ASTR 660 / Class 6

Homework 1 feedback

Big numbers

1 kB = 1024 Bytes

1 MB = 1024 kB = (1024**2) Bytes

1 GB = 1024 MB = (1024**3) Bytes

1 TB = 1024 GB = (1024**4) Bytes

For the history of the 1024, see: https://en.wikipedia.org/wiki/Byte

64-bit integers

examples/numbers/ranges.f90:

```
program test_range
  implicit none

  integer(4) :: i32
  integer(8) :: i64

  write(*,'(a,i20,i20)') "Integer(4)            ", -huge(i32)-1, huge(i32)
  write(*,'(a,i20,i20)') "Unsigned Integer(4) * ",            0, 2_8*huge(i32)+1
  write(*,'(a,i20,i20)') "Integer(8)            ", -huge(i64)-1, huge(i64)
  write(*,*)
  write(*,*) "* There are no unsigned ints in Fortran!"
end program test_range
```

64-bit integers

Integer(4)	-2147483648	2147483647
Unsigned Integer(4) *	0	4294967295
Integer(8)	-9223372036854775808	9223372036854775807

* There are no unsigned ints in Fortran!

Conclusion: if you want more than 4.29 billion unique numbers, you need int64.

Python data type sizes

Raw python “floats” and “integers” are not 32 bit / 4 byte. Each float is 24 bytes!

- *8 bytes for the underlying double-precision value;*
- *8 bytes for a pointer to the object type;*
- *8 bytes for a reference count (used for garbage collection)*

Quoting from: https://stackoverflow.com/questions/29826523/python-float-precision-float

This is why we use `numpy` rather than raw Python. With `numpy`, the 16 bytes of overhead are per array, not per value.

In Fortran (for simple numbers/arrays) there is no concept of “object type” or “reference count”.

Aquarius

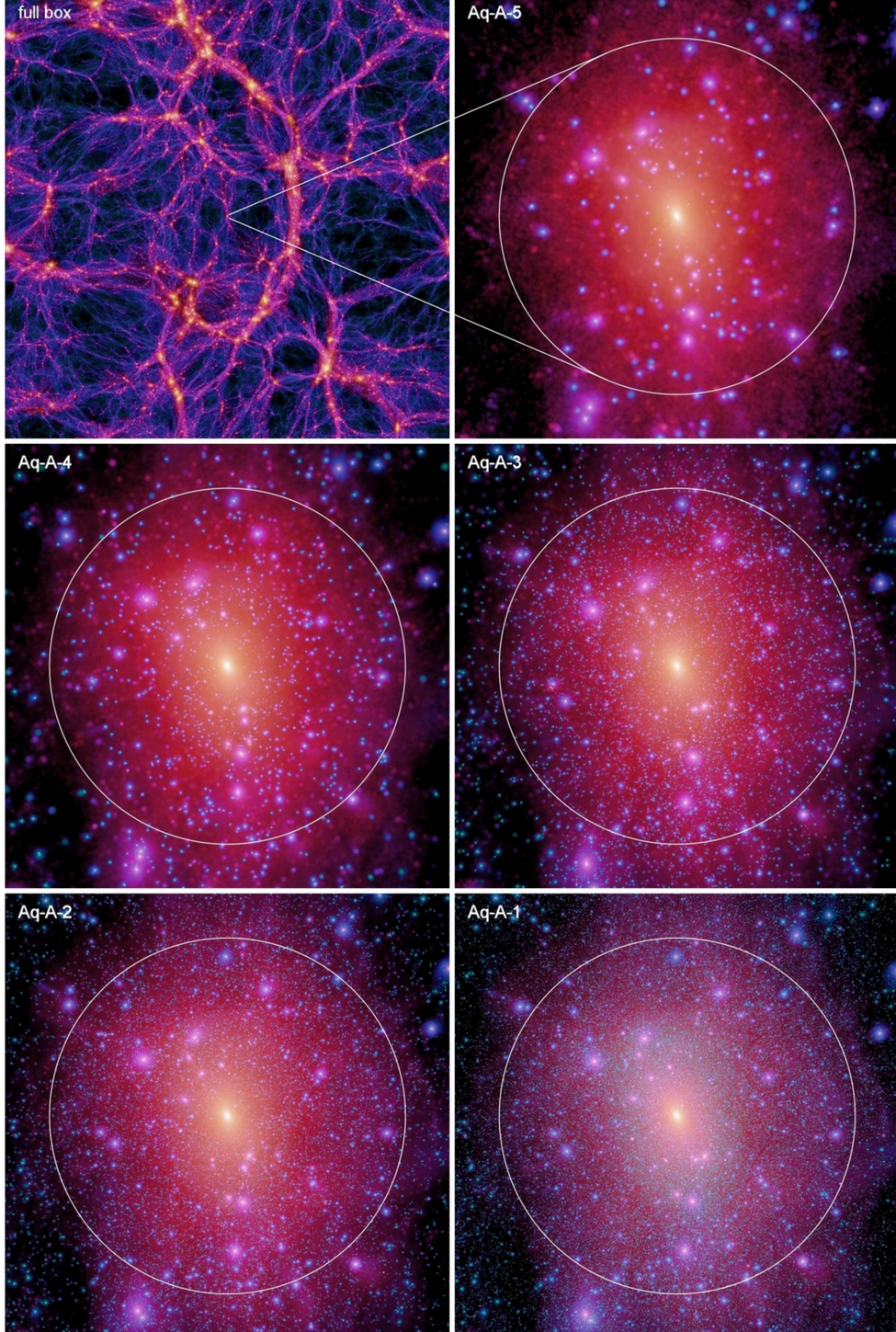
A **zoom** simulation: different regions of the initial conditions are sampled with **different** particle masses.

First, run a low resolution, uniform “parent box”, then pick one halo.

Trace back the **Lagrangian volume** of its particles to the initial conditions.

Replace particles in that volume with many more lower-mass (“light”) particles and re-run.

Heavy particles provide the same large-scale tidal forces but lower the computational cost.



Aquarius

Very important to test for **convergence** using a series of increasingly high resolutions.

This is important more generally: which results from a simulation depend on resolution?

Can only trust results that have converged (or can be extrapolated reliably).

Should try to do at least one run to higher resolution than the baseline for your ‘sample’ (like Aq-A-1) to test for convergence.

