Curso de Programación en C++.
TPL0-2015. Trabajo Práctico de Laboratorio 0. [2015-05-13]

PASSWD PARA EL ZIP: VPY QRA BFL 3Y6

Ejercicios

[Ej. 1] **[interlace]** Dado un vector de vectores de enteros `vector<vector<int» vv` escribir una función `void interlace(vector<vector<int> > &vv,vector<int> &v);` que junta los elementos de los diferentes vectores de `vv` en forma **entrelazada**, es decir primero todos los primeros elementos de los `vv[k]`, es decir todos los `vv[k][0]` después todos los `vv[k][1]` y así siguiendo. Los `vv[k]` pueden tener diferente longitud. En ese caso la función continúa como si se hubiera eliminado el vector correspondiente. Ejemplo:

```
vv[0] = [2 3 4 5]
vv[1] = [3 4 5 6 7]
vv[2] = [4 5 6 7 8 9]
```

debe quedar: `v=[2 3 4 3 4 5 4 5 6 5 6 7 7 8 9]`

Ayuda: Se sugiere el siguiente algoritmo. Para cada `j` empezando en 0 recorrer todos los vectores a ver si tienen al menos uno de ellos tiene la longitud apropiada (`j+1`). Si no se detiene. Si no es así recorre los vectores y de aquellos que tienen elemento `j` va insertando sus elementos en `v`.

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado `program.cpp`. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si `f` es la función a evaluar tenemos

  ```
  ev.eval1(interlace,vrbs);
  ```

  La primera `ev.eval1(interlace,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario `f` y compara la salida del usuario (`user`) con respecto a la esperada (`ref`). Si la verbosidad (el argumento `vrbs`) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

mstorti@galileo/texstuff-1.4.0-68-g058255cf/Tue Jun 22 09:09:53 2021 -0300

Curso de Programación en C++.
TPL1. Trabajo Práctico de Laboratorio 1. [2015-05-20]

PASSWD PARA EL ZIP: `88I J4P P7H JKC`

Ejercicios

[Ej. 1] **[countsets]** Dado un vector de vectores `vector<vector<int> > &VV` escribir una función `int countsets(vector< vector<int> > &VV,int x);` que retorna la cantidad de vectores en los cuales `x` está presente al menos una vez. Por ejemplo si

```
VV[0]=[3 4 1 4]
VV[1]=[1]
VV[2]=[4 1]
VV[3]=[1 3 3 0]
VV[4]=[3 4 0]
VV[5]=[]
VV[6]=[]
VV[7]=[2 3 3 2]
VV[8]=[1 0 4 0]
VV[9]=[0 2 2 0]
```

entonces `countsets(VV,4)` debe retornar 4 ya que el número `x=4` aparece en los vectores `VV[0], VV[2], VV[4], VV[8]`.

[Ej. 2] **[max-seq]** Dado un vector de enteros `w` y un entero `m` calcular todas las sumas de las subsecuencias de longitud `m` y retornar el máximo de ellas. Por ejemplo, si `w=[1 3 4 1 2 3 0 2 5 5 2 2]` y `m=5`, entonces las subsecuencias de 5 elementos son `[1 3 4 1 2]`, `[3 4 1 2 3]`, ..., `[2 5 5 2 2]`. De las cuales, las sumas son 11, 13, ..., 16. Por lo tanto la función debe retornar el máximo que es 16.
Consigna: Escribir una función `int max_mseq(vector<int> &w,int m);`

[Ej. 3] **[badpwd]** Bob es perezoso y cuando el sistema le pide que cambie el password simplemente permuta las sílabas del mismo. Por ejemplo si el password era **bazinga** tratará de usar **zingaba** o **gabazin**. La administradora del sistema Alice, quiere evitar esto, ya que si un individuo malicioso tuvo acceso al viejo password podrá probar con un reducido número de combinaciones y deducir el nuevo. Sin embargo tampoco quiere prohibirle permutaciones arbitrarias del viejo password ya que sería demasiado restrictivo. Después de mucho cavilar decide no permitir passwords que sean permutaciones de hasta 3 substrings del viejo password. Es decir si el viejo password es **bazinga**, entonces **gazinba** no es aceptable ya que se puede obtener dividiendo a **bazinga** en **ba|zin|ga** y permutar sus componentes. Pero si es permitido **agnizab** (**bazinga** al revés).

