# • Assignment 1

• 2.6:

- Newton-Raphson method for nonlinear eqns form $f(x) = 0$

given initial guess $x_0$

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})} \qquad i = 1, 2, 3, \ldots$$

terminated for $|x_i - x_{i-1}| < \varepsilon$ & e.g. $f(x) = e^x + x^3 - 5$ & $x_0 = 0$

⇒ code:

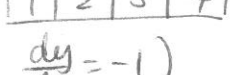
```
double fn( double x0, double E)
{  double x_{i-1} = x0;  // declare & initialise
   double x_i = x_{i-1} - f(x_{i-1}) / f'(x_{i-1});

   while ( fabs(x_i - x_{i-1}) > E)
   { x_{i-1} = x_i;
     x_i = x_{i-1} - f(x_{i-1}) / f'(x_{i-1}); }

   return x_i; }
```

• 3.3:

- implicit Euler method (writes $x_n, y_n$ for $y = e^{-x}$ as soln to $\frac{dy}{dx} = -1$)




$$y_0 = 1, \quad \frac{y_{n+1} - y_{n-1}}{h} = -y_n, \quad n = 1, 2, \ldots, N-1$$

⇒ code:

```
void fn(int n)  //since not returning anything
{ double const h = double(1)/double(n);
  std::ofstream write_output ("xy.dat");   // output stream variable write_output specified as type std::ofstream
  double y_{n-1} = 1.0;  //initial y value
  for (int i=0; i<n; i++)
  { double x_n = double(i) * h;
    double y_n = y_{n-1};  //so written in file as part of loop
    write_output << x_n << "," << y_n << "\n";
    y_n = y_{n-1} / (1.0 + h);  // by rearranging eqn
    y_{n-1} = y_n;  //used in next iteration until N-1 is reached }
  write_output.close();  // close file handle (flushes buffer so all data buffered is written to file before computer executes any further statements & no more data written; prevents file from being corrupted }
```

$x_n = nh$

like std::cout arrows point towards cout

like std::cout in console output

variable read_file eof() is true when end of file reached

- NOTE: if don't know length use while(!read_file.eof())

```
std::ifstream read_file ("Output.dat");   // input stream variable read_file type std::ifstream
assert (read_file.is_open());  //asserts Output.dat is on disk in correct location otherwise assertion flagged & code is terminated
read_file >> x >> y;  // like std::cin - arrows point to input variable
read_file.close();
```

• 5.3;
- double & a represents address of variable (reference variable) can be used as normal variables inside fn when defined in fn parameter as &
& any changes inside fn will have effect outside fn ∴ can be used to return more than one variable effectively without converting to an address & without de-referencing inside the fn
- double * a stores address of variable
& = address in memory
pointer = variable whose value is address of another variable
e.g. double* p_a; → pointer to a double
double b;
1. define pointer variable
2. assign (store) address of variable to pointer
3. access value at address in pointer using op_a e.g.
⇒ p_a = &b; // p_a stores address of b
allocate memory p_a = new double; // to assign address to p_a (avoid storing contents in unspecified location in a computer's memory since it is hard to locate)
dealloc memory delete p_a;
*p_a = 1.0; // stores 1.0 in memory with address p_a
• 5.6;
- dynamically allocating memory avoids over/underestimating array & hence computational memory
- double * vector = new double [5]; // vector of length 5 memory allocation
(then used as normal vector e.g. vector [1] = 3.0)
delete [] vector; // remember to do this
- double ** Matrix;
Matrix = new double* [5]; // 5 rows
for (int i=0; i<5; i++)
{ Matrix [i] = new double [3]; // 3 columns }
(then used as normal matrix i.e. Matrix [i] [j])
new & double should match
for (int i=0; i<5; i++) → i< rows should match
{ delete[] Matrix [i]; }
delete [] Matrix;
- function overloading = same fn name but different fn prototypes & fns for both different operations then compiler checks correct fn based on input arguments

• 5.4;
- for (int i=0; i<length; i++)
{ sum += a[i]; } // get sum of array (sum = sum + a[i])

• 5.9;
- make copy of data given (good practice)
- Au = b ⇒ u = $A^{-1}$ b
$A^{-1} = \frac{1}{|A|} adj(A)$
adj(A) = transpose of cofactor
take 3 elements & 3 cofactors (including ±) sum
$\begin{vmatrix} + & - & + \\ - & + & - \\ + & - & + \end{vmatrix}$
eliminate row & col of element to get 2×2 to get value for 2×

