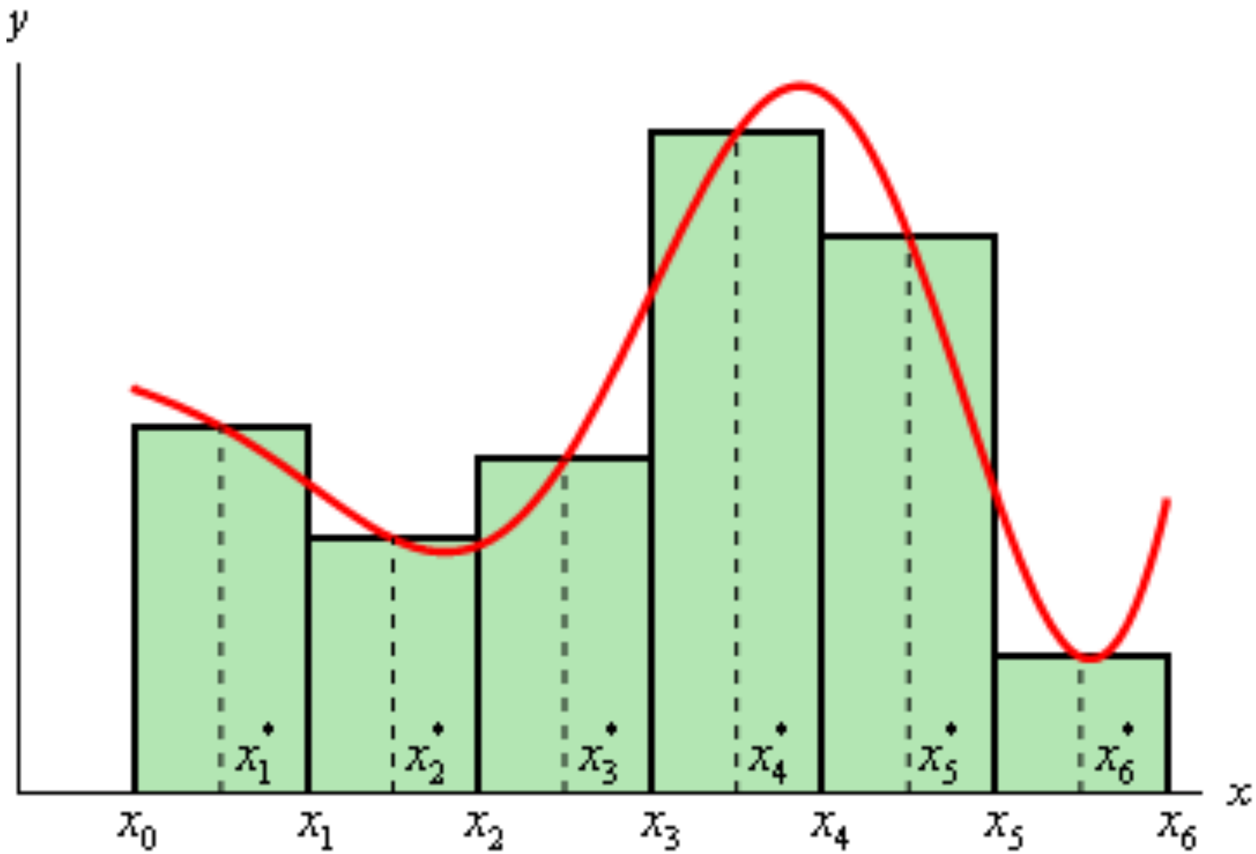
- **Write numbers with "d" after decimal for double precision**
  `2.d0` **or** `3.2d0`

- **Can also include a flag when compiling to force single-precision numbers to be treated as double precision.**