Por lo tanto Alice solicita que escribas una función `int badpwd(string &oldpwd,string &newpwd);` que retorna **1** si **newpwd** se puede obtener de **oldpwd** dividiéndolo en a lo sumo 3 substrings y permutándolos y **0** caso contrario.

Ayuda:

- Sea `N` la longitud de `oldpwd`, entonces recorrer todas las posibles tripletas de enteros `n1,n2,n3` tales que `N==n1+n2+n3` (`0<=n1,n2,n3<=N`).
- Extraer los substrings correspondientes de `oldpwd` y realizar las permutaciones correspondientes.
- Si `s1,s2,s3` son los substrings, las 6 posibles permutaciones son

```
s1+s2+s3, s1+s3+s2,
s2+s1+s3, s2+s3+s1,
s3+s1+s2, s3+s2+s1
```

- Para reocorrer las tripletas mencionadas `n1,n2,n3` lo más fácil es hacer un lazo triple anidado sobre los 3 índices entre `[0,N]` y chequear si la tripleta satisface la condición `N==n1+n2+n3`.

[Ej. 4] **[larger-odd-subvec]** Dado un vector de vectores de enteros `VV` determinar para cada vector `VV[j]` la subsecuencia de elementos impares `odd(VV[j])`, y de todos ellas aquella que tiene longitud máxima `oddmax`.

```
void larger_odd_subvec(vector< vector<int> > &VV,vector<int> &oddmax);
```

```
VV[0]= [3 8 7 4 7 4 4 1 3 4 6],          odd(VV[0])= [3 7 7 1 3]
VV[1]= [4 1 8 0 6 5 1 4 8 4 4],          odd(VV[1])= [1 5 1]
VV[2]= [7 2 7 6 1 3 1 0 4 1 3 7 3 8 3],  odd(VV[2])= [7 7 1 3 1 1 3 7 3 3]

oddmax = odd(VV[2]) = [7 7 1 3 1 1 3 7 3 3]
```

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado `program.cpp`. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si `f` es la función a evaluar tenemos

```
ev.eval1(f,vrbs);
```

`ev.eval1(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario `f` y compara la salida del usuario (`user`) con respecto a la esperada (`ref`). Si la verbosidad (el argumento `vrbs`) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

mstorti@galileo/texstuff-1.2.8-61-g36ebe6a/Wed May 20 14:48:14 2015 -0300

Curso de Programación en C++.
TPL2-2015. Trabajo Práctico de Laboratorio 2. [2015-06-17]

PASSWD PARA EL ZIP: `67P 1LY IPM K1N`

Ejercicios

[Ej. 1] **[ppt-game]** Consideremos el jugo de **piedra, papel, tijera** (**ppt**). Los jugadores se basan en la siguiente clase

```
class ppt_player_t {
  public:
  virtual int play(vector<int> &you,vector<int> &me)=0;
};
```

- Las jugadas son un entero representando 0=piedra, 1=papel, 2=tijera. Dada una jugada `j` la ganadora corriente es `(j+1)%3` y la perdedora es `(j+2)%3`.
- El jugador recibe la historia de jugadas del oponente `vector<int> you` y propias `vector<int> me`. Debe devolver su nueva jugada en base a estas dos o puede ser aleatoria. Escribir las siguientes implementaciones
  - `random_t`: juega random (usar la función `rand()` de `<cstdlib>`
  - `fixed_t`: juega siempre lo mismo (piedra por ejemplo).
  - `repeater_t`: piensa (ingenuamente) que es mejor no jugar la jugada anterior. Es decir si en la jugada anterior jugó piedra, entonces en la siguiente elegirá random entre papel y tijera.
  - `repeater_killer_t`: Si el oponente es un `repeater_t` entonces puede basarse en su anterior jugada para mejorar su estrategia. Por ejemplo, si el oponente P1 jugó `j1=0` (piedra), entonces en la siguiente P2 sabe que jugará 1=papel o 2=tijera, por lo tanto le conviene jugar 2=tijera, ya que con esa jugada gana o empata. Es decir la jugada ganadora para P2 es `(j1+2)%3`.
- Implementar una función

  ```
  vector<int>
  ppt_game(ppt_player_t &p1,ppt_player_t &p2,int Nrounds=1000);
  ```