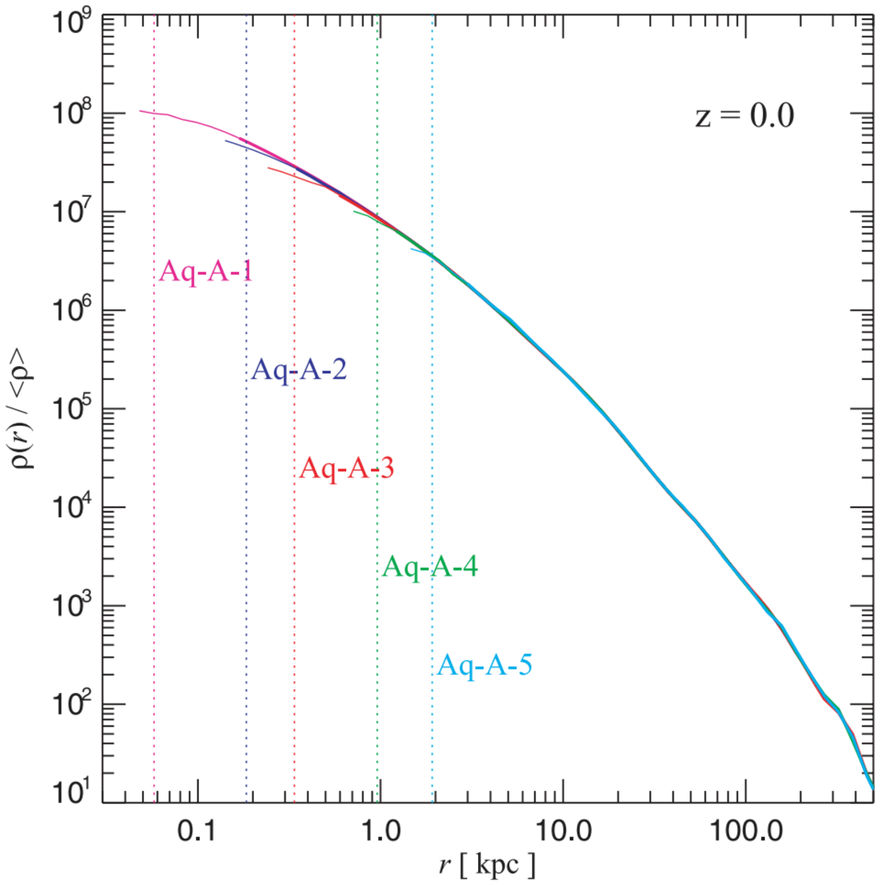
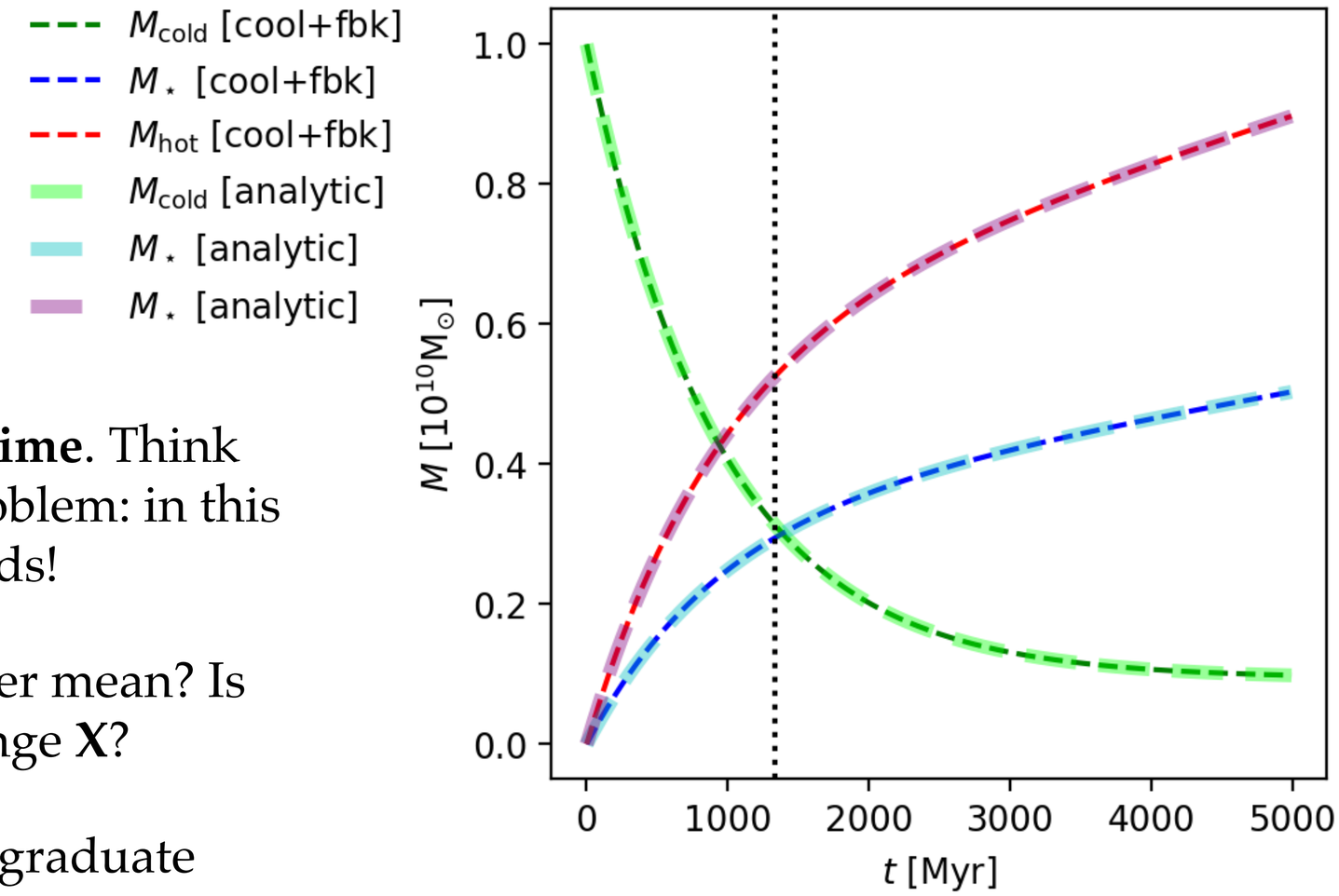


Figure 4. Spherically averaged density profile of the Aq-A halo at $z = 0$, at different numerical resolutions. Each of the profiles is plotted as a thick line for radii that are expected to be converged according to the resolution criteria of Power et al. (2003). These work very well for our simulation set. We continue the measurements as thin solid lines down to 2ϵ, where ϵ is the Plummer-equivalent gravitational softening length in the notation of Springel et al. (2001b). The dotted vertical lines mark the scale 2.8ϵ, beyond which the gravitational force law is Newtonian. The mass resolution changes by a factor of 1835 from the lowest to the highest resolution simulation in this series. Excellent convergence is achieved over the entire radial range where it is expected.

