Fis Computacional 2021-22
2º semestre

A set of application projects and examples aiming to guide you through C++ learning on solving physics problems

Fernando Barão

April 2022

Histórico de alterações ao ficheiro

- 21 Abril, 2022:
  Homework ODE's: introduzido argumento tempo total da solução, nos métodos solver's
- 29 Março, 2022:
  Homework 3: introduzido o sistema de equações do circuito eléctrico
- 25 Março, 2022:
  Homework 1: A fórmula do coeficiente de correlação foi corrigida para ter em conta o facto da amostra se ir reduzindo à medida que k avança. Foi introduzido o factor,

$$\left(\frac{N}{N-k}\right)$$

- 11 Março, 2022:
  Homework 1: foi adicionado na alínea 1.) do problema a declaração da função que faz a leitura dos dados

Análise de séries de dados variáveis em tempo

Uma série temporal de dados corresponde a uma sequência ordenada de valores obtidos a intervalos regulares de tempo. A obtenção do sinal pode ser feita com intervallos de tempo igualmente espaçados ou não. Este tipo de dados podem ser obtidos em muitas áreas de Física mas não só; por exemplo, são quantidades variáveis no tempo o registo da actividade sísmica (geofísica), da actividade solar (astrofísica/partículas) ou ainda os sinais detectados pelos interferómetros na procura de ondas gravitacionais (astrofísica/gravitação).

A série temporal de dados pode resultar da adição de várias componentes harmónicas com amplitudes diversas e ainda de ruído instrumental. Pode, no entanto, ser mais complexa e incluir sinais periódicos de frequência variável e localizados num curto intervalo de tempo, como é o caso das ondas gravitacionais.

A análise directa da série temporal frequentemente não permite o conhecimento detalhado da sua composição. Torna-se assim necessário a decompôr a série num outro domínio como por exemplo a frequência, ou em casos mais complexos, em tempo e frequência.

Transformada de Fourier

A transformada discreta de Fourier (DFT) é uma ferramenta frequentemente usada em análise de sinais temporais [1]. Dado um sinal de $\theta(t)$ obtido a intervalos regulares de tempo (*sampling time* Δt_s) e durante um tempo total $T = N\Delta t_s$,

$$\{\theta_0, \theta_1, \theta_2, \theta_3, \theta_4, \cdots, \theta_{N-1}\}$$

a aplicação da ferramenta DFT permite a obtenção das frequências fundamentais (harmónicas f_k) que compôem o sinal.

Matematicamente, trata-se de obter a convolução da função em análise $\theta(t)$ com cada uma das harmónicas representadas sob a forma de uma função complexa, $h(t, f) = e^{-j2\pi ft} = \cos(2\pi ft) - j\sin(2\pi ft)$,

$$X(f) = \int_{-\infty}^{+\infty} \theta(t)\, h(t, f)\, dt = \int_{-\infty}^{+\infty} \theta(t)\, e^{-j2\pi ft}\, dt \tag{1}$$

O resultado $X(f)$, um número complexo, mede a importância do contínuo de harmónicas de frequência f na descrição de $\theta(t)$.

Uma vez que a variável em análise θ foi obtida para um conjunto discreto de valores, transformamos a expressão integral em somatório e obtemos,

$$X(f_k) = \sum_{n=0}^{N-1} \theta_n\, e^{-j2\pi f_k t_n} \tag{2}$$

$X(f_k)$ é um número complexo e corresponde aos coeficientes de Fourier que nos fornecem a amplitude e fase das diferentes componenetes harmónicas do sinal.

A frequência máxima detectável (f_k^c) depende da frequência de amostragem do sinal ($f_s = \frac{1}{\Delta t_s}$) e é obtida tendo em conta que a harmónica de período mínimo detectável corresponde a $T = 2\Delta t_s$. Resulta daí que $f_k^c = \frac{f_s}{2}$ ou seja, a frequência máxima da harmónica observável corresponde a metade da frequência de amostragem.

O conjunto de valores de frequência amostráveis ou identificáveis (f_k), para uma amostra com um número N par de medidas é dado por:

$$f_k = \frac{k}{N} f_s \quad (k = 0, 1, 2, \cdots, \frac{N}{2} - 1) \tag{3}$$

De notar que o passo do *varrimento* de frequência - a sua resolução ($\Delta f_k = \frac{f_s}{N}$), depende da dimensão da amostra N.

Tendo em conta que $t_n = n\Delta t_s$, obtém-se:

$$X(f_k) = \sum_{n=0}^{N-1} \theta_n \, e^{-j\frac{2\pi}{N}nk} \quad (k = 0, 1, 2, \cdots, \frac{N}{2} - 1) \tag{4}$$

De notar que $X(0)$ se relaciona com o ângulo médio da oscilação, $< \theta >= \frac{X(0)}{N}$.

A densidade de potência espectral ($|X|^2$) também designada como *Periodograma* da transformação, pode ser obtido da seguinte forma:

$$P(f_k) = \frac{1}{N} \left| \sum_{n=0}^{N-1} \theta_n \, e^{-j\frac{2\pi}{N}nk} \right|^2 \quad (k = 0, 1, 2, \cdots, \frac{N}{2} - 1)$$

Dados da série temporal

O ficheiro de dados existente neste link possui uma série temporal cuja amostragem foi feita a intervalos regulares de tempo, com $\Delta t = 0.01$ segundos. A figura seguinte mostra a função complexa e os dados amostrados, num pequeno intervalo de tempo.

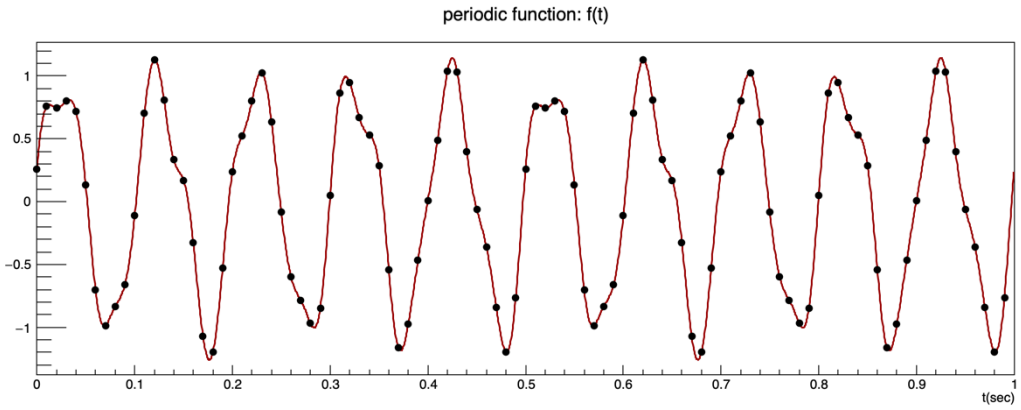

Figura 1

1. Realize um programa em C++ onde faça a leitura dos dados registados ao longo de 20 segundos, que se encontram no ficheiro `tDFT.dat`

   > Comece por realizar um programa principal onde fará a leitura dos dados para `arrays` de variáveis (por exemplo: **`double`** `t[N]`) onde possa colocar os valores lidos de tempo (t) e amplitude (A)

   > De seguida, estruture o código de leitura dos dados numa função autónoma cujo nome seja `ReadFile`. A declaração da função que se mostra de seguida e que deve ser colocada no ficheiro `src/ReadFile.h`, recebe do programa principal os `arrays` passados por referência.