  que toma dos jugadores `p1` y `p2` y los enfrenta en `Nrounds` rondas y devuelve un vector de 3 enteros que contabilizan las veces que ganó `p1`, `p2` y empates. `ppt_game()` además debe mantener la historia de las jugadas de `p1` y `p2` y pasárselas apropiadamente `(his2,his1)` a `p1` y `(his1,his2)` a `p2`.
- Enfrentar jugadores de diferente tipo y verificar los siguientes resultados (los porcentajes son: gana P1, gana P2, empate):

  ```
  random vs. random: 33% 33% 33%
  repeater vs. repeater: 33% 33% 33%
  repeater vs. repeater_killer: 0% 50% 50%
  repeater_killer vs. repeater_killer: 0% 0% 100%
  ```

[Ej. 2] **[map2freq]** Dado una lista de conjuntos, encontrar la frecuencia con la cual un elemento dado aparece en los conjuntos (es decir la cantidad de conjuntos a los cuales pertenece).

`void map2freq(list< set<int> > &LS, map<int,int> &freq);`
La correspondencia **freq** es tal que **freq[x]** es la cantidad de conjuntos en los cuales **x** aparece. Por ejemplo

```
LS[0]: 2 3 4
LS[1]: 1 4
LS[2]: 2 4
LS[3]: 2 3
LS[4]: 0 4
LS[5]:
LS[6]: 0
freq:
M[0] = 2
M[1] = 1
M[2] = 3
M[3] = 2
M[4] = 4
```

[Ej. 3] **[is-perm-fun]** Dado un conjunto **S** y una función de mapeo **map_fun_t f** determinar si el conjunto **fS** resultado de aplicar **f** a los elementos de **S** es una permutación de los mismos, es decir si **fS==S**. Por ejemplo si **S={0,1,2,3...,9}** entonces la función **int f(x) { return 9-x; }** es una permutación ya que genera los elementos **fS={9,8,...,0}** que es igual a **S**.

```
typedef int (*map_fun_t)(int k);
bool ispermfun(set<int> &S,map_fun_t f);
```

Los conjuntos utilizados en los ejemplos tienen todos elementos en **[0,10)** y las funciones de mapeo son:

```
const int N1=10;
typedef int (*map_fun_t)(int k);
int f1(int x) { return N1-x-1; }
int id(int x) { return x; }
int shift2(int x) { return (x+2)%N1; }
```

[Ej. 4] **[filt-by-img]** Dada una correspondencia (**map<int,int> M**) hacer una copia de la misma con sólo aquellos pares de asignación para los cuales la **imagen** satisface un cierto predicado
`bool pred(int x);`
`void filtbyimg(map<int,int> &M,map<int,int> &M2,pred_t pred);`
Por ejemplo, si el predicado es **even()** entonces debemos tener:

```
M[0] = 4
M[5] = -2
M[6] = 1
M[7] = 4
M[8] = 1
M[9] = 2
predicate: even ->
M2[0] = 4
```

```
M2[5] = -2
M2[7] = 4
M2[9] = 2
```

Los predicados utilizados en los ejemplos son:

```
typedef bool (*pred_t)(int x);
bool even(int x) { return x%2==0; }
bool odd(int x) { return x%2>0; }
bool positive(int x) { return x>0; }
```

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado **`program.cpp`**. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si **`f`** es la función a evaluar tenemos

  ```
  ev.eval1(f,vrbs);
  ```