• 5.10: row n & column k; find largest |Ank| so for each column, find row with largest absolute value
– loop through each m of n rows in for loop & replace largest value with current one until largest is found
→ initiate with fabs(X[k][k]) & int mmax = k;
then for (int m=k+1; m<n; m++) {
if (fabs(X[m][k]) > largest)
{then replace largest with fabs(X[m][k]) & mmax = m} // end of if loop
→ then swap row with |Ank| with current k row ~~with for~~ each column using typo variable (see tablet notes for clarification)
→ do this for the vector b as well
} // end of m in n for loop
→ then form upper triangular matrix
→ solve using last eqn in system
→ iterate backwards } iterate using a backwards for loop from N-1 to 0

1st column → swap row → row operation with multiple subtracted → for all rows > k → repeat for next column

Matrix[row][column]

by looping through

3. row operation after swapping
$\lambda_r = \frac{A_{rk}}{A_{kk}}$ & take away that multiple (to zero the rows below current kth row) $r = k+1, \ldots, N-1$
$A_{rc} = A_{rc} - \lambda_r A_{kc}$

m rows
$A_{00} \ldots A_{0N-1}$
$A_{N-1\,0}$
→ k = 0, ... N-1

1. find mmax row
2. swap mmax row with k row so in a for loop Amax,c = Akc for c = 0, ..., N-1

# Assignment 2

• 6.1.1 – 6.1.6

– Classes (supports object-oriented programming)
→ specify form of an object & combines data representation & methods (functions) for manipulating data
⇒ members of the class

– class ClassName
{ public: // members can be accessed from outside the class anywhere within scope of class object (outside class but within a program)
double a; // variable (public) }; ClassName Object; // declare Object of type ClassName
// access objects of a class in int main() using operator (.) e.g. Object.a = 7.0;

– class Name
{ public:
double method(); };
methods have access to all members of a class
remember to use class name before :: = scope resolution operator to define methods separately i.e. not within class definition

double Name::method() { }
→ declare return type first

private & protected members cannot be accessed directly using direct member access operator (.) without using a function (which would be declared inline)

– private members can only be accessed by class & friend fns
so class Name
{ private:
double b; };
→ then int main()
{ Name Object;
Object.b = 10.0; }
if not in class defn, needs to be stated explicitly before fn
Error: b is private

– protected members similar to private but can be accessed by child (derived) classes

e.g. class ~~Derived~~ Parent { protected:
double c; };

class Child : Parent // Child is the derived class
{ public:
double getC(); }; — methods in child class can access protected members in parent class

double Child::getC() { return c; } // method of child class

Then: int main() // main fn for program
{ Child derived; → derived is an object of Child
derived.getC(); } // so C can be obtained using operator(.) with method in child if c is protected (not private) in Parent

member fn is called using this operator for the object ↘

- constructor executed whenever new objects of that class is created
  → same name as class but no return type, not even void
  → useful for setting initial values for member variables
- default constructor does not have any parameters
  e.g. class Parent
  { public:
  Parent(); // default constructor };

  ← object of Parent created with no initial value

but can use parametrised constructor ~~e.g.~~ to assign initial value to an object when it is created
e.g. Parent(double d);
then int main() { Parent D(10.0); } → object of Parent created with initial value

→ with parametrised constructor, can use list to initialise fields:
Parent::Parent(double d); var(d) → : var(d) ≡ { var=d; }
≡ Parent::Parent(double d)
{ var = d; }
→ for multiple fields X, Y, Z etc. to be initialised in class C, use same syntax separated by commas e.g.
C::C(double a, double b, double c): X(a), Y(b), Z(c)
{ ... } like Student, GraduateStudent, PhdStudent

- destructor ~Parent(); executed when object class goes out of scope or delete expression applied to pointer to object in class
  → releasing memories, closing files etc.
  Parent::~Parent() { } → (var) class:: (function/constructor/destructor eh.)
- copy constructor creates object by initialising with object of same class that was previously created
  → initialise one object from another of same type
  → copy object to pass as argument to function
  → copy object to return it from a fn
  → needed if class has pointer variables & dynamic memory allocations

e.g. classname (const classname &obj) { //body of constructor }
→ obj is reference to object used to ~~initialise~~ initialise another object
→ in body copy from ~~def~~ normal constructor with ~~obj~~ variable = obj.variable

e.g. int main() { classname obj1;
classname obj2 = obj1; //calls copy constructor