   ```
   /*
   argumentos:
   fname = nome e localizacao do ficheiro
       t = array de valores tempo
       A = array de valores amplitude
   */
   void ReadFile(string fname, double (&t)[2000], double (&A)[2000]);
   ```

2. Divida os dados em blocos de 1 segundo e determine para o conjunto de blocos de tempo:

- a média,

$$< A >= \frac{1}{N}\sum_{t=0}^{T} A_t$$

- o desvio padrão (σ),

$$\sigma \simeq \sqrt{\frac{1}{N}\sum_{t=0}^{T}(A_t - \bar{A})^2} = \sqrt{\frac{1}{N}\sum_{t=0}^{T} A_t^2 - \left(\frac{1}{N}\sum_{t=0}^{T} A_t\right)^2}$$

3. O sinal possui eventualmente uma padrão de repetição temporal. Para o detectarmos e em termos contínuos, devemos realizar um estudo de correlação do sinal com ele próprio (autocorrelação) deslocado de um tempo t_0,

$$\rho(t') = \int_{-\infty}^{+\infty} f(t)\cdot f^*(t')dt \quad \Rightarrow \quad \rho(t_0) = \int_{-\infty}^{+\infty} f(t)\cdot f^*(t-t_0)dt$$

   Em termos discretos, a autocorrelação pode-se calcular como sendo:

$$\rho_k = \left(\frac{N}{N-k}\right)\frac{\sum_{t=k+1}^{T}\left(A_t - \bar{A}\right)\left(A_{t-k} - \bar{A}\right)}{\sum_{t=1}^{T}\left(A_t - \bar{A}\right)^2}$$

   N : número total de pontos de amostra
   k : desfasamento temporal ($t_0 = k\,t_s$)
   t_s : intervalo de tempo entre medidas do sinal

4. Realize a transformada de Fourier do sinal obtendo os coeficientes da transformação e ainda o Periodograma (densidade de potência espectral).
5. Substitua os `arrays` definidos no programa por elementos `vector`.
6. Realize a filtragem do sinal para a frequência mais elevada, utilizando a técnica da média deslizante.
7. Realize os seguintes plots usando o ROOT:

   - o histograma dos valores médios obtidos para os blocos temporais de 1 segundo
   - o histograma dos valores do desvio padrão obtidos para os blocos temporais de 1 segundo

   - o espectro de frequências: amplitudes $|X(f_k)|$ e Periodograma $P(f_k)$
   - o pontos do sinal conjuntamente com o sinal filtrado para um intervalo de 4 segundos

Mapas de luz

O projecto de iluminação de um espaço exige a simulação das fontes de luz e da geometria espacial. Uma etapa fundamental é o cálculo da iluminação produzida por uma fonte pontual de luz na superfície em estudo e de seguida integrar sobre todas as fontes.

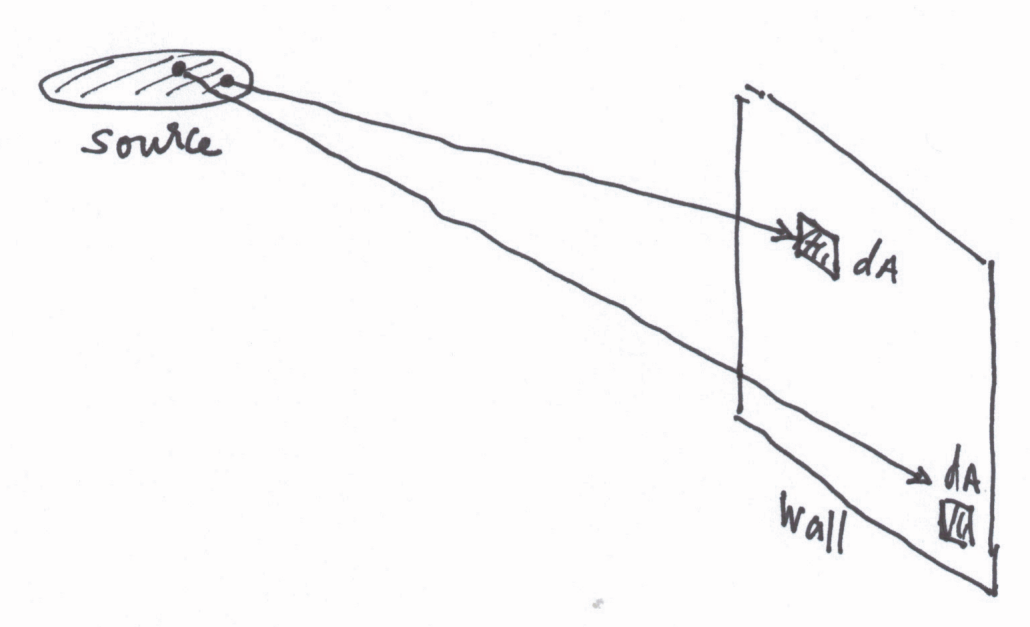

Figura 2

Fonte pontual isotrópica e superfície plana

Uma fonte de luz pontual encontra-se a uma distância $d = 1$ metro de um plano de dimensões 2×3 metros. O fluxo de energia por unidade de tempo, conhecido como **fluxo radiante**, emitido pela fonte (Φ) é de 100 Watts. Queremos produzir um mapa espacial que mostre a iluminação do plano.

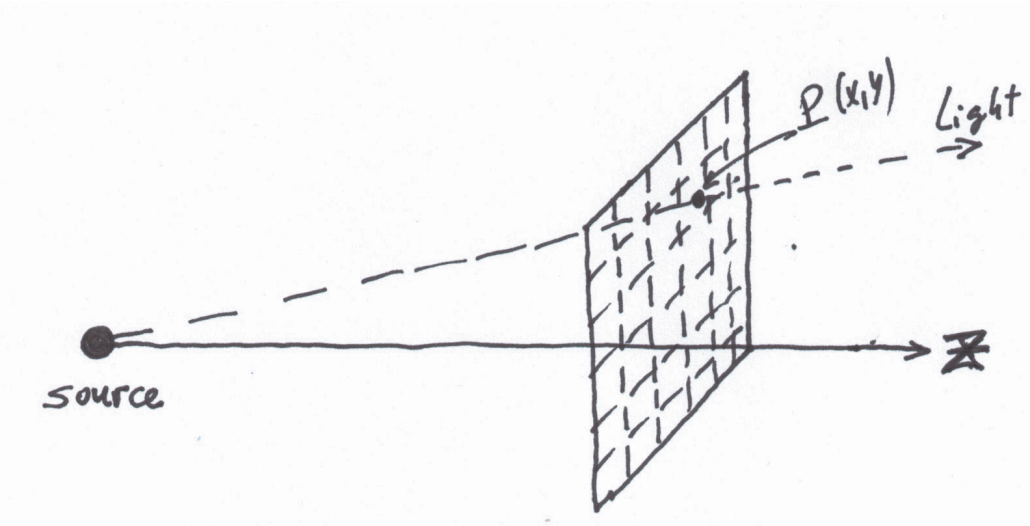

Figura 3

1. Determine a **Intensidade radiante**, I, da fonte, $I \equiv \frac{d\Phi}{d\Omega}$.
2. O plano pode ser representado por uma grelha de células (discretização do plano receptor), cada uma delas definidas pelo ponto central da célula, o vector normal à célula e a área da célula.

