Examen 02: Programación y métodos numéricos

William Oquendo, woquendo@gmail.com

1 Ejercicios

1.1 Matriz idempotente

((2.0/5.0) `idempotent.cpp`) Una matriz A es idempotente si se cumple que $A^2 = A$. Por ejemplo, la matriz

$$\begin{pmatrix} 3 & -6 \\ 1 & -2 \end{pmatrix} \tag{1}$$

es idempotente de forma exacta. La matriz identidad también lo es. En general, dada la precisión finita de los computadores, se puede extender la definición para indicar que una matriz es idempotente com precision ϵ si se cumple que $|A^2 - A| \leq \epsilon$, donde la norma $||$ se define comparando elemento a elemento con la precisión dada. Escriba un programa que tenga una función que reciba una matriz bidimensional de tamaño $n \times n$ (modelada con un `std::vector<double>` **unidimensional**) y una precisón `eps`, y retorne un booleano indicando si la matriz es idempotente o no, bajo esa precisión). Utilice el código base que se le ha entregado, sin cambiar el nombre del archivo:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <cmath>

bool is_idempotent(const std::vector<double> &  M, const double eps);

int main(int arg, char *argv[])
{
  const int N = 4;
  const double EPS = std::atof(argv[1]);
  std::vector<double> A = {1.99, -2.01, -4,
  -1, 2.999, 4,
  0.999, -2, -3.01};
  std::cout << is_idempotent(A, EPS) << std::endl;

  return 0;
}
bool is_idempotent(const std::vector<double> &  M, const double eps)
{
  fill here and delete this line
  // calcule A*A, puede crear una nueva matriz
  // haga la resta  A*A - A, para  ir comparando cada elemento.
  //   calcule la diferencia comparando con cero
  //   retorne true o false de acuerdo a lo anterior
}
```

Ejemplos de ejecución y salida:

```
./a.out 0.050
0
./a.out 0.080
1
./a.out 0.100
1
```

Puede hacer la verificación elemento por elemento: primero calcula A^2 (esto no es necesario pero le puede facilitar pensar el problema), luego hace la multiplicación y compara elemento a elemento con la matriz A, es decir, calcula $A^2 - A$. Para retornar `true`, cada elemento debe cumplir que su valor absoluto debe ser menor que `eps`. Recuerde que la multiplicación de matrices se define como

$$C_{ij} = \sum_k A_{ik} B_{kj}. \tag{2}$$

1.2 Roots y funciones lambda

(Entegrables: `(1.0/5.0) damping.cpp`, `(0.5/5.0) c_masses.pdf`, `(0.5/5.0) c_times.pdf`)

Cuando se modela el movimiento de un paracaidista de masa m bajo la acción de la graveded g, en el régimen de bajas velocidades, se obtiene que la rapidez en función del tiempo t es

$$v(t) = \frac{gm}{c}(1 - e^{-ct/m}), \tag{3}$$

donde c es el coeficiente de arrastre. Se desea calcular c para un valor fijo de la rapidez en un tiempo t. Es decir, dados los demás parámetros se busca resolver un problema de raíces, en donde se fija v (y g, m, t) y se encuentra el c correspondiente. Pero a la vez se busca encontrar cómo cambia esta raíz (valor de c) en función de la masa y del tiempo. Por ejemplo, se fijan todos los parámetros g, v, t, se da un valor de m y se calcula la raíz. Luego se da otro valor a m y se calcula la raíz, y así sucesivamente. Lo mismo con t: se fijan m, g, v, se da un valor de t y se calcula la raíz. Como se puede ver, la función de la raíz depende de varios parámetros pero la interfaz de nuestra función de búsqueda de raíces de Newton requiere una función que use solamente un parámetro ($f(x)$). Esto se puede resolver creando funciones auxiliares, functores, o, como se hace en este caso, funciones lambda. Debe completar el código `damping.cpp` para explorar el cambio en t, tomando como base lo implementado para el cambio en m. El código no debe imprimir nada a la pantalla, sino que debe imprimir a los archivos `data_masses.txt` y `data_times.txt`. Ejemplos de los contenidos de estos archivos en sus primeras líneas son

- `data_masses.txt`