  `ev.eval1(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario **`f`** y compara la salida del usuario (**`user`**) con respecto a la esperada (**`ref`**). Si la verbosidad (el argumento **`vrbs`**) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

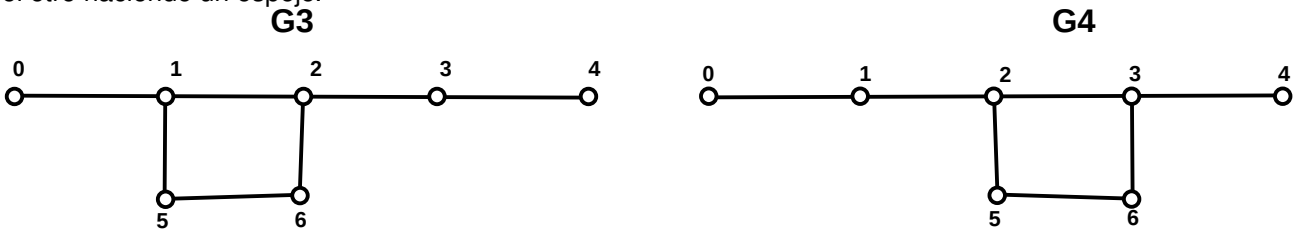
mstorti@galileo/texstuff-1.4.0-68-g058255cf/Tue Jun 22 10:34:49 2021 -0300

Curso de Programación en C++.
TPL3-2015. Trabajo Práctico de Laboratorio 3. [2015-08-12]

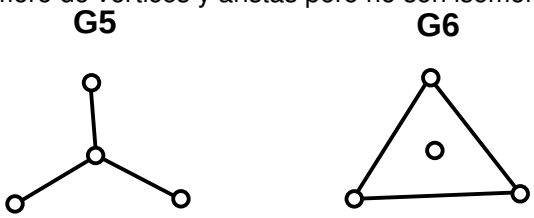
PASSWD PARA EL ZIP: MWC 48U 4E5 EN2

Ejercicios

[Ej. 1] **[isomorph]** Dos grafos `G1`, `G2` son isomorfos si existe una renumeración de los índices de los vértices tal que lleva `G1` a `G2`. Por ejemplo los dos grafos `G3` y `G4` de la figura son isomorfos, ya que uno se obtiene del otro haciendo un espejo.



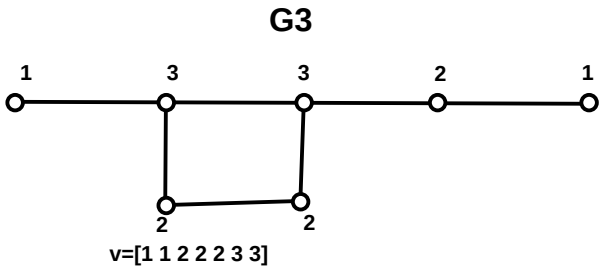
Dados dos grafos `G1,G2` no existen algoritmos de tiempo polinomial para determinar si son isomorfos, el algoritmo más simple conceptualmente consiste en explorar todas las posibles permutaciones de índices del grafo `G2` y ver si coincide con `G1`. Este algoritmo es al menos $O(N_v \cdot N_v!)$. Pero podemos encontrar tests que pueden descartar en forma relativamente rápida si dos grafos NO son isomorfos. Esto se basa en determinar propiedades que son **invariantes** ante una renumeración de los índices, por ejemplo la cantidad de vértices, y la cantidad de aristas. Sin embargo con esto no basta porque por ejemplo los dos grafos `G5,G6` tienen el mismo número de vértices y aristas pero no son isomorfos.