Defina uma estrutura `cell` para armazenar os dados da célula.

```
struct cell {
  float center_coo[3] = {0,0,100}; // cm
  float normal[3]; // unitary vector
  float area; // cm^2
  float power; // W
  ...
};
```

- Comece por definir um array bi-dimensional onde armazenar a grelha numérica.

  ```
  código simbólico: grid[ncellx][ncelly]
  ```

  Trabalhe com o array assim definido no programa principal, e preencha-o com as características de cada célula.

- Como faria o código C++ se a grelha numérica fosse preenchida no interior da seguinte função?

  ```
  void makegrid(cell**, int, int);
  ```

- E usando uma *lambda function*?

- Veremos na matéria teórica que o C++ possui maneiras mais simbólicas para criação da grelha usando o objecto `vector` da biblioteca STL. Por isso, seria interessante aqui construir também uma função `makegrid`, usando o "overloading" e que permita a construção da grelha de células.

3. Determine a **Irradiância**, E, que corresponde à potência por unidade de área recebida num ponto do plano receptor, $E = \frac{d\Phi}{dA}$

   - Elabora uma função em C++ que calcule a Irradiância para os pontos centrais das células

   > solução: $\frac{d\Phi}{dA} = \frac{\Phi}{4\pi} \frac{\cos\alpha}{r^2}$

4. Determine a potência média recebida em cada célula

   - Elabora uma função em C++ que calcule a potência média em cada célula

   > solução: $\frac{d\Phi}{dA}\Big|_i = \frac{\Phi}{4\pi} \frac{\cos\alpha_i}{r_i^2}$

5. Determine o mapa de iluminação do plano (potência em cada célula) e a potência total recebida pelo plano.

6. Biblioteca STL: organize as células num vector por ordem crescente da luminosidade

7. Implementar a classe `lightmap` para resolução do problema

```
class lightmap {
public:
 lightmap(int ncellx, int ncelly); // number of cells along x and y
 lightmap(const vector<float>& vx, const vector<float>& vy);

 pair<int,int> GetCellIndex(float x, float y); // return cell indices
 pair<float,float> GetCellCoo(int index_x, int index_y); // return cell center coo

 double GetCellPower(int index_x, int index_y); // return cell power Watts
 double GetCellPower(float x, float y); // return cell power Watts

 int GetCellNx(); // get number of cells along x
 int GetCellNy();

 const cell& GetMaxCell(); // get cell with max power

 std::vector<std::vector<cell> >& GetCells(); // return cells grid
```

```
 // (...) other methods you find useful: distance_to_cell(...), ...

private:

 vector<vector<cell> > GRID;

};
```

8. Representação gráfica do resultado

Representação gráfica com ROOT

Exemplo indicativo:

- programação orientada por objectos (objecto lightmap)
- com comentários sobre a utilização de ROOT para a representação gráfica do mapa de luz

For a better understanding check ROOT documentation:

- reference documentation: link
- histogram classes: link
- about colors in ROOT: link

```
#include "lightmap.h"

#include "TCanvas.h"  // include class TCanvas from ROOT library
#include "TRootCanvas.h"
#include "TH2F.h"    // histogram 2D
#include "TApplication.h"

int main() {

  // instantiate object lightmap

  lightmap L(...); // constructor arguments depend on your implementation

  // instantiate ROOT histogram object 2D where cell power will be placed
  // TH2F (float precision), TH2D (double precision)

  int nx = 200, ny=300;
  double size_x=200, size_y=300;
  auto h2 = new TH2F("h2", "mapa de luz", nx, 0, size_x, ny, 0, size_y);

  // now we have to fill every histogram cell with the calculated power

  for ( auto& v: L.GetCells() ) {
     for ( auto& c: v) {
        h2.Fill(c.center_coo[0], c.center_coo[1], c.power);
     }
  }

  // Draw
  // - we need to instatiate TApplication to have a graphics display
  // - produce a canvas (tela gráfica) where graphics objects will be placed
  // - draw histogram: check the many options you have available;
  //   here we choose a colored gradient representation
  // - save plot to file
  // - Run application

 TApplication app("app", nullptr, nullptr);
 auto c = new TCanvas("canvas", "lightmap canvas", 0, 0, 800, 800); // size 800x800
 h2->Draw("COLZ");
 c->Update(); // update display canvas
```

```
 c->SaveAs("lightmap.pdf"); // save graphics in pdf format (eps, png, ...)
 // this gives you control of graphics window buttons
 TRootCanvas *rc = (TRootCanvas *)c->GetCanvasImp();
 rc->Connect("CloseWindow()", "TApplication", gApplication, "Terminate()");
 // without the line that follows ( very last line of your program), nothing is
    displayed
 app.Run();

}
```

> Para compilar o programa necessitam de ter acesso às bibliotecas de ROOT e aos ficheiros *header* onde as classes de ROOT estão declaradas
>
> Na linha de comandos, para um programa tmap.C situado na pasta 'main/', fazer:

```
g++ -std=c++11 -o bin/main.exe main/tmap.C -I `root-config --incdir --libs`
```