ODE problem



Sensible unit choices: in particular, for **time**. Think about the characteristic scales of the problem: in this case millions of years rather than seconds!

What am I doing? What does this answer mean? Is that reasonable? What happens if I change **X**?

Clear graphics: scale, units on axis. For graduate students especially: look at figures in papers.

Well done to those who attempted the second part!

ASTR 660 / Class 5 update

Variable initialisation

Initialising Variables: Correction!

Last week I said this:

- 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?)

I was wrong. Here is why not!

Initialising Variables: Correction!

https://fortran-lang.org/learn/quickstart/gotchas/#implied-save

```
program main
implicit none
    integer :: i

    do i = 1, 5
        call foo()
    end do
end program
```

```
subroutine foo()
implicit none
    integer :: c=0

    c = c+1
    print*, c
end subroutine
```

Initialising Variables: Correction!

https://fortran-lang.org/learn/quickstart/gotchas/#implied-save

```
subroutine foo()
implicit none
    integer :: c=0

    c = c+1
    print*, c
end subroutine

program main
implicit none
    integer :: i

    do i = 1, 5
        call foo()
    end do
end program
```

People used to C/C++ expect this program to print 5 times `1`, because they interpret `integer :: c=0` as the concatenation of a declaration and an assignment, as if it was:

```
integer :: c
c = 0
```

But it is not. This program actually outputs:

```
1
2
3
4
5
```

Initialising Variables: Correction!

https://fortran-lang.org/learn/quickstart/gotchas/#implied-save

```
subroutine foo()
implicit none
    integer :: c=0

    c = c+1
    print*, c
end subroutine

program main
implicit none
    integer :: i

    do i = 1, 5
        call foo()
    end do
end program
```

`integer :: c=0` is actually a one-shot **compile time initialization**, and it makes the variable persistent between calls to `foo()`. It is actually equivalent to:

```
integer, save :: c=0
```

Lesson learned:

Always initialise variables, but in two separate steps: declare, then assign.

ASTR 660 / Class 6

Fortran crash course 2: Subroutines

Subroutines

Python only has one kind of ‘function’:

```
b = compute_something(x,y) # Return value is assigned to “b”

compute_something(x,y) # Return value isn’t assigned to anything
```

Fortran distinguishes functions (which return values) from subroutines (which do not).

```
b = compute_something(x,y) # Function; return value assigned to “b”

call compute_something(x,y) # Subroutine; no return value!
```

Fortran functions can be used on the **right-hand side of an assignment**, but subroutines **cannot** (because they don’t return anything). Subroutines have to be called using `call`.

Subroutines

```
subroutine print_triangle(n)
  implicit none

  integer, intent(IN) :: n
  integer :: i, j

  outer_loop: do i = 1, n
    inner_loop: do j = 1, i
      write(*,'(a)',advance="no") '* '
    end do inner_loop
    write(*,*)
  end do outer_loop

end subroutine
```

```
program print_shapes
  implicit none

  ! No type declaration for subroutine here

  call print_triangle(10)

end program print_shapes
```

Subroutines

Subroutines have to be called using `call`.

```
subroutine print_triangle(n)
  implicit none

  integer, intent(IN) :: n
  integer :: i, j

  outer_loop: do i = 1, n
    inner_loop: do j = 1, i
      write(*,'(a)',advance="no") '* '
    end do inner_loop
    write(*,*)
  end do outer_loop

end subroutine

program print_shapes
  implicit none

  ! No type declaration for subroutine here

  call print_triangle(10)

end program print_shapes
```

Argument `intent`

`intent(IN)` is optional, but always use it.

It tells the compiler to **complain** if you try to alter an argument that should **not** be changed by the function or subroutine.

It also makes the code easier to read.

It also helps the compiler to optimise in situations where large objects (like arrays) are passed as arguments.

```
subroutine print_triangle(n)
  implicit none

  integer, intent(IN) :: n
  integer :: i, j

  outer_loop: do i = 1, n
    inner_loop: do j = 1, i
      write(*,'(a)',advance="no") '* '
    end do inner_loop
    write(*,*)
  end do outer_loop

end subroutine

program print_shapes
  implicit none

  ! No type declaration for subroutine here

  call print_triangle(10)

end program print_shapes
```

Loop labels

Loops can be labelled. This is just to help **readability**, when the loops run over many lines and have nested levels.

Indentation also helps readability, but is not essential in Fortran.

```
subroutine print_triangle(n)
  implicit none

  integer, intent(IN) :: n
  integer :: i, j

  outer_loop: do i = 1, n
    inner_loop: do j = 1, i
      write(*,'(a)',advance="no") '* '
    end do inner_loop
    write(*,*)
  end do outer_loop

end subroutine

program print_shapes
  implicit none

  ! No type declaration for subroutine here

  call print_triangle(10)

end program print_shapes
```

Terminal newline

A newline after each write statement is automatic, unless you give the option
advance = “no”

```
subroutine print_triangle(n)
  implicit none

  integer, intent(IN) :: n
  integer :: i, j

  outer_loop: do i = 1, n
    inner_loop: do j = 1, i
      write(*,'(a)',advance="no") '* '
    end do inner_loop
    write(*,*)
  end do outer_loop

end subroutine

program print_shapes
  implicit none

  ! No type declaration for subroutine here

  call print_triangle(10)

end program print_shapes
```

Python functions

In Python, what happens when a function returns nothing?

```
def return_nothing():
    pass

b = return_nothing()
```

Subroutines and scope

Subroutines **can** alter things outside their own scope, e.g. variables passed as arguments.

```
subroutine triangle_area(n,area)
  implicit none
  integer, intent(IN)  :: n
  real,    intent(OUT) :: area

  area = 0.5*real(n)**2
end subroutine

program print_area
  implicit none
  ! Area is declared here
  real :: area
  ! Area is assigned by the subroutine
  call triangle_area(10,area)

  write(*,'(a,1x,f5.2,1x,a)') "The area of the triangle is", area, "sq. units"
end program print_area
```

Subroutines and scope

```
subroutine triangle_area(n,area)
  implicit none
  integer, intent(IN)  :: n
  real,    intent(OUT) :: area

  area = 0.5*real(n)**2
end subroutine

program print_area
  implicit none
  ! Area is declared here
  real :: area
  ! Area is assigned by the subroutine
  call triangle_area(10,area)

  write(*,'(a,1x,f5.2,1x,a)') "The area of the triangle is", area, "sq. units"
end program print_area
```

What happens if a literal number or PARAMETER is passed as “area”?