  In gfortran:

```
$ gfortran –freal-4-real-8
```

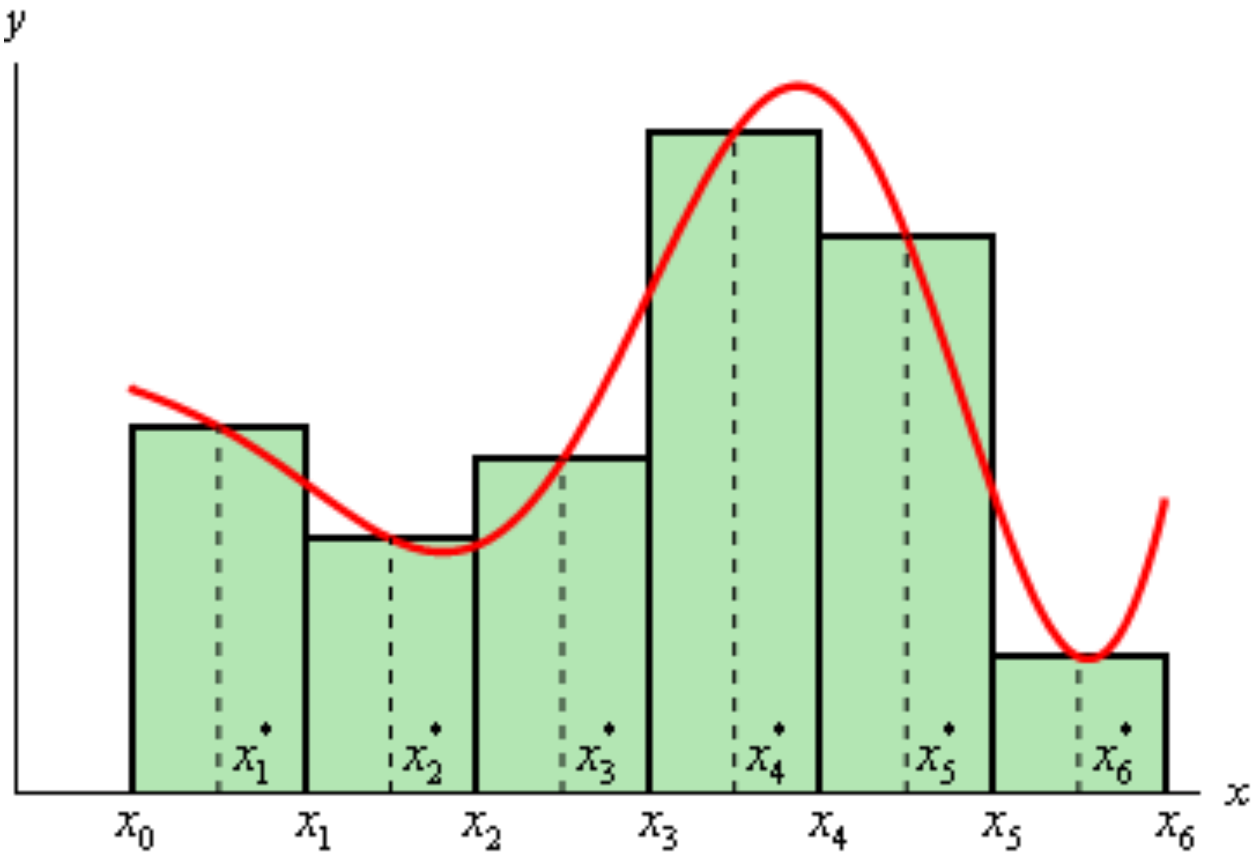
Example: computing an integral



- **Estimate integral with midpoint rule,**