- friend fns defined outside class' scope (so not member fns) but can access all private & protected members of class
→ declare using friend in front of fns within class defn
→ since friend fns are not a member fn of any class, do not need :: when defined separately

- this pointer to access objects own address; implicit parameter to all member fns (so only member fns have this)
→ used to refer to invoking object

- pointer to a class is accessed using member access operator →
→ ∴ need to ~~in~~ initialise pointer before using it

e.g. class Box
{ public:
double Volume();
};
int main() { Box Box1;
Box Box2;
Box *ptrBox; // declare pointer to a class
ptrBox = &Box1; //saves address of Box1
ptrBox → Volume(); //access a member using member access operator }

- 6.1.7: if matrix is diagonalisable then its exp. is the exp of each element along main diagonal (else it is just 0)

→ $e^x = \sum_{k=0}^{\infty} \frac{1}{k!} X^k = X^0 + X^1 + \sum_{k=2}^{\infty} \frac{1}{k!} X^k = 1 + X + \sum_{k=2}^{\infty} \frac{1}{k!} X^k$

→ $(a+ib)(c+id) = ac + ibc + iad - bd = (ac-bd) + i(ad+bc)$

e.g. $X = a+ib$

$$e^X = 1 + a+ib + \underbrace{\frac{(a+ib)^2}{2!} + \frac{(a+ib)^3}{3!} + \ldots}_{zSum}$$

$$= 1 + a+ib + \underbrace{(a+ib)}_{zPrev_0}\underbrace{\frac{(a+ib)}{2}}_{zNext_0} + \underbrace{\frac{(a+ib)(a+ib)}{2}}_{zPrev_1}\underbrace{\frac{(a+ib)}{3}}_{zNext_1} + \ldots$$

$zSum = zCurrent_1 + zCurrent_2 + \ldots$

↓ then this replaces zPrev

↓ then this replaces zPrev etc.

- matrix entries need to be of ComplexNumber type (like double, int etc.
- when testing, need to compile .o files ~~of Complex~~ to declare from running .cpp files of ComplexNumber & ComplexExponential as variable) well before running main.cpp to test the functions → need to like object files from other cpp files

• 7.1:
→ inherit members of an existing class (base class) & new class (derived class)
→ class Derived : public Parent, protected AnotherParent
(inherit members from more than one class separated by comma)
↳ access-specifier (can be public, protected, private, not specified then private used by default)

can access public & protected, not private but does not inherit fns

cannot inherit constructors, destructors, copy constructors, overloaded operators, friend constructors, destructors,

• 6.1.1 – 6.1.6:
– overloading operators (redefine built in operators) so can use operators with user-defined types as well (like ComplexNumber)
– unary operators (e.g. ++ (increment), decrement (--), minus (-), logical not (!)) operate on object for which they were called

e.g. to apply negation of object (so gives negative of all of the object's variables if defined in operator)

– binary operators take 2 arguments (+, -, /)
e.g. Name operator+(const Name &n)
{ Name name; (object) takes parameter of a referenced object of same class
name.a = this->a + n.a;
of a class return name; }

e.g. Box operator-() takes no parameters
{ length = -length;
return Box(length); }
needs to be declared as double len in constructor Box(double length) first

– relational operators (e.g. <, >, <=, >=, == etc.) to compare objects of a class e.g.
class Distance { bool operator<(const Distance &d)
{ if (feet < d.feet)
{ return True; }
} else { return False; } };

– input/output operators (stream extraction >> & insertion <<
e.g. → std::ostream &operator<<(std::ostream &output, const Distance &D)
defined in class Distance { output << D.feet << D.inches;
return output; // needed since std::ostream is a type }

– assignment operator (=) to create object just like copy constructor
e.g. → void operator=(const Distance &D)
in class Distance { feet = D.feet; }
then int main() { Distance D1, D2;
D1=D2; // use assignment operator }

– function call operator () to create operator fn that can be passed an arbitrary number of parameters
e.g. → Distance operator()(int a, int b, int c)
in class Distance { Distance D;
D.feet = a+b+c+10;
return D; }
then int main() { Distance D1, D2;
D2 = D1(10,10,10); // invoke operator() }