What if we don’t have `intent(OUT)`?

Subroutines and scope

```
subroutine triangle_area(n,area)
  implicit none
  integer, intent(IN)  :: n
  real :: area

  area = 0.5*real(n)**2
end subroutine
```

```
. . .
! Area is declared here
real, PARAMETER :: area = 10.0
call triangle_area(10,area)
```

Output:

```
Program received signal SIGBUS: Access to an undefined portion of a memory object.

Backtrace for this error:
#0  0x102245b07
#1  0x102244b53
#2  0x1a49a6a23
#3  0x10202bdb3
[1]    33963 bus error  ./subroutine_inout_broken
```

Subroutines and scope

```
Program received signal SIGBUS: Access to an undefined portion of a memory object.

Backtrace for this error:
#0  0x102245b07
#1  0x102244b53
#2  0x1a49a6a23
#3  0x10202bdb3
[1]    33963 bus error  ./subroutine_inout_broken
```

This is a stack trace: where did the program crash?

After compiling, the names you give to your functions and variables have been replaced by their locations in memory.

You can tell the compiler to remember which names go with which addresses (which just makes your binary file slightly larger).

Adding debugging symbols with -g

```
gfortran -g -o subroutine_inout_broken subroutine_inout_broken.f90
```

```
Program received signal SIGBUS: Access to an undefined portion of a memory object.

Backtrace for this error:
#0  0x100ff9ad3 in ???
#1  0x100ff8b53 in ???
#2  0x1a49a6a23 in ???
#3  0x100ddfdb3 in print_area
  at /Users/apcooper/…/subroutines/subroutine_inout_broken.f90:17
[1]    34124 bus error  ./subroutine_inout_broken
```

Subroutines summary

Next week we will look at organising subroutines and functions into separate modules.

For this week the important lessons are:

Use functions where there is a single clear "return" value and the procedure has no side effects. Use subroutines for everything else.

Use `intent(IN)`, `intent(OUT)` and `intent(INOUT)` to make it clear what functions and subroutines do with their arguments.

ASTR 660 / Class 6

Fortran crash course 2: Arrays

Arrays in Fortran

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

All serious scientific programming has to use "vectorised operations" on arrays.

We said this was one of the main strengths of Fortran, so let's see it in action:

```
program arrays
  implicit none

  ! 1D integer array
  integer, dimension(10) :: array1

  ! An equivalent array declaration
  integer :: array2(10)

  ! 2D real array
  real, dimension(10, 10) :: array3

  ! Custom lower and upper index bounds
  real :: array4(0:9)
  real :: array5(-5:5)

end program arrays
```

Arrays in Fortran

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

For simple arrays, the size is part of the declaration, optionally using the “dimension” keyword.

```
program arrays
  implicit none

  ! 1D integer array
  integer, dimension(10) :: array1

  ! An equivalent array declaration
  integer :: array2(10)

  ! 2D real array
  real, dimension(10, 10) :: array3

  ! Custom lower and upper index bounds
  real :: array4(0:9)
  real :: array5(-5:5)

end program arrays
```

Arrays in Fortran

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

Multi-dimensional arrays are straightforward.

```
program arrays
  implicit none

  ! 1D integer array
  integer, dimension(10) :: array1

  ! An equivalent array declaration
  integer :: array2(10)

  ! 2D real array
  real, dimension(10, 10) :: array3

  ! Custom lower and upper index bounds
  real :: array4(0:9)
  real :: array5(-5:5)

end program arrays
```

Arrays in Fortran

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

```
program arrays
  implicit none

  ! 1D integer array
  integer, dimension(10) :: array1

  ! An equivalent array declaration
  integer :: array2(10)

  ! 2D real array
  real, dimension(10, 10) :: array3

  ! Custom lower and upper index bounds
  real :: array4(0:9)
  real :: array5(-5:5)

end program arrays
```

You can also assign custom ranges of index values to your arrays. Use this power sparingly and with care.

Note: In Fortran, you can't use negative indices to count backwards from the end of an array.

Array access is similar to numpy

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

```
program array_slice
  implicit none

  integer :: i
  integer :: array1(10)  ! 1D integer array of 10 elements
  integer :: array2(10, 10)  ! 2D integer array of 100 elements

  array1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  ! Array constructor
  array1 = [(i, i = 1, 10)]  ! Implied do loop constructor
  array1(:) = 0  ! Set all elements to zero
  array1(1:5) = 1  ! Set first five elements to one
  array1(6:) = 1  ! Set all elements after five to one

  print *, array1(1:10:2)  ! Print out elements at odd indices
  print *, array2(:,1)  ! Print out the first column in a 2D array
  print *, array1(10:1:-1)  ! Print an array in reverse

end program array_slice
```

Array access is similar to numpy

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

```
program allocatable
  implicit none

  integer, allocatable :: array1(:)
  integer, allocatable :: array2(:,:)

  allocate(array1(10))
  allocate(array2(10,10))

  ! ...

  deallocate(array1)
  deallocate(array2)

end program allocatable
```

As in numpy, Fortran arrays have a fixed size.

Often we don't know how big the array should be until we run the program. Perhaps it depends on user input.

Such arrays can be declared `allocatable`; we still have to give their dimensions.

When we know the size we want, we `allocate` the array before using it. We can free the memory later using `deallocate`.

Arrays allocated in functions and subroutines are automatically deallocated when they go out of scope.

Memory management in python

In Python, you usually don't have to worry about how memory is allocated and deallocated.

Behind the scenes, python is tracking references to the objects you have created. When an object has no references in scope, Python will free the memory it used — **but only when Python feels like doing so** (e.g. when a function returns, or a the program ends).

