MA4: Parallellprogrammering och integration med C++

Denna uppgift handlar främst om parallellprogrammering, högre ordningen funktioner, och basal hantering av `git`, LINUX och C++. Uppgiften består av två huvuddelar:

1. Parallellprogrammering i Python
   - Använda Monte Carlo för att approximera π och volymen för en d-dimensionell hypersfär
   - List comprehension, Lambda-funktioner, `map`, `reduce`, `filter`, och `zip`
   - Plottning med `matplotlib`
   - Parallellprogrammering i Python
2. C++ integration i Python
   - `git`
   - LINUX
   - Skriva enkel C++ kod
   - Bygga en Pythonmodul skriven i C++
   - Jämförelse av exekveringstid Python vs C++ vs Numba

Redovisning MA4:

1. Alla delar (1.1, 1.2, 1.3 och 2.2) av uppgiften redovisas tillsammans muntlig vid ett tillfälle över Zoom på lektionstillfälle.
2. När muntlig redovisning är godkänd laddar ni upp filerna (med tydliga filnamn, och kom ihåg att skriva i filerna det datum när du godkändes muntligt, samt av vem) på STUDIUM. Alla filer enskilt, endast text-filer (.cpp, .py, ...) och bildfiler (t.ex. .png). Ingen zip, wordfil, pdf, Jupyter notebook, eller liknande.

Observera: Du får samarbeta med andra studenter, men du måste skriva och kunna förklara din egen kod. Du får inte kopiera eller skriva av kod vara sig från andra studenter eller från nätet förutom från de ställen som explicit pekas ut i denna lektion. Att byta variabelnamn och liknande modifikationer räknas inte som att skriva sin egen kod.
Eftersom uppgifterna ingår som moment i examinationen är vi skyldiga att anmäla brott mot dessa regler.

1 Parallellprogrammering i Python

I denna del av uppgiften ska du först skriva ett program som använder Monte Carlo för att beräkna en approximation av π. Sedan ska du modifiera detta program så att beräkningar sker parallellt. Därefter ska du återigen modifiera koden så att du beäknar volymen av en d-dimensionell hypersfär.

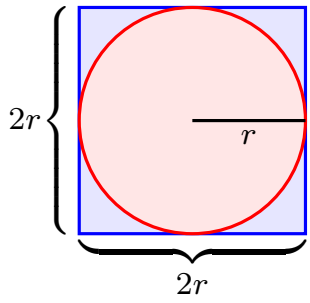
1.1 Monte Carlo beräkning av π

Monte Carlo-metoder (https://en.wikipedia.org/wiki/Monte_Carlo_method) är en stor och viktig klass av metoder som används som del av många olika algoritmer för att lösa problem numeriskt; främst inom optimering, integration och sannolikhetstoeri.

Man utnyttjar slummässiga tester för att få en uppskattning av det verkliga bakomliggande egenskapen. Detta är främst viktigt i multidimensionella problem där t.ex. integration in alla dimensioner kan vara omöjligt att göra exakt.

I denna del av uppgiften ska du skriva ett program som använder Monte Carlo för att uppskatta konstanten $\pi = 3.14159\ldots$

I Figur 1 ser du i rött en cirkel med radien r, och arean $A_c = \pi r^2$. Den är placerad i en blå kvadrat med sidorna $2r$, som har arean $A_k = (2r)^2 = 4r^2$.



Figur 1: Cirkel med radie r inskriven i kvadrat med sidor $2r$.

Om man dividerar arean A_c med A_k har du

$$\frac{A_c}{A_k} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4},$$

eller

$$\pi = 4\frac{A_c}{A_k}.$$

Antag nu att $r = 1$ och att centrum på cirkeln ligger i origo. Om man skapar n uniformt slumpvisa koordinater $(x, y) \in [-1, 1] \times [-1, 1]$ i kvadraten, och sorterar dessa i n_c punkter som ligger i cirkeln och n_k som ligger utanför cirkeln. Då kan man approximera