> Para correr o programa:

```
./bin/tmap.exe
```

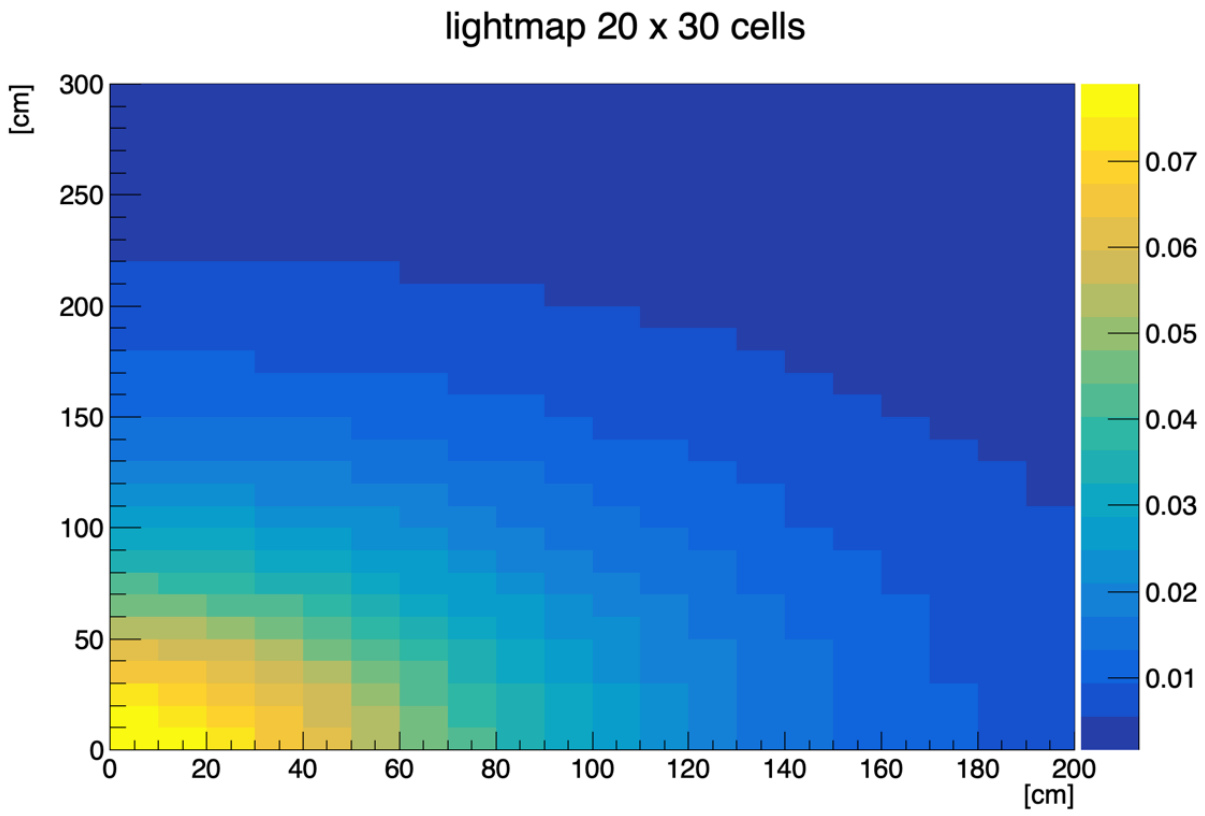

Figura 4: lightmap: potência em Watts para 20 células em x e 30 células em y

PGM file

A PGM file is a grayscale image file saved in the portable gray map (PGM) format and encoded with one or two bytes (8 or 16 bits) per pixel. It contains header information and a grid of numbers that represent different shades of gray from black (0) to white (up to 65,536). PGM files are typically stored in ASCII text format, but also have a binary representation.

PGM files include a header that defines the PGM format type ("P2" for text or "P5" for binary), image width and

height, and the maximum number of shades. While binary PGM files may contain multiple images, ASCII PGM files may only include one image.

The PGM format is one of several image formats defined by the Netpbm project, which is an open source package of graphics programs. Other formats include the portable bitmap (.PBM) format and portable pixmap (.PPM) format.

```
#include <fstream>

int main() {

 std::ofstream f("lightmap.pgm", std::ios::binary | std::ios::out);
 int maxColorValue = 255;

 // grid of cells [200][300]

 int width=200, height=300;
 f << "P2\n" << width << " " << height << "\n" << maxColorValue << "\n";

 // convert power to grayscale
 // - assign maximal power to 255
 // - assign zero power to 0
 int scale = (power/power_max)*maxColorValue;

 f << scale << " ";

 f.close();
}
```

Resolução de sistemas de equações lineares

Sistemas de equações lineares podem-se encontrar na resolução de problemas de estática de forças, na determinação dos pontos de equilíbrio de sistemas oscilantes ou por exemplo na determinação das correntes eléctricas nos diferentes ramos de um circuito eléctrico. Em qualquer dos casos, o problema pode ser equacionado através da seguinte equação matricial, $\mathbf{A} \cdot \mathbf{X} = \mathbf{b}$, onde:

> **A**, matriz de coeficientes ($n \times n$) cuja dimensão n depende do número de incógnitas do problema
>
> **X**, vector solução (n)
>
> **b**, vector de constantes do problema (n)

Determinação das correntes elétricas de um circuito

O circuito que se segue possui oito resistências alimentadas por dois potenciais eléctricos (baterias), -10 V e $+10$ V.

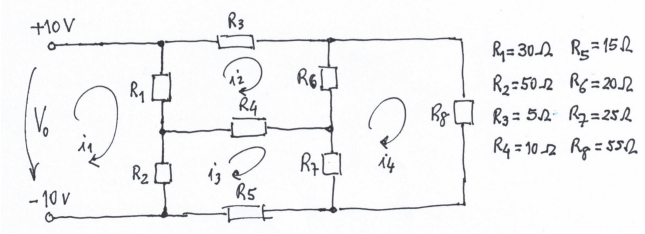

Figura 5: Circuito eléctrico

1. Estabeleça as equações do circuito usando as leis de Kirchoff, das malhas $\sum_i V_i = 0$ e dos nós $\sum_j I_j = 0$.

   solução

   A aplicação da lei das malhas, $\sum_k V_k = 0$, a cada uma das malhas do circuito, e do conhecimento da lei de Ohm, $V = Ri$, permite-nos obter o seguinte sistema de equações:

$$
\begin{aligned}
-V_0 + R_1(i_1 - i_2) + R_2(i_1 - i_3) &= 0 \\
R_1(i_2 - i_1) + R_3 i_2 + R_6(i_2 - i_4) + R_4(i_2 - i_3) &= 0 \\
R_2(i_3 - i_1) + R_4(i_3 - i_2) + R_7(i_3 - i_4) + R_5 i_3 &= 0 \\
R_6(i_4 - i_2) + R_8 i_4 + R_7(i_4 - i_3) &= 0
\end{aligned}
$$

2. Escreva as equações em termos matriciais, identificando a matriz **A** e os vectores **X** e **b**.
3. No sentido de da resolução do sistema de equações crie as classes necessárias (vá criando os métodos à medida que necessita).