If your code is comes close to running out of memory, you might need to force objects to be deleted when **you** know you don't need them.

```
massive_array = np.zeros(1_000_000_000,dtype=np.int64)
…

del(massive_array) # Memory for the array can be reclaimed ASAP

gc.collect() # Free memory RIGHT NOW (don't overuse this)
```

Array maths

$$\mathbf{x}_i = \log_{10}(1 + i^2)$$

```
x( 1):  0.301
x( 2):  0.699
x( 3):  1.000
x( 4):  1.230
x( 5):  1.415
x( 6):  1.568
x( 7):  1.699
x( 8):  1.813
x( 9):  1.914
x(10):  2.004
```

```
program array_maths
  implicit none

  real, allocatable :: x(:)
  integer, parameter :: N = 10
  integer :: i

  if (.not. allocated(x)) then
    write(*,*) "At the start of the program, the array x has not been allocated!"
  end if

  allocate(x(N))

  if (allocated(x)) then
    write(*,*) "x has been allocated now!"
  end if

  ! Assign the same value to all elements of x
  x(:) = 1

  ! Loop over x
  ! Remember: the first element of x is x[1]
  do i = 1, size(x)
    x(i) = x(i) + i**2
  end do

  ! Vectorised operation on x:
  x(:) = log10(x)

  do i = 1, size(x)
    write(*,'(a,i2,a,f6.3)') 'x(',i,'): ', x(i)
  end do

  deallocate(x)
end program array_maths
```

Array maths

$$\mathbf{x}_i = \log_{10}(1 + i^2)$$

```
x( 1):  0.301
x( 2):  0.699
x( 3):  1.000
x( 4):  1.230
x( 5):  1.415
x( 6):  1.568
x( 7):  1.699
x( 8):  1.813
x( 9):  1.914
x(10):  2.004
```

```
! Vectorised operation on x:
x(:) = log10(x)
```

The `(:)` is not really needed here. I include it to remind myself (and whoever is reading the code) that this is an operation on an array.

Fortran arrays summary

```
x( 1):  0.301
x( 2):  0.699
x( 3):  1.000
x( 4):  1.230
x( 5):  1.415
x( 6):  1.568
x( 7):  1.699
x( 8):  1.813
x( 9):  1.914
x(10):  2.004
```

Python has a rich syntax for array manipulation, somewhat similar to numpy (most obvious differences: access elements using *round* brackets and by default the first element index has index 1, not 0).

The shape of an array has to be specified when the array is declared. Array `rank` and `size` can be checked with builtin Fortran functions.

Arrays can be fixed-size or allocatable (`allocate`).

Allocatable arrays can be checked (`allocated`) and deallocated (`deallocate`) to free memory.

As in numpy, try to avoid loops. Vectorized operations on arrays are fast! (Fortran loops are not as bad as Python loops).

ASTR 660 / Class 6

Numerical Derivatives

Derivatives

The derivative of $f(x)$ with respect to x is understood in terms of the difference over an infinitely small interval h:

$$\frac{df}{dx} \equiv \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

When we can't evaluate this analytically, we have to take a finite step size and live with the error in the approximation: primarily truncation error from using a limited number of terms, but also rounding error if we use very small values of h.

We have seen examples of the idea of taking finite steps in ODE solvers.

Derivatives

Given the Taylor series:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \ldots$$

We can rearrange for $f'(x)$:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(x) + \ldots$$

So to first order:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Forwards

Given the Taylor series:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \ldots$$

We can rearrange for $f'(x)$:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(x) + \ldots$$

So to first order:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Backwards

Given the Taylor series:

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \ldots$$

We can rearrange for $f'(x)$:

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{h}{2}f''(x) + \ldots$$

So to first order:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

Central difference

By subtracting the 'backwards' Taylor series from the 'forwards' Taylor series, we obtain a 'central difference' formula for the first derivative that is accurate to second order:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

If we add the two sequences instead, we get a central difference approximation for the second derivative, also second-order accurate:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)$$

Of course, we could keep going to higher derivatives.

Finite difference methods

We already looked **initial value problems**: ODEs for which we had complete knowledge of the solution at one point (e.g. position and velocity at $t = 0$) and needed to find the solution at another point, by moving forward in small steps.

The alternative case is a **boundary value problem**, in which we have partial information at two points and need to find the unique solution (if any) that satisfies both constraints.

BVPs can be solved with the same techniques as IVPs, using an iterative approach called the **shooting method**. This is computationally quite expensive.

An alternative approach is to split up the domain into small intervals (*discretise*) the domain on a *grid*/*mesh*) and use finite difference formulae.

This is the basis of solving important PDE problems, which we will look at in later classes.

Finite difference methods

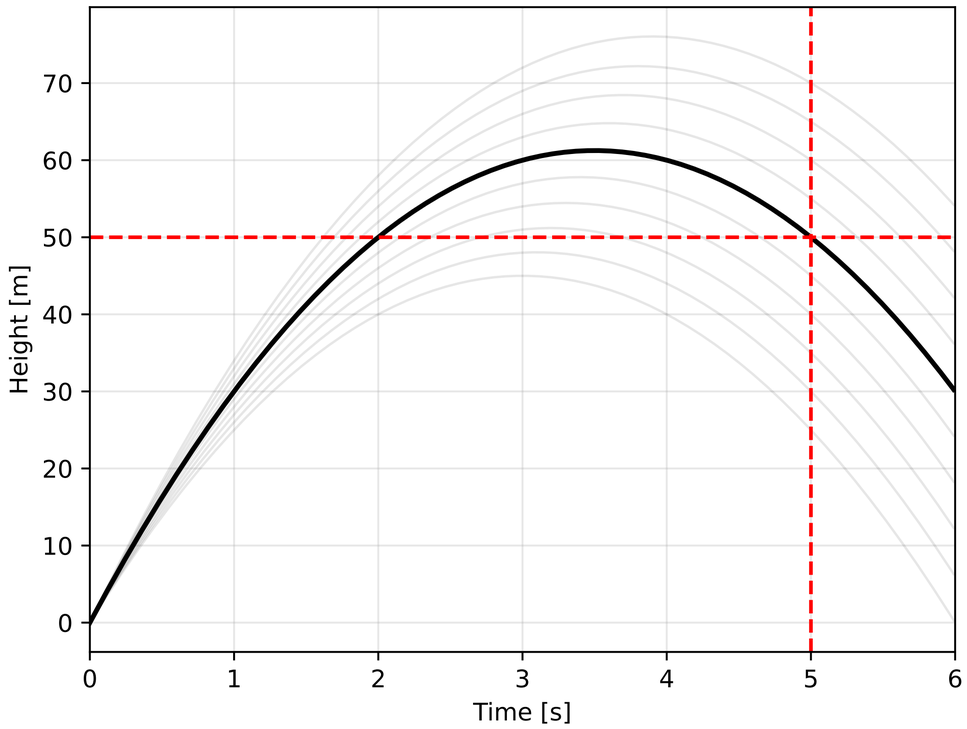
As a simple example, consider the height, $y(t)$, of a projectile under constant acceleration due to gravity.