$$\pi \approx 4\frac{n_c}{n}.$$

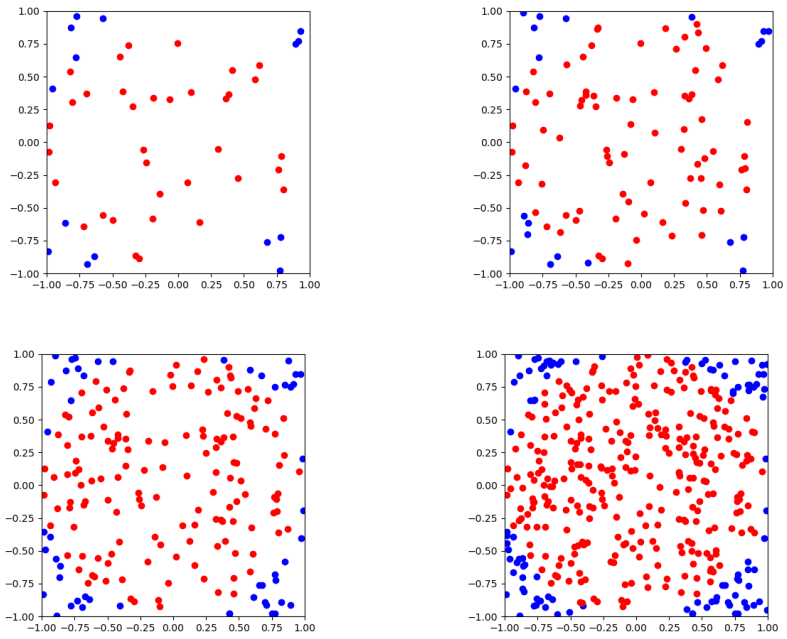
Då $n \to \infty$ kommer vi få ekvivalens. I Figur 2 visas fyra exempel, med $n = \{50, 100, 200, 400\}$. Approximationerna blir då $\pi \approx \{2.8, 3.16, 2.96, 2.97\}$. Notera att det kan hända att man får en bättre approximation för ett n som är lägre än ett annat.

Hur hittar du att en punkt är inne i cirkeln? I detta enkla fall, då $r = 1$ och centrum är i origon kan du helt enkelt testa om $x^2 + y^2 \leq 1$ (ekvivalent med $\sqrt{x^2 + y^2} \leq 1$). Hade du haft ett mer komplicerat objekt att räkna antalet koordinater som hamnar i det så kan man dela upp domänen i ett rutnät och i förhand definiera vilka rutor som räknas vara innuti och vilka som är utanför.

Skriv nu ett program som har n, d.v.s. antalet slumpvisa koordinater som ska testas, som inparameter och producerar följande output:

1. Skriv ut antalet punkter n_c som hamnar i cirkeln
2. Skriv ut approximationen av $\pi \approx 4n_c/n$
3. Skriv ut den inbyggda konstanten π (`math.pi`) i Python
4. Producera en `png`-fil som visar alla punkter inne i cirkeln som röda prickar och punkterna utanför cirkeln som blå prickar.

Tips: Använd `random.uniform()` för att skapa de n slumpvisa koordinaterna. Använd `matplotlib.pyplot` för att skapa figurerna (OBS, Ta inte en screenshot, utan se till att du skriver ut bilden till disk. Du behöver kunna göra detta i deluppgift 2.2).



Figur 2: Approximation av π med Monte Carlo. Övre: $n = 50, \pi \approx 2.8$ och $n = 100, \pi \approx 3.16$. Botten: $n = 200, \pi \approx 2.96$ och $n = 400, \pi \approx 2.97$.

Muntlig Redovisning MA4 1.1:

1. Presentera och förklara koden
2. Visa figurerna för $n = \{1000, 10000, 100000\}$
3. Redovisa vad approximationen av π blev för $n = \{1000, 10000, 100000\}$

1.2 Approximera volymen av en d-dimensionell hypersfär

- I denna uppgift ska du använda högre ordningen funktioner. Se separat pdf i STUDIUM (Modul M4) för en genomgång av dessa. Du ska använda minst tre av koncepten/funktionerna
  - list comprehension
  - Lambda-funktion
  - `map()`
  - `functools.reduce()`
  - `filter()`
  - `zip()`

  när du skriver koden till denna uppgift.

I denna deluppgift ska du skriva ett program som utför en Monte Carlo simulering som approximerar volymen $V_d(r)$ för en d-dimensionell hypersfär av radien r. Programmet ska ha två inparametrar, n som är antalet slumpvisa koordinater som ska testas och d är antalet dimensioner. Du kan anta, som i deluppgift 1.1, att $r = 1$ och centrum för hypersfären ligger i origo.