   > - a biblioteca de matrizes Eigen; Eigen é uma biblioteca rápida para manipulação de matrizes.
   > - implementando a classe **FCmatrixAlgo**, onde os métodos (algoritmos) de redução das matrizes sejam construídos usando a biblioteca Eigen.

```
#include <Eigen/Core>

class FCmatrixAlgo {
  public:
  FCmatrixAlgo() = default; // compiler do it
  ~FCmatrixAlgo() = default;
```

```
  /*
  Implements Gauss elimination
  */
  static void GaussElimination(
              Eigen::Matrix<double,Eigen::Dynamic,Eigen::Dynamic>&, // matrix
                  coeffs
              Eigen::Matrix<double,Eigen::Dynamic,1>&        // vector of constants
              ); //no pivoting
  static void GaussEliminationPivot(
              Eigen::Matrix<double,Eigen::Dynamic,Eigen::Dynamic>&, // matrix coeff
              Eigen::Matrix<double,Eigen::Dynamic,1>&,       // vector of constants
              Eigen::Matrix<double,Eigen::Dynamic,1>&        // row order indexing
                ); //make pivoting

  /*
  Implements LU decomposition (Doolitle)
  */
  static void LUdecomposition(
              Eigen::Matrix<double,Eigen::Dynamic,Eigen::Dynamic>&, // matrix coeff
          Eigen::Matrix<int,Eigen::Dynamic,1>&, // row order indexing
          bool bpivot=false // activate pivoting
             );

}
```

> - implementando a classe **EqSolver** onde proceda à resolução do sistema de equações.

```
#include <Eigen/Dense>

class EqSolver {
  public:
  // constructors and destructor
  EqSolver();
  EqSolver(
           const Eigen::Matrix<double,Eigen::Dynamic,Eigen::Dynamic>&, // matrix
               coeffs
           const Eigen::Matrix<double,Eigen::Dynamic,1>& // vector of constants
       );
  ~EqSolver() = default;

  // output (optional)
  friend ostream& operator<<(ostream&, const EqSolver&);

  // solvers
  const Eigen::Matrix<double,Eigen::Dynamic,1>& GaussSolver(bool pivot=false);
  const Eigen::Matrix<double,Eigen::Dynamic,1>& LUSolver(bool pivot=false);
  void IterativeJacobiSolver(
       Eigen::Matrix<double,Eigen::Dynamic,1>&, // starting solution
       int& itmax,  //nb of max iterations
       double tol=1.E-3); // tolerance on convergence
  void IterativeGaussSeidelSolver(
       Eigen::Matrix<double,Eigen::Dynamic,1>&,
       int& itmax,
       double tol=1.E-3);

  private:
  Eigen::Matrix<double,Eigen::Dynamic,Eigen::Dynamic> M; // coefficients matrix
  Eigen::Matrix<double,Eigen::Dynamic,1> b; // constants vector
};
```

4. Resolva o sistema utilizando os diferentes métodos: Gauss, LU, iterativo.

> Exemplo de um programa principal **tEqSolver.C** onde se procede à resolução do sistema de equações.

```
#include "FCmatrixAlgo.h"
```

```
#include "EqSolver.h"
#include <iostream>

int main() {

  // define matrix equations: A.X=b

  (..) // create matrix A and vector of constants b

  // solution with Gauss (no pivoting)

  EqSolver S(A,b);
  auto X = S.GaussSolver();

  std::cout << "Gauss solution: \n" << X << std::endl;

}
```

5. Utilização de Makefile para compilação do programa principal e das classes necessárias à resolução do problema

   Nota: a versão de Makefile mais actualizada encontra-se na página da disciplina na secção aulas, com o nome `Makefile_libraries`

   Admitimos:
   - a existência do programa principal `tEqSolver.C` na pasta `main/`
     (ou .cpp, .cc, ... mas aí terão que adaptar o Makefile)
   - a existência da classe `FCamtrixAlgo` implementada nos ficheiros
     `FCmatrixAlgo.h` e `FCmatrixAlgo.C` na pasta `src/`
   - a existência da classe `EqSolver` implementada nos ficheiros
     `EqSolver.h` e `EqSolver.C` na pasta `src/`
   - os ficheiros binários `.o` e `.exe` serão colocados na pasta `bin/`

   Definimos o seguinte Makefile:

```
### definitions

CXX := g++
CXXFLAGS := $(shell root-config --cflags)

EIGEN_INC := $(shell pkg-config --cflags eigen3)

ROOT_INC := -I $(shell root-config --incdir)
ROOT_LIB := $(shell root-config --libs)

### rules (target: dependencies / actions)

VPATH = main:src

OBJS := lightmap.o FCmatrixAlgo.o EqSolver.o

bin/%.exe: bin/%.o $(OBJS)
     $(CXX)  $(CXXFLAGS) -o $@ $< -I src $(EIGEN_INC) $(ROOT_INC) $(OBJS) $(
         ROOT_LIB)

bin/%.o: %.C
     @echo compiling... [-o $@ $<]
     $(CXX) $(CXXFLAGS) -c -o $@ $< -I src $(EIGEN_INC) $(ROOT_INC)

clean:
     @echo deleting...[$(wildcard bin/*)]
     rm -f $(wildcard bin/*)
```

   Para produzirmos o executável, fazer:

```
make bin/tSolver.exe
```

Interpolação e integração numéricas

Pêndulo simples

A realização da experiência do pêndulo simples de massa $m = 500$ gramas e comprimento $L = 4$ metros em laboratório, permitiu obter as seguintes medições para as oscilações:

tempo (sec)	ângulo (rad)	velocidade (rad/sec)
0.71	0.806	-1.594
1.51	-0.702	-1.700
2.31	-1.395	0.065
3.11	-0.608	1.781
3.91	0.889	1.495
4.71	1.371	-0.347
5.51	0.391	-1.918
6.31	-1.051	-1.259
7.11	-1.314	0.626
7.91	-0.162	1.996
8.71	1.183	1.001
9.51	1.225	-0.900

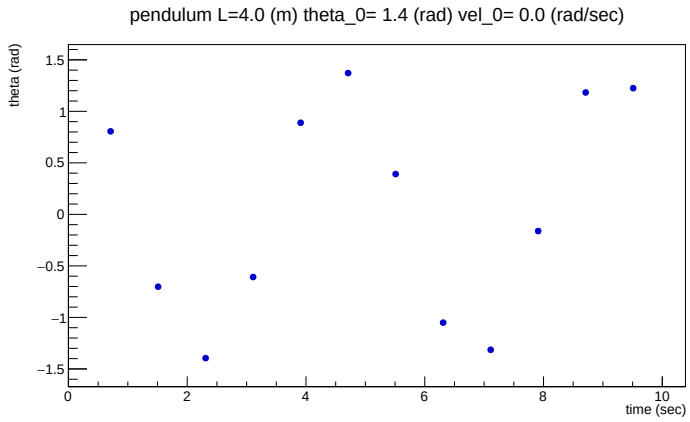

Figura 6: Valores registados para as oscilações do pêndulo simples com as condições iniciais: $\theta(t_0) = 80$ graus, $\dot{\theta}(t_0) = 0$

1. Realize uma função interpoladora da velocidade: $\frac{d\theta}{dt}(t)$. Para tal implemente a classe `DataPoints` que conterá os dados (tempo, velocidade) e a classe `Interpolator` que conterá os métodos de interpolação que necessite de trabalhar. De forma a poder aceder facilmente aos dados, implemente a classe `Interpolator` como herdando da classe `DataPoints`.