We have $y'(t) = v(t)$ and $y''(t) \equiv v'(t) = -g = -10\,\mathrm{m\,s^{-1}}$ (for simplicity here).

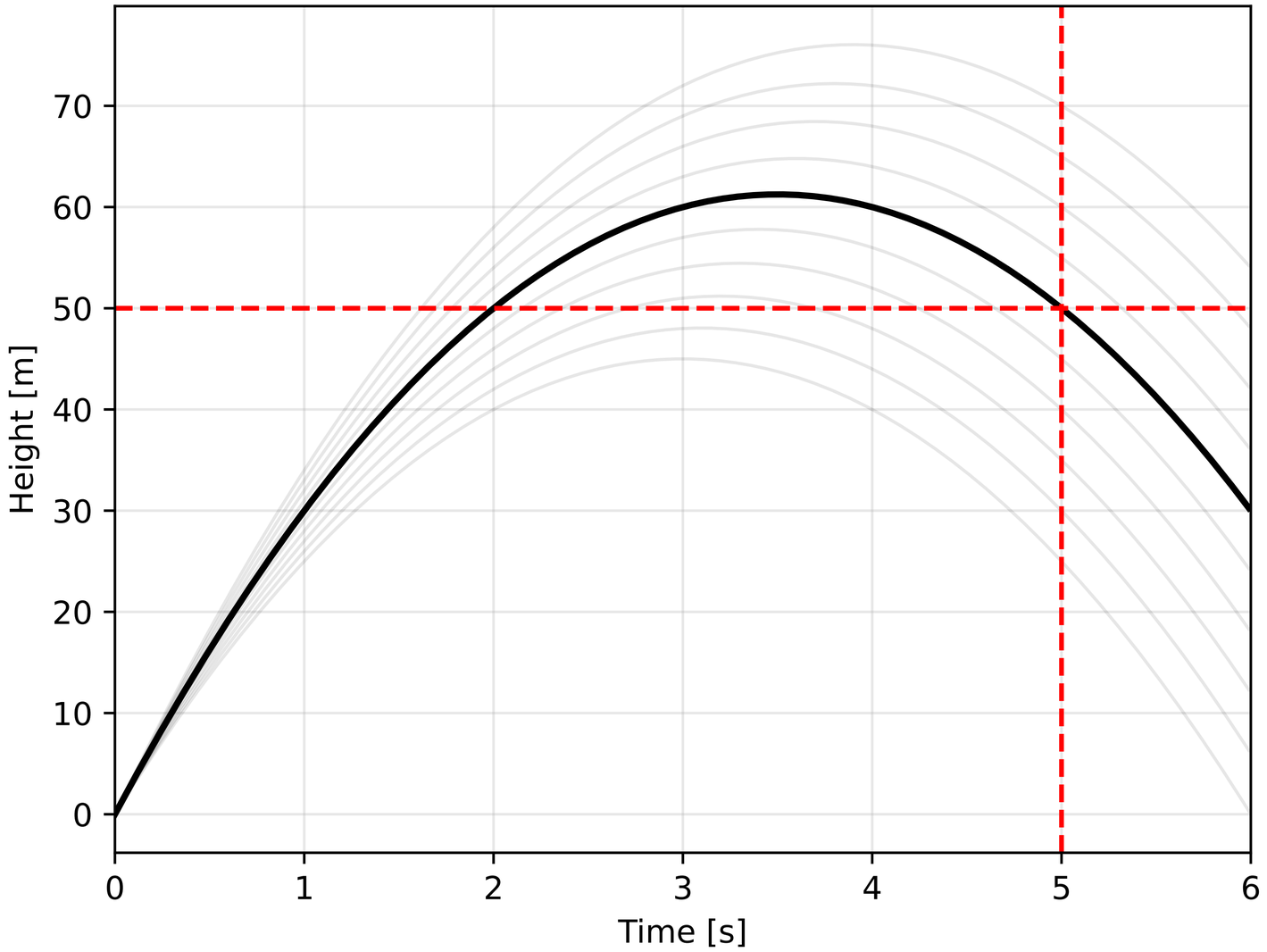
If we want the projectile to reach a height of 50m after 5s, what should $u \equiv v(t=0)$ be?

We can quickly find the right answer from the equations of motion: $u = 35\,\mathrm{m\,s^{-1}}$.

The projectile reaches $y = 50\,\mathrm{m}$ at $t = 2\mathrm{s}$ (going up) and $t = 5\mathrm{s}$ (coming down).