– subscript operator [] normally used to access array elements
e.g. → int &operator[](int i) // can only take one parameter unlike ()
defined in a class { // body of operator }

• 7.1
• polymorphism means call to member fn cause different fn to be executed depending on type of object that invokes fn.
→ e.g. using virtual in front of fns - where each child class has a separate implementation for that fn → in base class ∴ signals don't want static linkage for this fn
→ pure virtual if virtual fn = 0; in base class to tell compiler fn has no body
• data abstraction is the keeping implementation details separate from associated data
• class made abstract by ~~declaring~~ declaring ~~one~~ at least one fn as pure virtual, abstract classes serves ally as an interface, cannot be used to instantiate
→ subclass to be instantiate then has to implement all virtual fns of abstract meaning that it supports interface declared by abstract class
ones used to do that are called concrete classes
• coden notation ; MyClass var(arguments)
* MyClass var;
var = MyClass(arguments); } only brackets method using variables
• use list to initialise fields from parent class in derived class e.g.
Derived :: Derived(double var1, double var2) : Parent(var1, var2){}
• 8.2
→ fn returns same type as type of input parameter val
→ template <class type> → placeholder name for data type used by fn
→ "pattern" ~~compiler use~~ compiler uses to generate a family of classes or fns; to generate, compiler needs template defn & trying to make e.g. Class<double> at the same time
everything defined in header file
∴ if defn of template class is inside another .cpp (like defining fns in a .cpp separate from a header file) then compiler will not see template code & Class<double> at the same time
• 9.1
→ exceptions: try, catch & throw
→ throw: throws exception when problem shows up
→ catch: catches with exception handler at place in program where you want to handle problem
→ try: identifies block of code for exceptions, followed by one or more catch blocks
(multiple catch statements in catch different type of exceptions in case try block raises more than one exception)

Assignment 3

- template <class T> ← T stands for Type & instantiated specified when class D instantiated
  class Vector { }
  → Vector & becomes Vector<T>& since Vector in argument could be any type

- writing code indt of any particular type
  - friend fn D not a member of class ∴ cannot use T since that is fixed to class template
    ∴ need to define another template <class U> its friends
    & Vector & in argument of friend fn becomes Vector<U>& ⇒ so all instantiations of this template are

# Assignment 4

- STL (Standard Template Library) / template classes
  - → containers e.g. vector
  - → algorithms act on containers (perform initializing, sorting, searching & transforming contents of containers)
  - → iterators to step through elements of collections (containers or subsets of containers) of objects

→ std::vector<int> vec; //declare vectors of integers
- → vec.size() ⇒ display original size of vec
- → vec.push_back(i) ⇒ push values into vector (inserts value at end of vector, expanding its size
- → vec[i] ⇒ access vector
- → use iterator to access values:
  std::vector<int>::iterator v = vec.begin(); → returns an iterator to start of vector
  while (v != vec.end()) { *v; v++; }
  ↓ returns iterator to end of vector

vec.resize() to specify size

- iterators traverse a container class without user having to know how the container is implemented
  - → * ⇒ dereferencing iterator returns element iterator is currently pointing at
  - → ++ makes iterator to next element in container (-- move to previous element)
  - → = ⇒ assign iterator to new position; to assign value of element iterator is pointing at, dereference iterator first then use =
- → container::iterator ⇒ read/write iterator
  container::const_iterator ⇒ read-only iterator
- struct combine data items of different kinds
  - → access any member of a structure using member access operator (.)
- std::distance to get length of SetIterator
  - ↘ 2 parameters < 1. iterator pointing to first element
    2. iterator pointing to end of range
    ⇒ returns number of increments from 1st to last
- std::sort (first element, last element, comparison)
  - → e.g. std::sort(v.begin(), v.end(), [](int a, int b){return b<a;});
    - lamda expression for comparison
    - descending order
    - ↓ for ascending order a<b (like in assignment)
- distanceBegin to store results
- a vector to store distances & of same length as SetIterator
- a vector to access numbers in struct Pair that is same length of SetIterator & for each element in this vector initialise Pair struct of distance & iterator pointer ⇒ then sort using std::sort in ascending order

- store all sorted iterators in any type
- a vector length of k to store results
- then loop through all sorted iterators to get the 1st k elements