Testet för att se om en punkt ligger i hypersfären blir således i detta fall,

$$x_1^2 + x_2^2 + \ldots + x_d^2 \leq 1.$$

Formeln för volymen $V_d(r)$ (https://en.wikipedia.org/wiki/Volume_of_an_n-ball), för att verifiera din kod är

$$V_d(r) = \frac{\pi^{d/2}}{\Gamma(d/2+1)} r^d,$$

där Γ är Gamma-funktionen (https://en.wikipedia.org/wiki/Gamma_function). Gamma-funktionen finns inbyggd i Python i modulen `math`. Ni får gärna använda `numpy`.

Muntlig Redovisning MA4 1.2:

1. Presentera och förklara koden
2. Visa att du använt minst tre olika högre ordningens funktioner
3. Redovisa approximationen, samt det korrekta värdet för $V_d(1)$, med $(n, d) = (100000, 2)$, $(n, d) = (100000, 11)$.

1.3 Parallellprogrammering i Python

Läs separat pdf i STUDIUM (Modul M4) för en genomgång av parallellprogrammering.

Modifiera din kod i deluppgift 1.2 så att du kan köra parallella körningar, genom att använda `futures.ProcessPoolExecutor`. Tidtagning för redovisningen kan du göra med `time.perf_counter`.

Muntlig Redovisning MA4 1.3:

1. Kör tidtagning med koden från 1.2 med $(n, d) = (10000000, 11)$, samt koden i denna uppgift med $(n, d) = (1000000, 11)$ och 10 processer (så totala antalet samplingar n blir detsamma för de två körningarna)
2. Redovisa resultaten. Vilken var snabbast? Hur mycket snabbare och varför?

2 C++ integration i Python och Numba

Ren Python-kod är ofta för långsam för att vara praktiskt användbart. Då kan man skriva delar av koden i andra effektivare språk, t.ex. Fortran, C++ eller Julia, och anropa den från Python. Vi kommer här beskriva ett (av många) sätt att anropa C++ kod, och ni kommer undersöka prestandaskillnaden jämfört med ren Python-kod. Ni kommer även att prova att använda biblioteket *Numba* för att få snabbare Python-kod.

Som tidigare nämnts är Python ett interpreterat språk, man kör koden grenom en interpretator som exekverar rad för rad. C++ å andra sidan kompileras, dvs man använder en kompilator för att omvandla koden till en exekverbar binär. Varje gång man ändrar i koden så måste man kompilera om den innan man kan köra programmet.

2.1 Förberedelse

Denna deluppgift ska utföras på IT-institutionens linuxmaskiner, man kan dock utföra alla delar lokalt på era egna maskiner också. Vid redovisningen ska ni visa att ni utfört den på en av linuxmaskinerna.

2.1.1 IT-institutionens linuxmaskiner

Lista på maskiner på institutionen (välj en under Students):

https://www.it.uu.se/datordrift/maskinpark/linux

Logga in (med ett terminalprogram, macOS Terminal.app, windows PuTTY.exe eller installera Windows Subsystem for Linux, WSL, eller https://www.bitvise.com/) t.ex.:

```
$ ssh abcde123@arrhenius.it.uu.se
```

där `abcde123` är ditt UU-konto och använd lösenord A. Om det inte fungerar, kontakta `sven-erik.ekstrom@it.uu.se`. Notera att `$` inte ska skrivas, utan indikerar bara att det är en terminalprompt, och man ska endast skriva kommandot efter.

Här är en sammanfattning av några användbara linux-kommandon:

`ls`	Lista filer i nuvarande katalog (`ls -la` för att visa mer information)
`pwd`	Visa "var" du är (print working directory)
`cd abc`	Gå in i katalog med namn `abc`
`cd ..`	Gå "upp" en katalog i filträdet
`cd`	Gå till hemkatalogen (där dina filer finns, och där du är när du loggar in)
`mkdir abc`	Skapa en katalog som heter `abc`
`rm -fr abc`	Radera en katalog eller fil `abc`
`nano hej.txt`	Editera en fil `hej.txt` med editorn `nano` (det finns många andra editorer). Lämna nano med `ctrl-x`
`python3 test.py`	Kör pythonkoden i filen `test.py`

En tutorial som introduktion till linux (sök annars på nätet eller fråga lärare)

https://maker.pro/linux/tutorial/basic-linux-commands-for-beginners

Installera `matplotlib` i Python på ditt konto på linuxmaskinen. Om du inte vill använda något speciellt environment-verktyg på linuxmaskinen (Se föreläsning om environments under modul M0 i Studium) installerar du enklast med kommandot:

```
$ pip3 install matplotlib
```

OBS: detta görs på linuxmaskinen!