$$I = \int_{x_0}^{x_6} f(x)dx$$

Example: computing an integral



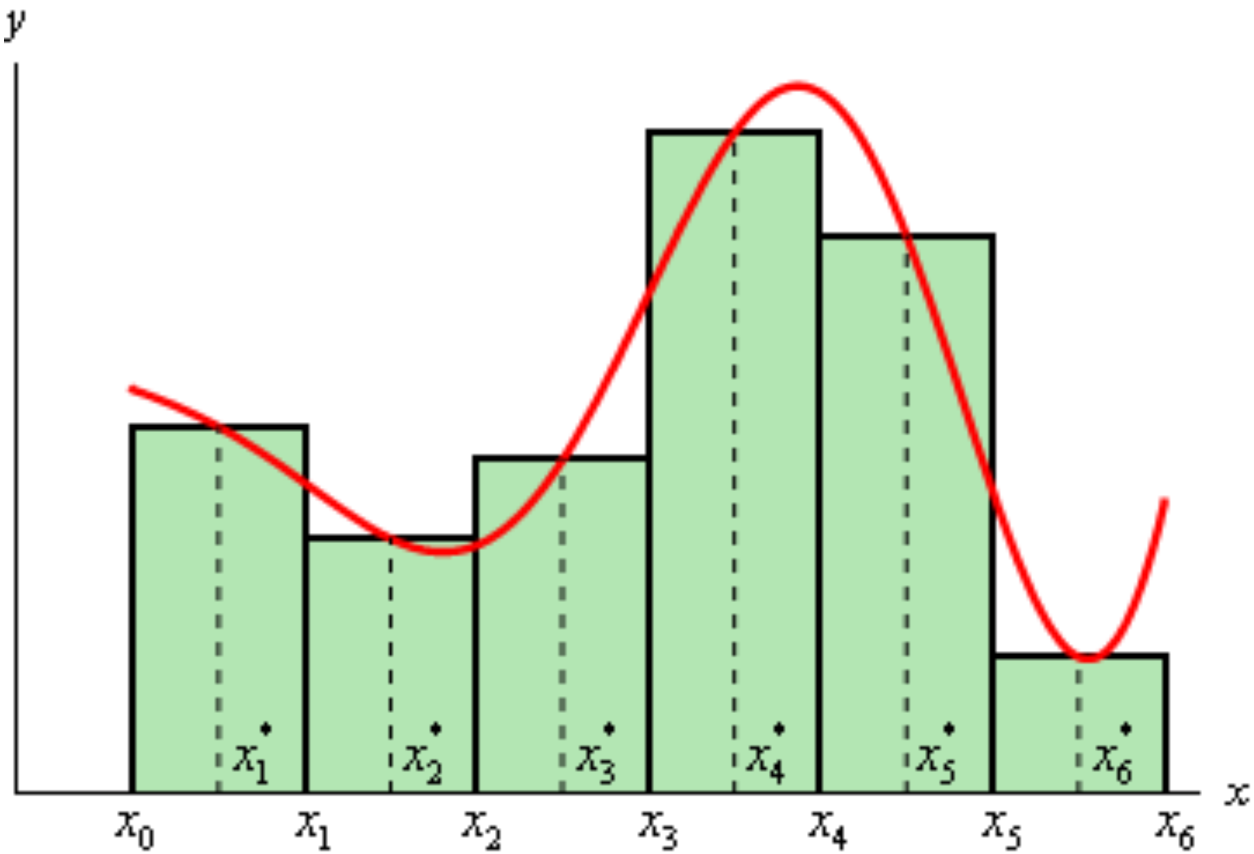
- **Estimate integral with midpoint rule,**