```
5.000000000000000e+00   1.086720700713089e+00
1.000000000000000e+01   2.173441294446889e+00
2.000000000000000e+01   4.346882626099370e+00
3.000000000000000e+01   6.520324055390558e+00
```

- `data_times.txt`

```
5.000000000000000e+00   5.758476730672769e+00
6.000000000000000e+00   9.415855085932460e+00
7.000000000000000e+00   1.166773290070473e+01
8.000000000000000e+00   1.312957030052528e+01
```

Los tiempos toman los valores $5, 6, \ldots, 14, 15$. Las masas toman los valores que se muestran.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>

double f(double x, double m, double g, double t, double vf);
template <class fptr>
double newton(double x0, fptr fun, double eps, int & niter);

void explore_masses(void);
void explore_times(void);

int main(int argc, char *argv[]) {

  explore_masses();
  explore_times();

  return 0;
}

double f(double x, double m, double g, double t, double vf)
{
  return g*m*(1 - std::exp(-x*t/m))/x - vf;
}

// xi+1 = xi - f(xi)/f'(xi)
template <class fptr>
double newton(double x0, fptr fun, double eps, int & niter)
{
  double h = 0.001;
  double xr = x0;
  int iter = 1;
  while(std::fabs(fun(xr)) > eps) {
    double fderiv =  (fun(xr+h/2) - fun(xr-h/2))/h;
    xr = xr - fun(xr)/fderiv;
    iter++;
  }
```

```
  niter = iter;
  return xr;
}

void explore_masses(void)
{
  int nsteps = 0;
  std::vector<double> masses {5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
  std::ofstream fout("data_masses.txt");
  fout.precision(15); fout.setf(std::ios::scientific);
  for (auto m : masses) {                    /* g,  t,  vf */
    auto faux = [m](double x){ return f(x, m, 9.81, 10, 40); }; // lambda function captures m but exposes only one parameter x = c
    double root  = newton(10, faux, 1.0e-4, nsteps);
    fout << m << "\t" << root << std::endl;
  }
  fout.close();

}

void explore_times(void)
{
  fill here, delete this line
}
```

Debe realizar la gráfica de los datos en la herramienta que desee pero debe quedar guardada en el repositorio con los nombres que se indican al inicio de este ejercicio, en resolución apropiada, con labels en los ejes y demás. Por si acaso, las siguientes instrucciones de gnuplot le permiten generar la primera figura:

1.3 Números aleatorios

(`random.cpp` , 1.0/5.0) Escriba un programa que genere números aleatorios con la distribución de Weibull con parámetros $a = 1.0$ y $b = 2.0$, usando la librería estándar `random`. Recuerde que debe compilar con el estándar `c++11`. El `engine` debe ser Mersenne-Twister `mt19937`. Tanto la semilla como la cantidad de numeros aleatorios a imprimir se leerán desde la consola. Utilice como base el código a continuación. Esta plantilla también le muestra cómo escribir a un archivo desde c++ .

```
#include <fstream>
#include <cstdlib>
#include <random>
#include <string>

void print_random_numbers(int seed, int n, const std::string & fname);

int main(int argc, char *argv[])
{
  const int SEED = std::atoi(argv[1]); // read first arg
  const int N = std::atoi(argv[2]); // read second arg

  print_random_numbers(SEED, N, "datos.txt");

  return 0;
}

void print_random_numbers(int seed, int n, const std::string & fname)
{
  std::ofstream fout(fname, std::ios::out); // open file to write
  fout.precision(15); fout.setf(std::ios::scientific); // set format

  // fiil here
  // to print a number to the file, use it just as cout
  // fout << number << "\n";

  // close the file
  fout.close();
}
```

Por ejemplo, si el código compilado se ejecuta como `./a.out 10 1000` (semilla 10 y 1000 datos), las últimas 10 líneas del archivo `datos.txt` serán

```
3.491088986557811e+00
2.408636082814356e+00
6.296754142659176e-01
6.449891046699171e+00
1.275927765678437e+01
3.687202085039959e-02
9.334337694401843e-01
1.523113253048122e-01
7.717674762209574e-01
3.340195188045395e+00
```

El archivo `datos.txt` NO debe estar en el repositorio, ya que se genera a partir del cpp.