Om du vill prova allt detta lokalt behöver du även installera en C++ kompilator (t.ex. `g++` som är den del av `gcc`). Kompilatorn `gcc` är förinstallerad på linuxmaskinerna. På macOS installerar du XCode genom Appstore. I Windows kan du använda https://visualstudio.microsoft.com/. Notera dock att för redovisningen krävs att du gör denna del på linuxmaskinerna enligt ovan.

2.1.2 `git` konto

Se till att ha ett konto på någon `git` server. Se separat instruktion i Studium (Modul M0).

2.1.3 C++

Vi kommer i denna uppgift att använda `ctypes` för att kommunicera med C++-kod. Detta är en inbyggd modul i Python som egentligen är skriven för att kommunicera med kod skriven i språket C, men med en enkel modifikation fungerar det även för C++.

Antag att du skapat ett git-repositorium `prog2` och har det klonat lokalt på din dator.

Ladda nu ner `MA4_files.zip` från STUDIUM Uppgift OU4 (där du laddade ner denna pdf) och packa upp den i din lokala klon av repositoriet. (Vi antar här att den är `prog2/MA4/MA4_files`). Commita och pusha filerna till ditt repositorium, t.ex. i VSCode, eller i terminalen,

```
$ cd prog2/MA4
$ git add MA4_files
$ git commit -m "added files for MA4"
$ git push
```

Filerna i `MA4_files.zip` är:

- `integer.cpp` C++-koden för modulen `integer` med en klass `Integer`.
- `Makefile` Filen som berättar för kommandot `make` hur C++-koden i `integer.cpp` ska kompileras.
- `integer.py` Python-koden för modulen `integer`.
- `MA4_2.py` Ett test-program för att använda modulen `integer`.

Vi går nu igenom översiktligt dessa filers funktion i programmet.

integer.cpp

```
1  #include <cstdlib>
2  // Integer class
3
4  class Integer{
5          public:
6                  Integer(int);
7                  int get();
8                  void set(int);
9          private:
10                 int val;
11         };
12
13 Integer::Integer(int n){
14         val = n;
15         }
16
17 int Integer::get(){
18         return val;
19         }
20
21 void Integer::set(int n){
22         val = n;
23         }
```

- `#include <cstdlib>` på rad 1 betyder att definitioner i `cstdlib` inkluderas.
- Kommentarer i C++ görs med `//` (resten av raden är en kommentar) som på rad 2 eller `/* kommentar */` om man vill skriva över flera rader
- `Integer` (rad 4–11) är en klass med tre publika metoder (konstruktor, `get` och `set` på raderna 6–8) och ett privat värde (`val` på rad 10).
- Datatyper måste deklareras i C++ till skillnad från Python. I detta exempel har vi bara `int`, som står för integer (heltal). Andra exempel är `double` (flyttal) och `char` (text).

  - `Integer(int);` betyder att konstruktorn tar en `int` som argument.
  - `int get();` betyder att metoden `get` inte har några argument och returnerar en `int`
  - `void set(int);` betyder att metoden `set` har ett argument av typen `int` och returnerar ingenting, `void`.

- Notera att uttryck avslutas med `;`. Istället för att ta hänsyn till space/tab som i Python, används `{}` för att organisera koden i block.

I slutet på `integer.cpp` finns även följande

integer.cpp (cont.)

```
1  extern "C"{
2          Integer* Integer_new(int n) {return new Integer(n);}
3          int Integer_get(Integer* integer) {return integer->get();}
4          void Integer_set(Integer* integer, int n) {integer->set(n);}
5          void Integer_delete(Integer* integer){
6                  if (integer){
7                          delete integer;
8                          integer = nullptr;
9                          }
10                 }
11         }
```

Denna kod gör att modulen `ctypes` i Python kan anropa C++-koden, genom att det finns en "brygga" av C-funktioner. `Integer_delete` är en så kallad destruktor, som tar bort ett objekt.