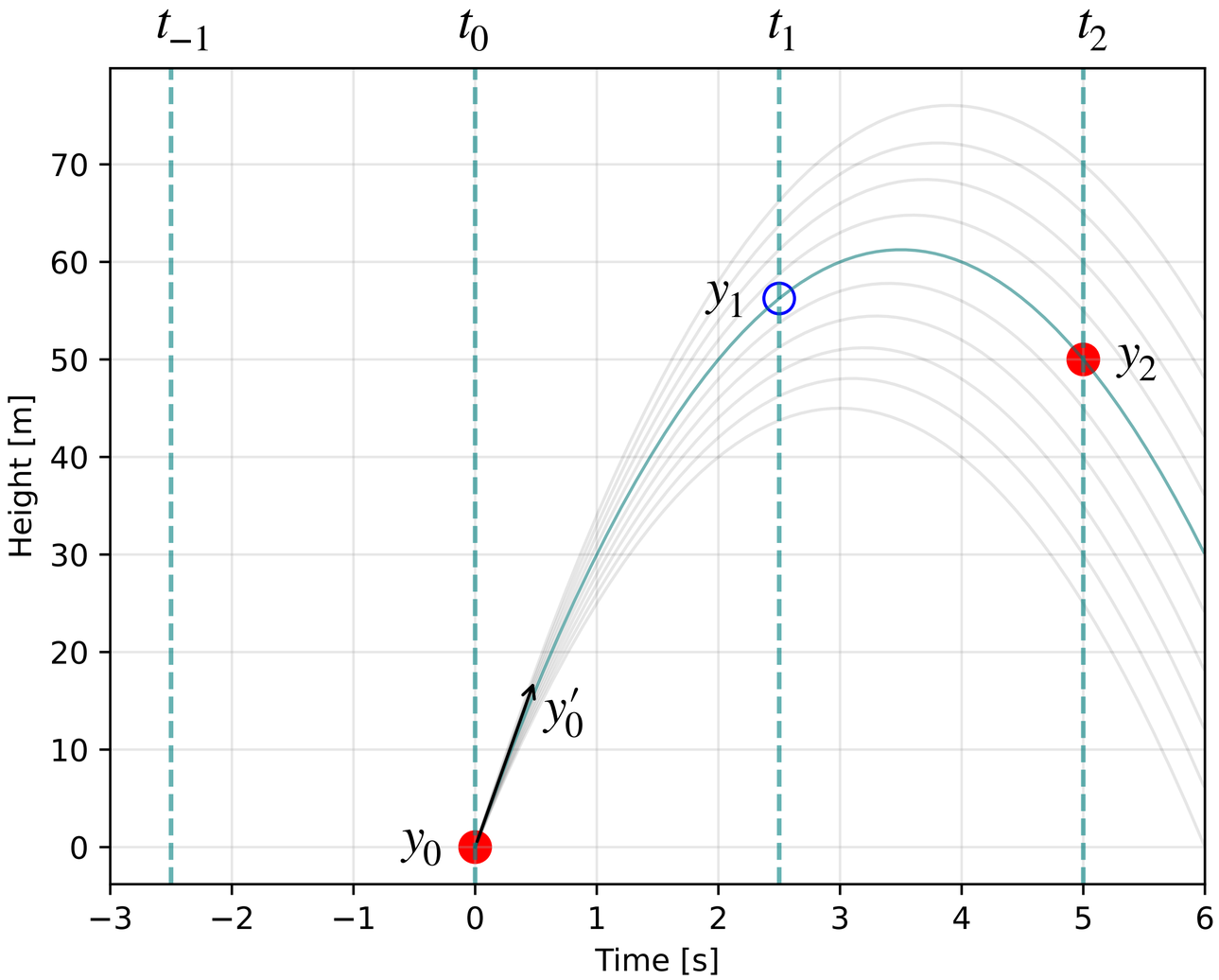


Finite difference methods



Finite difference methods

We can find the same solution using finite difference formulae.



Finite difference methods

Writing the 2nd derivative formula with different notation to match the plot, we have that:

$$y_{i+1} - 2y_i + y_{i-1} = h^2 y_i'' = -gh^2.$$

We can solve this for y_1,

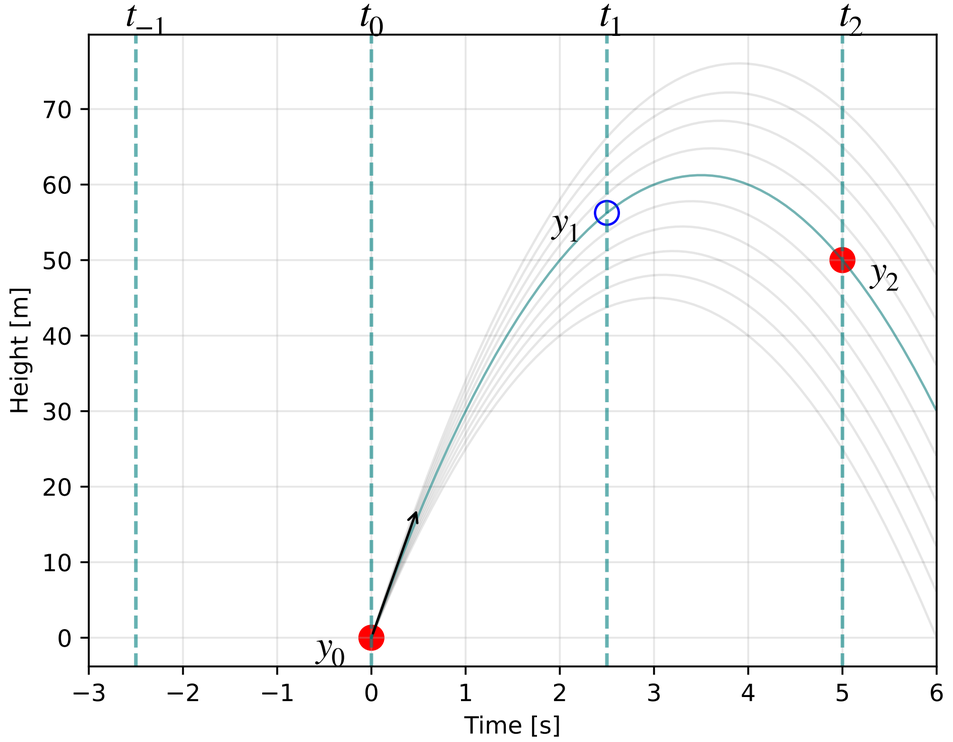
$$y_1 = \frac{50 + 0 + (2.5)^2(10)}{2} = 56.25\,\text{m}.$$

In this case the solution is exact. We really want the velocity at t_0, which we could get from the first derivative formula,

$$y_0' = \frac{y_1 - y_{-1}}{2h}.$$

We can find y_{-1} by substituting from the 2nd derivative formula evaluated at y_0: $y_{-1} = 2y_0 - y_1 - gh^2$.

$$\implies y_0' = \frac{(2)(56.25) + (2.5)^2(10)}{5} = 35\,\text{m}\,\text{s}^{-1}$$