$$I = \int_{x_0}^{x_6} f(x)dx$$

1. **Compute:**

$$f(x_1^*), f(x_2^*), \ldots$$

Example: computing an integral



- **Estimate integral with midpoint rule,**

$$I = \int_{x_0}^{x_6} f(x)dx$$

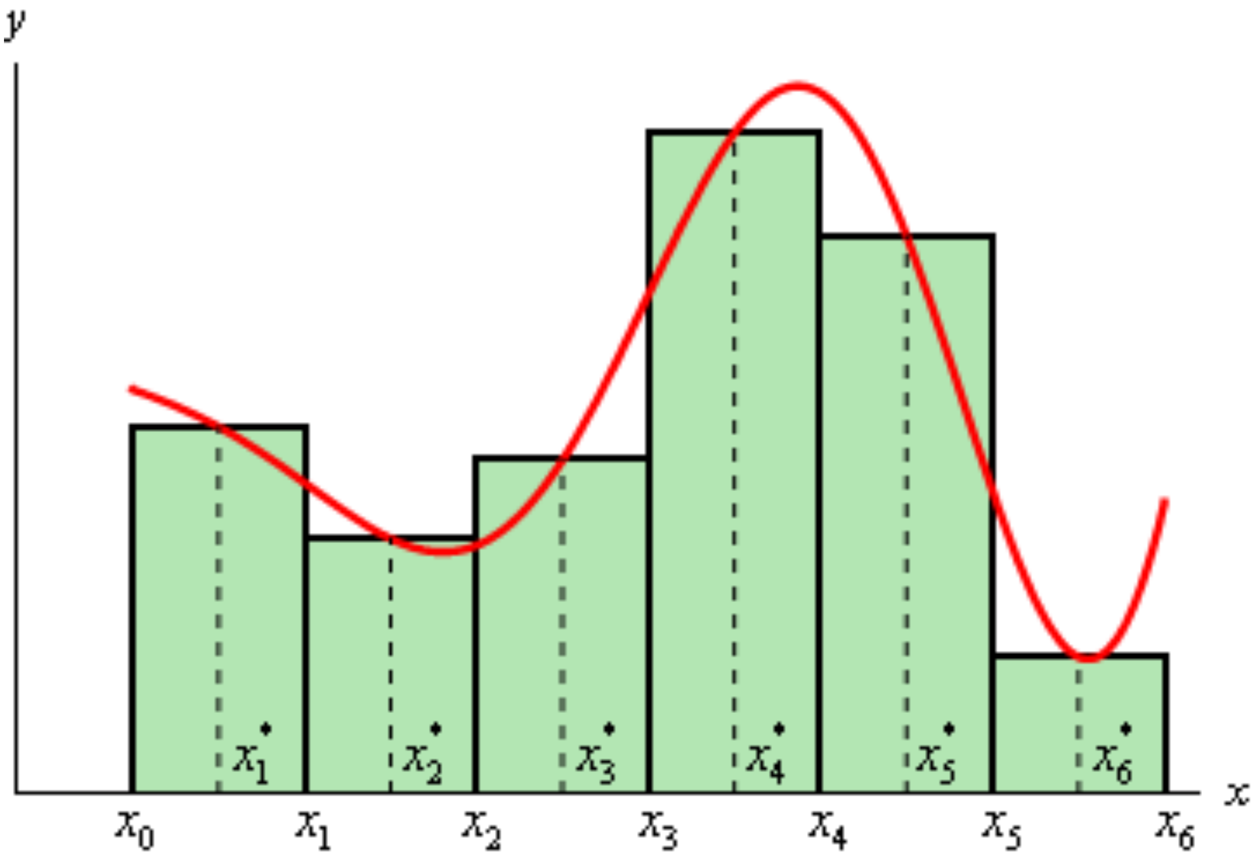
1. **Compute:**

$$f(x_1^*), f(x_2^*), \dots$$

2. **Compute areas of rectangles:**

$$I_1 = (x_1 - x_0) * f(x_1^*)$$

Example: computing an integral



- **Estimate integral with midpoint rule,**

$$I = \int_{x_0}^{x_6} f(x)dx$$

1. **Compute:**

$$f(x_1^*), f(x_2^*), \ldots$$

2. **Compute areas of rectangles:**

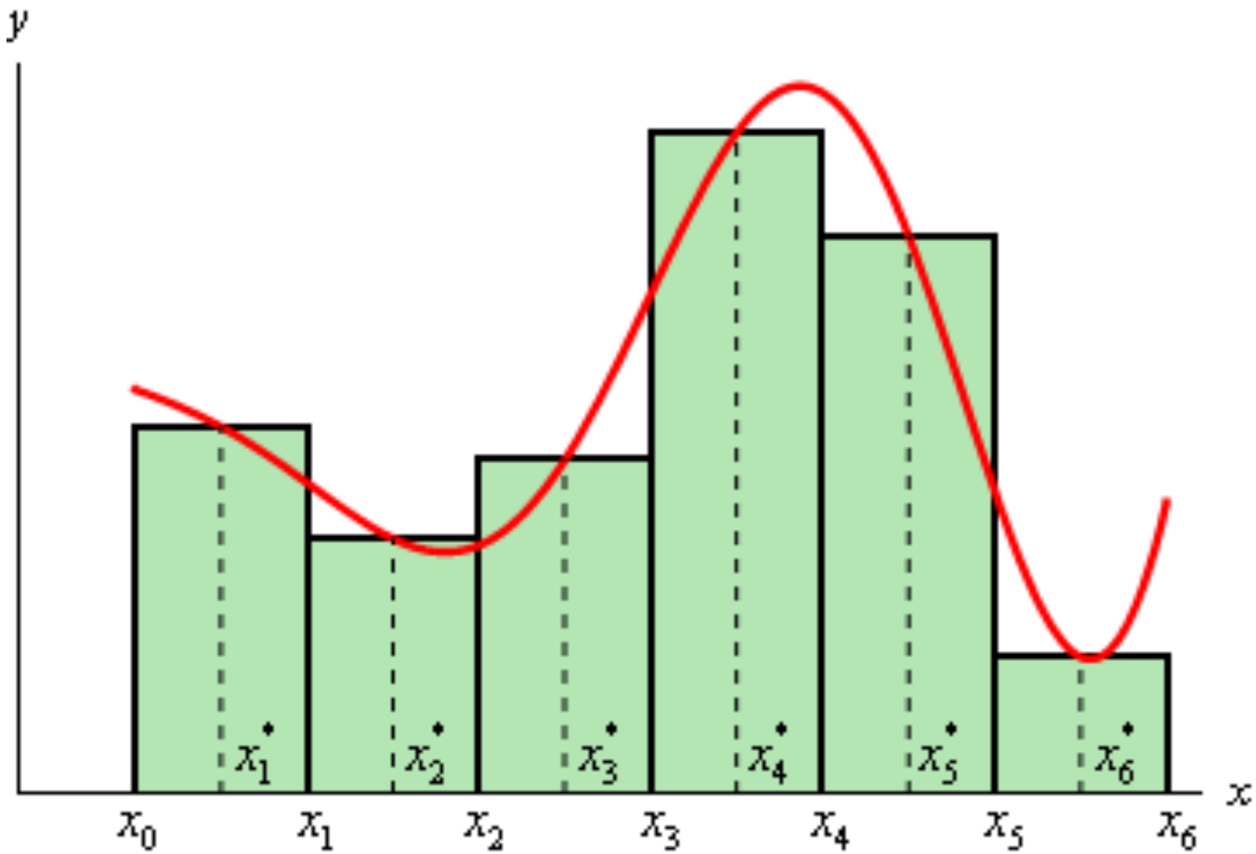
$$I_1 = (x_1 - x_0) * f(x_1^*)$$

3. **Sum areas:**

$$I \approx I_1 + I_2 + I_3 + \ldots$$

Imperial College
London

Example: computing an integral



- **Estimate integral with midpoint rule,**

$$I = \int_0^1 \frac{4}{1+x^2} dx$$

Quadrature example

Basic steps:

1. **Read in number of intervals,** N

2. **Compute interval size,** dx = 1.d0/N

3. **Loop over the N intervals, within each interval:**
    1. **compute the midpoint,** x_m
    2. **evaluate $4/(1+x^2)$ at midpoint**
    3. **compute area of i^{th} rectangle:** $sum_i = dx*f(x_m)$

See *midpoint.f90*

Quadrature example

Basic steps:

1. **Read in number of intervals,** N

```
!read data from data.in
    open(unit=10, file='data.in')
        read(10,*) N
    close(10)
```

Quadrature example

Basic steps:

1. **Read in number of intervals,** N
2. **Compute interval size,** dx = 1.0/N

```
!read data from data.in
    open(unit=10, file='data.in')
        read(10,*) N
    close(10)

    dx = 1.d0/dble(N) !interval size
```

Quadrature example

Basic steps:

3. Loop over the N intervals, within each interval:

1. **compute the midpoint,** x_m
2. **evaluate $4/(1+x^2)$ at midpoint**