Som nämnts innan måste C++-kod kompileras, till skillnad från Python-kod.

För att kompilera ovanstående kod (ska göras senare när filerna laddats upp till linuxmaskinen) ska det köras två kommandon

```
$ g++ -std=c++11 -c -fPIC integer.cpp -o integer.o
$ g++ -std=c++11 -shared -o libinteger.so  integer.o
```

Dessa kommandon skapar ett så kallat "shared object" `libinteger.so` som sedan Python kan importera. Kompilatorn på linuxmaskinerna är `gcc`, och `g++` är själva C++-kompilatorn.

Eftersom man efter varje gång man editerat `integer.cpp` måste köra bägge ovanstående kommandon kan det vara smidigt att använda en så kallad makefil, som finns med i `MA4_files.zip`

Makefile

```
# Makefile for MA4
all:
        g++ -std=c++11 -c -fPIC integer.cpp -o integer.o
        g++ -std=c++11 -shared -o libinteger.so  integer.o
clean:
        rm -fr *.o *.so __pycache__
```

Kommentarer i makefiler börjar ett #-tecken, och då blir resten av raden en kommentar, som på rad 1. Viktigt att det först är en <tab> på raderna som börjar med `g++` och `rm`. Istället för att skriva bägge `g++` kommandona, varje gång man vill kompilera sin kod, kan man nu skriva `make` i terminalen (samma sak som `make all`). Vill man radera "skräpfiler"

kan man skriva `make clean` (`rm` raderar filer, i detta fall alla som heter `*.o`, `*.so` och mappen `__pycache__`). Vill man lägga till andra standardiserade sekvenser av kommandon kan man bara skapa nya grupper av kommandon i `Makefile` (likt `all` och `clean`). Ett mer avancerat build-system av samma typ är `CMake` som på liknande sätt kan kompilera kod på alla plattformar.

Mer info för den intresserade om makefiler kan du hitta på t.ex. https://opensource.com/article/18/8/what-how-makefile och https://www.gnu.org/software/make/manual/make.html, och om Cmake hittar du på https://cmake.org/.

Själva Python-modulen (som du sen kan importera med `import integer` i andra Python-program) är

integer.py

```
1  """ Python interface to the C++ Integer class """
2  import ctypes
3  lib = ctypes.cdll.LoadLibrary('./libinteger.so')
4
5  class Integer(object):
6          def __init__(self, val):
7                  lib.Integer_new.argtypes = [ctypes.c_int]
8                  lib.Integer_new.restype = ctypes.c_void_p
9                  lib.Integer_get.argtypes = [ctypes.c_void_p]
10                 lib.Integer_get.restype = ctypes.c_int
11                 lib.Integer_set.argtypes = [ctypes.c_void_p,ctypes.c_int]
12                 lib.Integer_delete.argtypes = [ctypes.c_void_p]
13                 self.obj = lib.Integer_new(val)
14
15         def get(self):
16                 return lib.Integer_get(self.obj)
17
18         def set(self, val):
19                 lib.Integer_set(self.obj, val)
20
21         def __del__(self):
22                 return lib.Integer_delete(self.obj)
```

- Först på rad 2 importeras `ctypes` som är modulen som hanterar bryggningen till C (och C++). Det finns många sätt att göra detta, men `ctypes` är den som finns inbyggd i Python. Se t.ex. https://realpython.com/python-bindings-overview/ för andra alternativ.
- På rad 3 importeras C++-koden som kompilerats till shared object `libinteger.so`
- Från rad 5 har vi klassdefintionen av ett `Integer` i Python. I konstruktorn definieras argument typer (`argypes`) och returnerade typer (`restype`) för de olika funktionerna definierade i `extern "C"` delen av `helptal.cpp`. Typerna är här `ctypes.c_int` (`int`) och `ctypes.c_void_p` (`void`).
- På raderna 15 och 18 definieras Python-metoderna för `get()` och `set(val)`.

Sista filen i `MA4_files.zip` är exempelkod som använder Python-modulen `integer`.