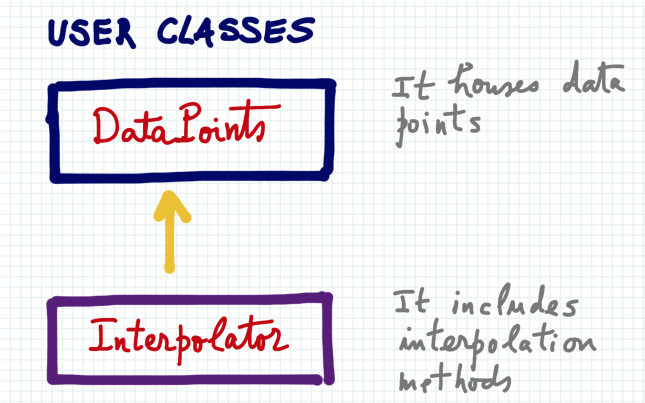

Figura 7: Esquema de implementação das classes `DataPoints` e `Interpolator`

Para começar a classe `DataPoints`, onde poderão implementar alguns dos seguintes constructores e métodos

```
class DataPoints {
public:

  // constructors, destructor

  DataPoints() = default; //default constructor (nothing to be done?)
  DataPoints(int N, double* x, double* y); // build DataPoints from C-arrays of x and y
      values
  DataPoints(const std::vector< std::pair<double,double> >&);
  DataPoints(const std::vector< double>& x, const std::vector< double>& y);
  virtual ~DataPoints();

  // getters

  const std::vector< std::pair<double,double> >& GetPoints();
  void GetGraph(TGraph&);

  // draw points using ROOT object TGraph

  virtual void Draw();

  // friend functions (optional)

  friend std::ostream& operator<< (std::ostream&, const DataPoints&);

protected:
  std::vector< std::pair<double,double> > P; // points
};
```

E de seguida a classe `Interpolator`:

```
class Interpolator : public DataPoints {
public:

  // constructors, destructor

  Interpolator() = default; //default constructor (nothing to be done?)
  Interpolator(int N, double* x, double* y); // build DataPoints from C-arrays of x and
      y values
  Interpolator(const std::vector< std::pair<double,double> >&);
  Interpolator(const std::vector< double>& x, const std::vector< double>& y);
  ~Interpolator();

  // interpolation methods

  double InterpolateLagrange(double); // Lagrange interpolation
  double InterpolateNewton(double); // Newton interpolation
```

```
  double InterpolateSpline3(doible); // spline3

  // draw points and function

  void Draw(std::string s); // s="Lagrange", "Neville", "Spline3"

  // getters

  const TF1& GetFunction(std::string s); // s="Neville", "Spline3"

private:
  std::map<std::string,TF1*> MI; // key="Lagrange", "Neville", "Spline3"
};
```

2. Determine a energia cinética do pêndulo, $E_k(t)$ e verifique a conservação da energia mecânica ao longo do tempo.

Integração de funções

Pretende-se integrar a seguinte função uni-dimensional entre os limites $[0, 2]$.

$$f(x) = x^4 \log(x + \sqrt{x^2 + 1})$$

Para tal propomos que use as classes `Functor`, classe de base das funções e a classe `MyFunction` onde deve definir a função integranda.

classe Functor

```
#ifndef __FUNCTOR__
#define __FUNCTOR__

#include <string>

#include "TCanvas.h"

class Functor {

 public:
  Functor(std::string s="Functor") : name(s) {;}
  ~Functor() = default;

  virtual double operator()(double x);

  // args:
  // xi, xf ..... xmin and xmax limits for function display
  // num ..... number of sampling points to be used o TGraph
  // xtitle, ytitle ... axis titles
  virtual void Draw(double xi, double xf, int num, std::string xtitle="x", std::string
      ytitle="y");

protected:
  static TCanvas *c;
  std::string name;
};

#endif
```

classe MyFunction

```
#ifndef __MYFUNCTION__
#define __MYFUNCTION__
```

```
#include "Functor.h"

class MyFunction : public Functor {

 public:
  MyFunction(std::string s="MyFunction") : Functor(s) {;}
  ~MyFunction() = default;

  double operator()(double x) {
     // implement here the function f(x)
     (...)
  }
};

#endif
```

1. Determine o integral da função com os métodos Trapezoidal e Simpson.

   A classe `IntegDeriv` deverá ser definida onde serão implementados os métodos de derivação e integração.

   classe Derivator/Integrator

```
#ifndef __INTEGDERIV__
#define __INTEGDERIV__

#include "Functor.h"

class IntegDeriv {

public:
 IntegDeriv(Functor&);
 ~IntegDeriv() = default;

 // integration methods

 void TrapezoidalRule(double xi, double xf, double& Integral, double& Error);
 void simpsonRule(double xi, double xf, double& Integral, double& Error);

 // derivative methods

 (...)

private:
  Functor& F;
};

#endif
```

Random walk e difusão

A difusão corresponde ao movimento aleatório de corpos num dado meio físico e em resultado da sua interação com o meio. Este fenómeno foi identificado pela primeira vez pelo botânico escocês Robert Brown no século 19, ao observar o movimento de partículas de pólen na água. Daí que este tipo de movimento aleatório tenha passado a designar-se como movimento **Browniano**. Este fenómeno permaneceu inexplicado...

O ano de 1905 é considerado um ano de referência para a Física. Porquê? Einstein deu contributos para três problemas abertos da Física e que exigiam uma visão radicalmente diferente. Concretamante, a explicação do movimento browniano, a introdução da dos quanta de luz e a Relatividade. A publicação de Einstein *"On the movement of small particles suspended in a stationary liquid demanded by the molecular-kinetic theory of heat"* refere o movimento browniano como uma marcha aleatória no espaço resultado das transferências de momento linear ($\Delta\vec{p}$) para as partículas de pólen em direções aleatórias, por parte das moléculas de água. Esta explicação foi pioneira e abriu definitivamente a via experimental para a explicação da matéria como sendo constituída por átomos e moléculas.

Random walk uni-dimensional (1D)

No sentido de extraírmos as características do movimento difusivo, vamos proceder à simulação a uma dimensão da marcha aleatória. Vamos considerar partículas que se podem deslocar ao longo do eixo do xx, partindo no instante $t = 0$, de $x = 0$ e que obedecem às seguintes regras:

- cada partícula pode deslocar-se para a direita ($-\delta$) ou para a esquerda ($+\delta$), em cada intervalo de tempo τ.
- a probabilidade em cada passo de a partícula se deslocar para a esquerda ou direita, é $1/2$.
- cada passo é independente do anterior (não há memória)
- cada partícula move-se de forma totalmente independente das outras

1. Comece por implementar a classe `Rwalk1D` que simula as trajectórias das partículas na marcha aleatória no método `Run`, armazenando as trajectórias das partículas num `std::map` **mT**

   declaração da classe Rwalk1D (ficheiro: src/Rwalk1D.h)

```
class Rwalk1D {

 public:

 Rwalk1D(int N=1, double x=0., // N=nb of particles, x=x(0)
       double pL=0.5, double pR=0.5, // probabilities Left Right
       double dt=1, double dx=1 // time and space steps
       );
 ~Rwalk1D();

 // particle simulation

 void Run(int nsteps); // number of steps

 // getters

 const std::vector<double>& GetTrajectory(int n=1); // particle number
 double GetTimeStep();
 double GetSpaceStep();

private:

 double x0; // init coo
 int N; // number of particles

 double pL, pR; // probabilities  (left, same, righ)
 double dt, dx; // steps (time, space)

 std::map<int, std::vector<double> > mT; // trajectories
```

```
};
```

2. Realize um programa principal `main/tRwalk.C` em que obtenha,

   a) a trajectória (x, t) de 5 partículas

   b) o valor médio do deslocamento de 200 partículas ao fim de 10 e 100 passos, $< x(n = 10, 100) >$.

   c) os histogramas das posições das 200 partículas ao fim de 10 e 100 passos.