3. **compute area of i^{th} rectangle:** $sum_i = dx*f(x_m)$

```
!loop over intervals computing each interval's contribution to integral
    do i1 = 1,N
        xm = dx*(dble(i1)-0.5d0) !midpoint of interval i1
        call integrand(xm,f)
        sum_i = dx*f
        sum = sum + sum_i !add contribution from interval to total
integral
    end do
```

Here, *integrand*, **is a subroutine which evaluates $4/(1+x^2)$ at xm**

Quadrature example

Here, *integrand*, is a subroutine which evaluates $4/(1+x^2)$ at xm

```
!----------------------------------
!subroutine integrand
!   compute integrand, 4.0/(1+a^2)
!----------------------------------

subroutine integrand(a,f)
    implicit none
    real(kind=8), intent(in) :: a
    real(kind=8), intent(out) :: f
    f = 4.d0/(1.d0 + a*a)
end subroutine integrand
```

Fortran reference: do loops

```
 integer :: i1,start,finish,step

!do-loop structure
    do i1 = start,finish,step

        !commands which depend on i1 in some way

    end do
```

- *do-loop* **index must be an integer**
- **Use** *exit* **to break a do-loop**

Fortran reference: if-then

```fortran
!if-then structure
    if (boolean expression here) then

        !some commands

    elseif (another boolean) then

        !more commands

    elseif (another boolean) then

        !even more commands

    else

        !more commands

    end if
```

- **Can have arbitrary number of *elseif* blocks**
- **Can also have just the 1st if statement**

Fortran reference: if-then

- **Relational operators:**

`.lt.`	or	`<`	less than
`.le.`	or	`<=`	less than or equal
`.eq.`	or	`==`	equal
`.ge.`	or	`>=`	greater than or equal
`.gt.`	or	`>`	greater than
`.ne.`	or	`/=`	not equal

`.not.`	not
`.and.`	and
`.or.`	inclusive or