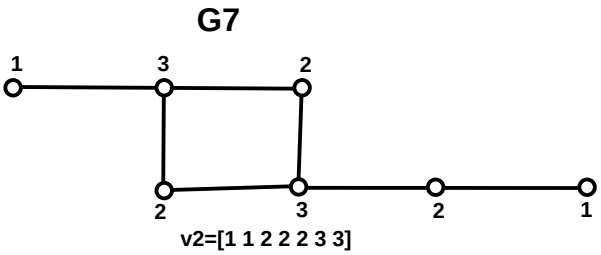


Esto es fácil de ver ya que hay otro invariante que es la **valencia** de sus vértices. La valencia de un vértice es la cantidad de aristas que salen del vértice, es decir el tamaño de su adyacencia. En `G5` hay un vértice con valencia 3 mientras que en `G6` hay uno con valencia 0. Entonces un predicado que puede determinar cuando los grafos NO son isomorfos podría ser determinar el vector de valencias ordenadas de menor a mayor y compararlos. En el ejemplo anterior tenemos

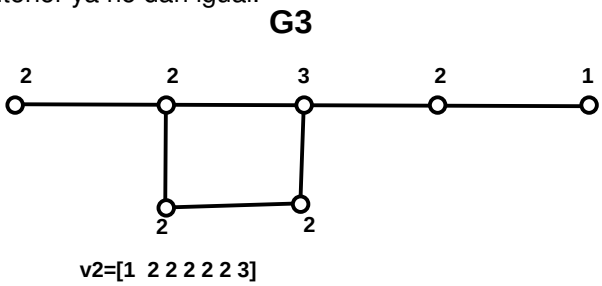
```
G5 -> [1,1,1,3]
G6 -> [0,2,2,2]
```

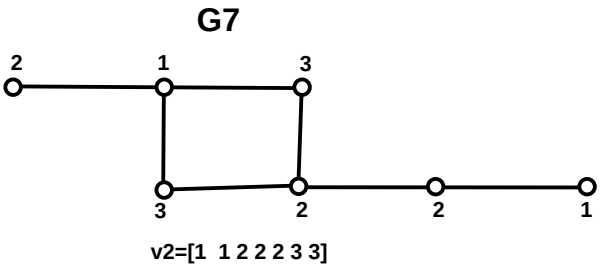
De todas formas, puede haber casos donde este criterio falle. Por ejemplo para los grafos `G4,G7` de la figura, las valencias ordenadas son `[1,1,2,2,2,3,3]`. (El número en el nodo es su valencia).





Una definición alternativa de valencia es que es el número de vértices a distancia 1. Podemos tomar como otro invariante el número de vecinos $v2$ a distancia 2. Podemos ver entonces que en el caso anterior ya no dan igual.





Por lo tanto sugerimos el siguiente algoritmo para determinar (en forma aproximada) si dos grafos son isomorfos

- Chequear si sus números de vértices son iguales.
- Calcular para cada vértice el número de vecinos a distancia 1 (valencia **v1**), ordenarlos y compararlos.
- Calcular para cada vértice el número de vecinos a distancia 2, ordenarlos y compararlos (**v2**).

Consigna: Es escribir una función
`bool isomorph(graph_t &G1,graph &G2);` que determina (en forma aproximada) si dos grafos son isomorfos.

Nota 1: El algoritmo descripto es **aproximado** en el sentido de que sólo puede dar falsos positivos, es decir, si **`isomorph(G1,G2)`** da **`false`** entonces los grafos seguramente NO son isomorfos. Pero si da positivo puede ser todavía que no sean ismorfos.

Nota 2: Se acepta cualquier otra propuesta (**`isomorph2()`**), mientras sea mejor o igual que la anterior. Es decir de un conjunto de falsos positivos menor o igual que el anterior. Es decir,

- Si son isomorfos debe dar **`true`**. (No debe haber falsos negativos).
- Si no son isomorfos y **`isomorph(G1,G2)`** da **`false`**, entonces **`isomorph2(G1,G2)`** debe dar **`false`**.