MA4_2.py

```
1  #!/usr/bin/env python3
2
3  from integer import Integer
4
5  def main():
6          f = Integer(5)
7          print(f.get())
8          f.set(7)
9          print(f.get())
10
11 if __name__ == '__main__':
12         main()
```

- Första raden är en så kallad *shebang* som låter en köra koden direkt i linux som `$ ./MA4_2.py` (kan krävas att man gjort filen exekverbar genom `$ chmod 755 MA4_2.py` först). Man kan även köra med `$ python3 MA4_2.py`.
- På rad 3 importeras `Integer` från vår modul `integer`.
- På rad 6–7 skapas ett objekt `f` av typen `Integer` med värde 5, som printas.
- På rad 8–9 ändras värdet på `f` till 7, som printas.

Inloggad på en linuxmaskin skriver du nu

```
git clone https://github.com/ditt_konto/prog2
```

där du modifierar så att länken är ditt repositorium för kursen.

Skriv nu

```
$ cd prog2/MA4/MA4_files
$ ls
```

så ser du dina filer.

För att testa din kod kör du

```
$ make
$ python3 MA4_2.py
5
7
```

Du kan även testa följande

```
$ ./MA4_2.py
$ ls -la
$ chmod 755 MA4_2.py
$ ls -la
$ ./MA4_2.py
```

Första gången du kör `$ ./MA4_2.py` får du ett felmeddelande. Notera skillnaden mellan första och andra gången du exekverade `$ ls -la` för filen `ou4_2.py`. Kommandot `chmod`

ändrar rättigheterna för en fil, och 755 gör att filen blir exekverbar. När du gjort detta en gång kommer du kunna exekvera koden som `$ ./MA4_2.py` tills du ändrar rättigheterna till något annat.

2.2 Numba

Generellt kan Python-kod vara långsam att exekvera, men den är lätt att läsa (jämfört med t.ex. C++). Generellt måste kod vara förståelig för användare och andra utvecklar för att underhållas; om koden är svårförstådd kan det leda till att den blir utdaterad och obsolet. Speciellt för vetenskapliga applikationer behöver koden vara så enkel och expressiv som möjligt, så användare kan verifiera och bygga ut koden med ny funktionalitet. För att få både snabbhet och läsbarhetkan man snabba upp Python-kod utan att skriva t.ex. C++-kod (ibland kallat "the two language problem") finns speciella bibliotek. Ett sådant bibliotek är *Numba*, som ni kommer att använda för att snabba upp er Python-kod och sedan jämföra med generisk Python och C++.

Numba är sen så kallad "Just-In-Time" (JIT) kompilator. Den kompilerar kod när den körs (till skillnad från C++ där koden måste kompileras innan körning), dvs första gången en kod körs så kompileras den till maskinkod. Detta är speciellt användbar i lopar, då samma kod körs många gånger. Eftersom koden kompileras första gången den den stöts på, så kommer alla följande exekveringar att ske mycket snabbare än i standard Python.

För att kunna snabba upp Python-kod, så behöver Numba kunna göra vissa antaganden om koden, t.ex. om data typer (precis som i C++, till skillnad från vanlig Python som kör duck typing). Se detta exempel:

```
1  # Example of a dynamically written function
2  def maxOfList():
3      result = 'Nothing yet'
4      lst = [1, 8, 3, 9, 14]
5      for x in lst:
6          if result=='Nothing yet':
7              result = x
8          elif x > result:
9              result = x
10     return result
```

Denna kod funkar bra i Python, men kan ej köras i Numba. Kan du gissa varför? Det beror på att variabeln `result` kan vara olika typer i olika delar av koden; för är den en sträng (`str`) på rad 3, och sedan ett heltal (`int`) på raderna 7 och 9. Därmed blir jämförelsen på rad 8, `>`, inte väldefinierad (kan bli en jämförelse mellan ett heltaloch en sträng). Numba analyserar hela funktionen när den ska kompilera koden för att göra den snabbare, och här har vi alltså en tvetydighet som Numba inte accepterar. Numba är inte lika strikt som C++, som inte hanterar några konverteringar alls, men är mycket mer strikt än standard Python. För den intresserade, finns här exempel på fall då man måste vara extra noga med koden för att Numba ska kunna kompilera koden och generera snabbare kod https://numba.pydata.org/numba-doc/latest/user/troubleshoot.html. I den uppgift ni ska göra med Numba så är koden så enkel att ni inte behöver göra något speciellt för att det ska fungera.