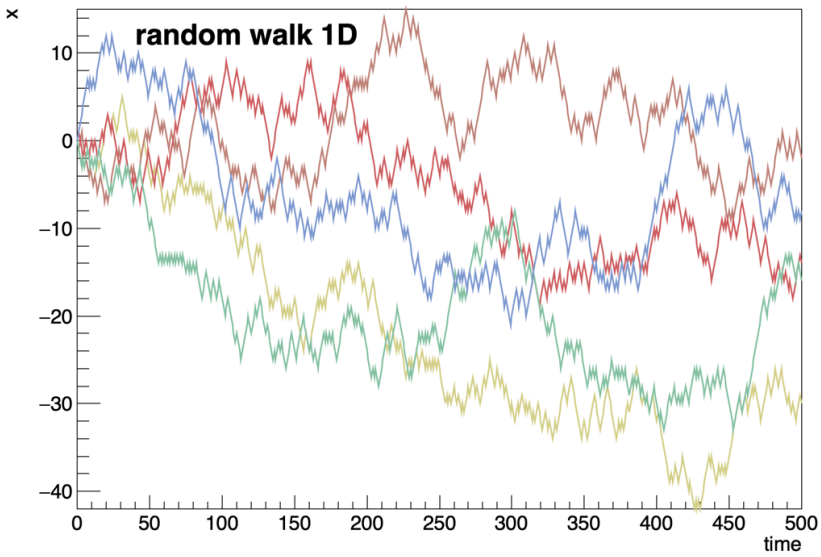

Figura 8: trajectórias $x(t)$ obtidas do random walk uni-dimensional para as cinco primeiras partículas

Integração de funções por métodos MC

Integre a função do problema 3. usando métodos monte-carlo.

ODE's (ordinary differential equations)

pêndulo gravítico

Consideremos um pêndulo simples gravítico sem presença de força de atrito, com um fio de comprimento $L = 4$ metros e uma massa $m = 500$ gramas na extremidade.

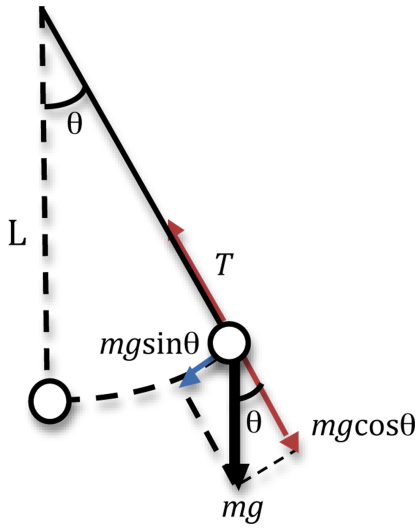

Figura 9: pêndulo simples

1. Determine a equação do movimento do pêndulo.
2. A equação do movimento do pêndulo é uma equação de segunda ordem (ordem da maior derivada) na coordenada angular (θ). Reduza esta equação a um sistema de duas equações diferenciais de primeira ordem.

   Implemente as funções como um array de *lambda functions* na classe solução do problema (*pendulum*)

```
#include <functional>

(...)

private:
  std::function<double(ODEpoint)> f[2];  // two functions, to be defined on
      constructor
```

3. Resolva numericamente a equação do movimento com os seguintes métodos:

   a) Euler
   b) Stormer-Verlet
   c) Runge-Kutta

   Para a resolução numérica implemente as classes **Xvar**, **ODEpoint** e **pendulum**. As primeiras duas classes (**Xvar** e **ODEpoint**) armazenam as variáveis independente (tempo) e dependentes (posição e velocidade). A classe **pendulum** deve possuir os diferentes métodos de resolução do problema.

   declaração da classe Xvar (ficheiro: src/ODEpoint.h)

```
class Xvar {
public:
Xvar() = default;
Xvar(int); // number of dependent variables
Xvar(std::vector<double>); // passing vector
// using initializer list to build object: X({1,2})
Xvar(const std::initializer_list<double>& v);
```

```
~Xvar();

Xvar(const X&); // copy constructor
Xvar& operator=(const Xvar&); // assignment operator
Xvar operator+(const Xvar&); // operator+
double& operator[](int);  // X[i]

friend Xvar operator*(double, const Xvar&); // scalar*X
friend std::ostream& operator<< (std::ostream&, const Xvar&);

std::vector<double>& X(); // accessor to x

protected:
std::vector<double> x;

};
```

declaração da classe ODEpoint (ficheiro: src/ODEpoint.h)

```
class ODEpoint : public Xvar {
private:
double t; // time

public:
ODEpoint() : t(-1) {;}
ODEpoint(double t_, Xvar a_) : t(t_), Xvar(a_) {;}
ODEpoint(double t_, const std::vector<double>& v) : t(t_), Xvar(v) {;}
ODEpoint(double t_, const std::initializer_list<double>& v) : t(t_), Xvar(v) {;}

void SetODEpoint(double t_, Xvar& p);
void SetODEpoint(double t_, const std::initializer_list<double>& v);
void SetODEpoint(double t_, std::vector<double> v);

double& T() { return t;} // accessor to time
};
```

declaração da classe pendulum (ficheiro: src/pendulum.h)

```
class pendulum {
public:

(...)

// solvers
// ... by default use method to solve for 20 seconds
const std::vector<ODEpoint>& StormerVerletSolver(double step=1E-3, double Time=20);
const std::vector<ODEpoint>& TrapezoidalSolver(double step=1E-3, double Time=20);
(...)

private:

double L; // length (m)
Xvar X0; // initial conditions: angle (rad), angular velocity (rad/s)

// solutions
std::map <std::string,  std::vector<ODEpoint> > MS; // key: "verlet", "euler", "
    trapezoidal"

// functions associated to dependent variables 1st order ODE's
std::function<double(ODEpoint)> f[2];
```

Appendix A: C++ classes: a brief introduction /clarification

what is a class?

A class is just the C++ implementation of a symbolic object we can think about. For instance, let's think about the `lightmap` object we have to build in order to solve our light map problem.

`lightmap` will be an object containing a numerical grid of the plane, because computationally the solution of the problem passes by making a discretization of the plane. So, we are going to split the plane in cells. Once that established, we need to build a C++ object that must represent the numerical grid of cells. That will be the `lightmap` object.