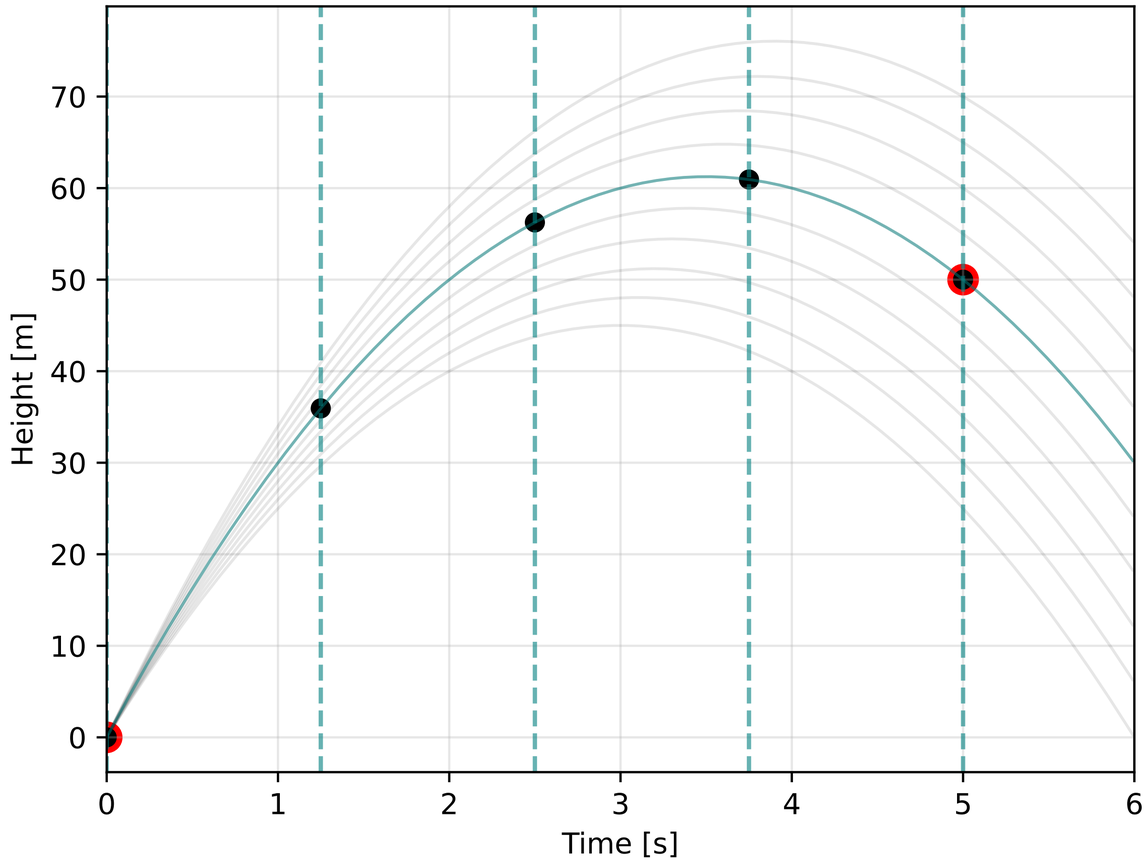
Linear systems

This step-by-step logic would get tedious if the system was not so simple.

Let's say we have three grid points between the boundaries, rather than one)

We can write the system of equations (including the boundary conditions) as a matrix equation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ -gh^2 \\ -gh^2 \\ -gh^2 \\ 50 \end{bmatrix}$$



Solving linear systems in Python

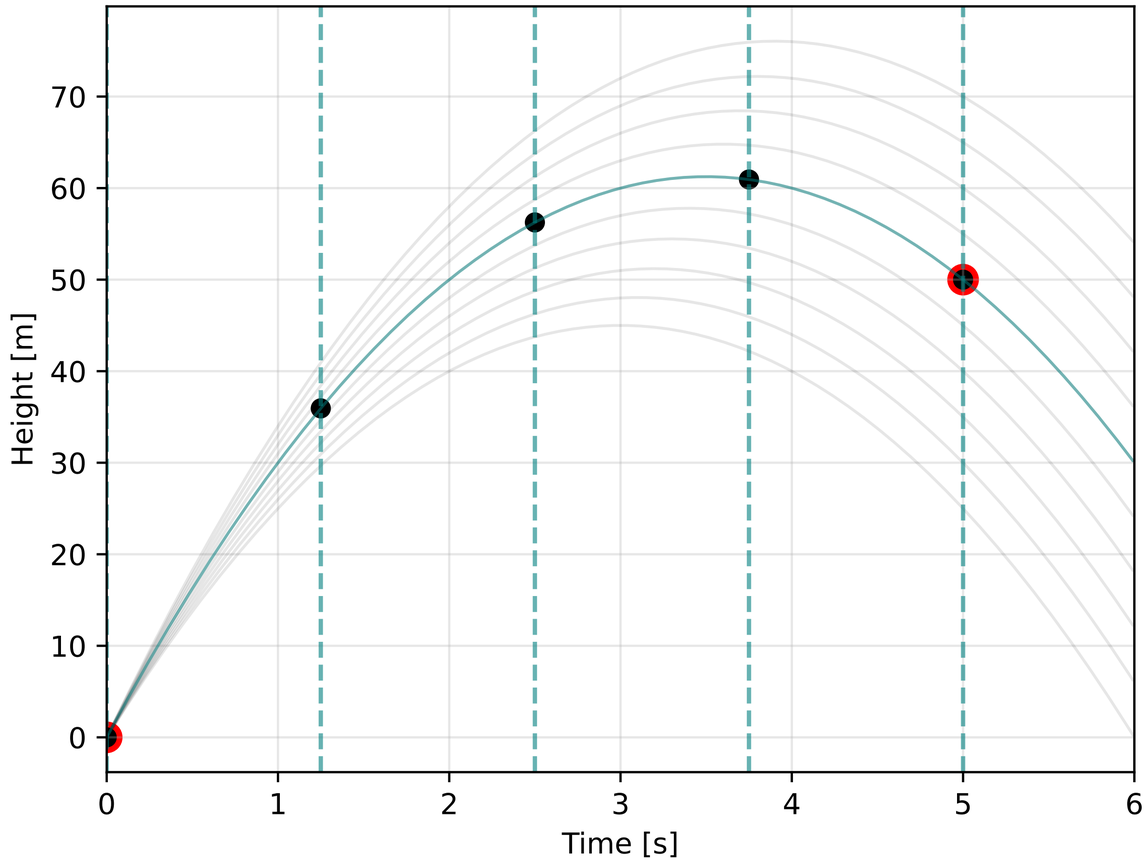
```
# The grid
t_max  = 5.0
N_grid = 5
h = t_max/(N_grid-1)
gridpoints = np.linspace(0,5,N_grid)

# We will solve the linear system Ax = B
A = np.matrix(np.zeros((N_grid,N_grid),np.float32))

A[0,0], A[-1,-1] = 1, 1 # BCs
for irow in range(1,N_grid-1):
    A[irow, irow-1:irow+2] = 1,-2, 1
print(A)

B = np.matrix(np.repeat(-g*h**2, N_grid)).T
B[0], B[-1]  = 0, 50 # BCs
print(B)

X = np.linalg.solve(A,B)
```



Finite difference methods

We should see this method in action on a problem where the solution is not so obvious!

We will look at several such examples in future classes, especially for the solution of fluid-dynamical PDEs.

Summary:

Central difference formula for the first derivative:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

Central difference formula for the second derivative:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)$$

Finite difference formulae can be used to solve boundary value problems on a mesh, using linear algebra.