2.2.1 Function decorators och installation

För att använda Numba måste man först installera Numba-biblioteket. Detta görs enklast med `pip3` på Linux-maskinerna, på samma sätt som ni tidigare installerade `matplotlib`. För att installera Numba skriver ni helt enkelt:

```
$ pip3 install numba
```

För att använda Numba å en Python-funktion, använder du en *funktions-dekorator* (function decorator). Dessa är funktioenr som modifierar andra funktioner (i detta fall snabbar upp dem genom kompilering, läs mer om dem online om du är intresserad).

Syntaxen är att lägga till `@decoratorName` ovanför funktions-defintionen. För att JIT-kompilera en funktion med hjälp av Numba så skriver man bara detta,

```
@njit
```

ovanför er funktions-defintion. Notera att ni måste ha importerat Numba i ert program:

```
from numba import njit
```

Exampel:

```
@njit
def someFunction(x):
    ...
```

Numba kommer då att JIT-kompilera den "dekorerade funktionen" första gången som den anropas.

2.3 Räkna ut Fibonacci-tal i Python, C++ och Numba

För att undersöka hr kompilerade språk (som C++) kan vara mer effektiva än interpreterade (som Python) ska du nu räkna ut Fibonacci-talen med båda. Du kommer även att använda Numba för att snabba upp Pythonkoden och se hur snabbt det blir.

Du ska nu modifiera koden du har på linuxmaskinen, så att du räknar ut Fibonacci-talen för givna n, dels med ren Python, Python med Numba och med C++ (genom att modifiera `integer` modulen). Använd rekursiv version av uträkningen,

```
def fib_py(n):
        if n <= 1:
                return n
        else:
                return(fib_py(n-1) + fib_py(n-2))
```

I uppgiften ska du även undersöka hur lång tid dessa tre olika funktioner tar för att beräkna olika Fibonacci-tal, samt plotta dessa resultat i en figur.

1. Du kan antingen använda de existerande filerna och bara lägga till funktionalitet, eller kopiera de existerande till annat ställe (använd t.ex. `cp`, `mkdir`, `mv`). Om du namnger filerna något annat, kom då ihåg att modifiera `Makefile` etc.

2. Använd nu en valfri editor i linuxterminalen (t.ex. `nano`, `vim` eller `emacs`). Man kan även använda VSCode:s remote edit funktionalitet (https://code.visualstudio.com/docs/remote/ssh). Enklast är `nano filnamn.txt`, då vanliga kommandon visas i terminalen; t.ex. Spara: `ctrl-o` Avsluta: `ctrl-x`.

3. Skapa en motsvarande C++-metod till Python-metoden `fib_py` ovan så följande fungerar i Python

```
f=Integer(n)
f.fib()
```

   Hint: Du behöver både skapa en publik och en privat metod i C++, och glöm inte att skriva lämplig bryggkod (både i C++-koden och i Python-modulen).

4. Kör tidtagning för `fib_py(n)` samt `f=Integer(n);f.fib()` för $n = 30, \ldots, 45$. Använd t.ex. `time.perf_counter`. Spara detta i en figur (x-axel är n och y-axel sekunder). Använd t.ex. `matplotlib.pyplot`.

5. Räkna ut Fibonacci-talet för $n = 47$ med C++-koden och Numba (för långsamt med ren Python).

6. Kom ihåg att använda `git` som backup av din kod

```
$ git add nyfil.txt  # lägg till en fil i gits indexering
$ git commit -m "la till nyfil.txt" # commita ändringar med ett meddelande
$ git push # skicka ändringar som är committade till repositoriet
$ git pull # hämta ändringar från repositoriet
$ git stash # spara undan det du gjort, men återgå till senaste oförändrade state
```

Muntlig Redovisning OU4 2.2:

1. Visa och beskriv din kod.
2. Logga in på en linuxmaskin och visa att du använt `git`. (t.ex., `ls -la` i katalogen med labbfiler).
3. Visa bilden med tidtagningen. Kommentera resultaten.
4. Vad får du för svar för Fibonacci med $n = 47$? Varför?