Think now about the `main` program and how we define a numerical grid object. We need to instantiate the object (create it!). Let's make a grid with 10×20 cells.

object instantiation

```
// main program
lightmap L(10,20);
```

Once the compiler finds this line, he is going to:

- check if this data type `lightmap` is defined (you need to have the declaration of your class on top on your main program)

  ```
  #include "lightmap.h"
  ```

- build a `lightmap` object named `L` having arguments `(10,20)`. To build an object you need to have defined the appropriate constructor function (or method) that in this case is declared as:

  ```
  // to be declared in the lightmap.h file
  lightmap(int,int);
  ```

- write the C++ code that makes the numerical grid in the file `lightmap.C`

  ```
  // file: lightmap.C

  lightmap::lightmap(int nx, int ny) {

    // we need to resize the grid to have nx x ny cells
    // note: variable grid was declared in lightmap.h as a vector<vector<cell>>

    grid.resize(nx);
    for (auto& v: grid) {
        v.resize(ny);
    }

    // now we have to fill the cell contents: coodinates, area, etc...
    (...)

    // and that's it! The constructor made the grid with the appropriate number of
        cells
    // and defined its contents

  }
  ```

Appendix B: notes about homework proposed C++ classes

Functor and functions definitions

The implementation of `operator()(double x)` method on a class makes it callable as a function $f(x)$. Identically, we could implement a $f(x,y)$ function by implementing `operator()(double x, double y)`, and so on. The base class Functor, can also have a `Draw` method to make the function plot. The `Draw` method can contain a ROOT `TGraph` object storing the function sampling at regular intervals and use the `TGraph.Draw("L")` method to display it (we use option "L" to have points connected by lines).

Inheritance

In addition, we use inheritance capability of C++ to allow any function re-implementation. I have defined as `virtual` in base class `Functor` the following methods: `operator()` and `Draw()`. Let's use the homework example to show it. We need to define $f(x) = x^4 \log(x + \sqrt{x^2+1})$ and I'm assung we have the `Functor` class in our library. I'm going to define a new functor class called `MyFunction`, that inherits from `Functor` and that will have the definition of $f(x)$. In addition function will be displayed using `Draw()` method of `Functor` class, because `MyFunction` has not redefined the `Draw()` method (only `operator()` was redefined).

program example with Functor function

Here is the program example:

main program (tMyFunction.C)

```
#include "Functor.h"

class MyFunction: public Functor {
public:
 MyFunction() : Functor("MyFunction") {};
 ~MyFunction() = default;

  double operator()(double x) {
    return pow(x,4)*log(x+sqrt(x*x+1));
  }

};

int main() {

  // create MyFunction object
  MyFunction F1;

  // create x vector with values: 0.0, 0.1, 0.2, ...
  vector<double> x(21,0); // create vector of 10 values filled with zero's
  double xi=0., step=0.1;
  transform(x.begin()+1, x.end(), x.begin()+1,
       [xi, step](double a) {
       static double buffer=xi; // buffer will keep its value between calls
       return buffer+=step;
       }); // 0.1 is the increment

  // calculate funtion value at x
  for (auto a: x) {
     cout << F1(a) << " " << flush;
  }

  // Draw function
  F1.Draw(0,2,1000); // using 1000 equally spaced values

}
```

TF1: ROOT general-purpose function

ROOT contains `TF1` as its general-purpose object for functions definitions. Check here its documentation and set of methods implemented.

Defining a TF1 function with a lambda function

Suppose you want to define a lambda function having n variables $x, y, z, t, v, \cdots$ and a set of parameters $p_0, p_1, p_2, \cdots$, that will be the input of a `TF1` object. That could be the way for instance for defining a function, $f(x, y, z) = a\,x^2 + b\,y^2 + c\,z^2$

```
auto l = [](double* x, double* p) {
         p[0]*x[0]*x[0] + p[1]*x[1]*x[1] + p[2]*x[2]*x[2];
};

auto f = new TF1("f", // all ROOT objects need a different name
                 l,  // lambda function
                 0., 2., // defined on integrval [0,2]
                 3, // number of parameters
                 3  // number of dimensions (variables)
                );
```

Defining a lambda function using a class method

This example aims to show how to deal with the problem of having a class method `InterpolateLagrange(double)` in class `Interpolator` that you need to call in order to make a `TF1` object.

For retrieving the method inside your lambda function, you need to have access to the object. That will be done through the **`this`** pointer, that points to current object and thus has access to all its members, either data or methods.

```
auto l = [this](double* x, double* p) {
         return this->InterpolateLagrange(x[0]);
};

auto f = new TF1("f", // all ROOT objects need a different name
                 l,  // lambda function
                 0., 2., // defined on interval [0,2]
                 0 // number of parameters
                );
```

Appendix C: software

C++ compiler

> check g++ version: `g++ --version`

ROOT

Instructions to install ROOT are on course web (Fenix) page.

> check if ROOT is installed: `root`

```
   ------------------------------------------------------------------
  | Welcome to ROOT 6.24/04                        https://root.cern |
  | (c) 1995-2021, The ROOT Team; conception: R. Brun, F. Rademakers |
  | Built for macosx64 on Aug 25 2021, 12:59:28                      |
  | From tags/v6-24-04@v6-24-04                                      |
  | With Apple clang version 12.0.5 (clang-1205.0.22.9)              |
  | Try '.help', '.demo', '.license', '.credits', '.quit'/'.q'       |
   ------------------------------------------------------------------

root [0]
```

> check your ROOT version: `root-config --version`

> check where ROOT libraries are: `root-config --libs`

```
-L/usr/local/Cellar/root/6.24.04/lib/root -lCore -lImt -lRIO -lNet -lHist -lGraf -
    lGraf3d -lGpad -lROOTVecOps -lTree -lTreePlayer -lRint -lPostscript -lMatrix -
    lPhysics -lMathCore -lThread -lMultiProc -lROOTDataFrame -stdlib=libc++ -lpthread -
    lm -ldl
```

> check where ROOT header files are: `root-config --incdir`

> check which compiler flags have been used while building ROOT: `root-config --ccflags`

Eigen

To install the **Eigen** library

> macOS: `brew install eigen`

You need to detect where include Eigen header files have been installed
1) detect where the library just installed is located: `brew info eigen`
2) catch the full path to arrive to Eigen/ directory and use it in the compilation process (see makefile)

> Ubuntu system: `sudo apt update && sudo apt install libeigen3-dev`

> CentOS 8 system: `sudo dnf update && sudo dnf groupinstall "Development Tools"`

You need to detect where include Eigen header files have been installed
1) detect where the library just installed is located
use `pkg-config`: `pkg-config --cflags eigen3`

Bibliografia

[1] D.J. Whitford, M.E.C. Vieira, J.K. Waters, Teaching time-series analysis. I. Finite Fourier analysis of ocean waves, Am. J. Phys. 69 (2001) 490–496.