[Ej. 2] **[flux]** Escribir utilizando la librería Eigen una función
`double flux(shape_t &shape,vector_field_t &vf);` que calcula (2D) el flujo del campo vectorial $\mathbf{V}$ (dado por la clase **`vector_field_t`**) a través de la curva Γ (dada por la clase **`shape_t`**).

$$\Phi = \int_{\Gamma} \mathbf{V}(x) \cdot \hat{\mathbf{n}} \, \mathrm{d}\Gamma$$

```
class vector_field_t {
  public:
  virtual void field(MatrixXd &x,MatrixXd &V)=0;
};

class shape_t {
  public:
  virtual int size()=0;
  virtual void xnode(int j,MatrixXd &x)=0;
};
```

La forma geométrica consiste en una serie de **N=shape.size()** segmentos (elementos lineales) en 2D. El segmento **j** está definido por las coordenadas de los puntos **j** y **j+1**, por lo tanto hay **N+1** vértices. Para encontrar el vértice **j** debe hacerse **shape.xnode(j,x)**.

Dado un punto en el espacio **x** el campo vectorial en ese punto **V** se obtiene haciendo **vf.field(x,V)**.

Todos los vectores (posiciones y campo vectorial) son **MatrixXd** de tamaño **(2,1)**.

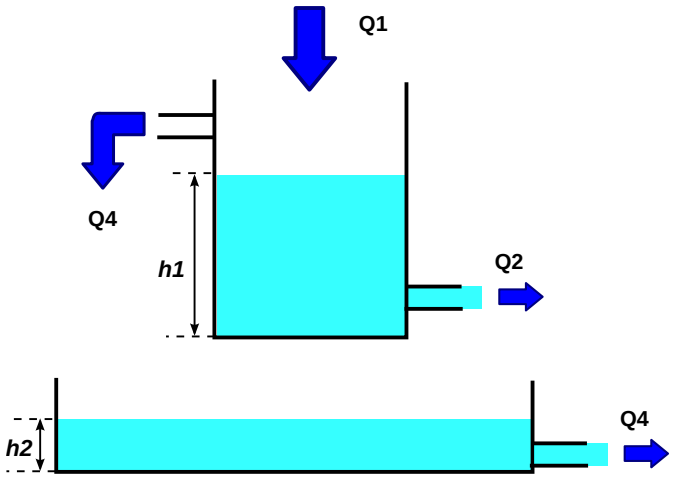
Ayuda:

- Recorrer los segmentos, calcular su punto medio **xc** y calcular el campo vectorial **V** llamando a **vf.field(xc,V)**.
- Calcular el área (con normal) $\hat{\mathbf{n}}\,\mathrm{d}Gamma$ del segmento $[x_j, x_{j+1}]$ haciendo $\Delta x = (x_{j+1} - x_j)$ y rotándolo 90 grados en sentido horario.
- Acumular en el flujo el producto escalar del campo con el área con normal.

La función de chequeo **check_flux(flux)** hace una serie de chequeos, con segmentos y círculos, la salida de los cuales debe dar:

```
flux (segment[1,0], uniform(0,1): -1.000000
flux (segment[1,0], uniform(1,0): 0.000000
flux (segment[0,1], uniform(1,0): 1.000000
flux (segment[0,1], uniform(0,1): 0.000000
flux (circle(c=(0,0),R=1), radial(c=(0,0),int=1): 6.279052
flux circle(c=(0,0),R=1), radial(c=(0,0),intens=1), N=1000: 6.283144
flux circle(c=(0,0),R=1), radial(c=(0,0),intens=100): 62.831440
flux circle(c=(0,0),R=1), sole(c=(0,0),intens=100): 0.000000
```

[Ej. 3] **[tanks]** Resolver con la librería ODEINT el sistema de dos ecuaciones que regula la altura de líquido en dos tanques conectados como indica la figura.



El tanque superior es alimentado con un caudal $Q_1(t)$ que descarga un caudal variable en el tiempo, de tipo onda cuadrada de período **T**

```
tau = t-floor(t/T)*T;
Q1 = Q10*(tau<T/2.0),
```

es decir durante **T/2** descarga un caudal constante **Q10** y durante la otra mitad del período no descarga. El tanque descarga a la red con un caudal **Q2** que es proporcional a la altura del líquido **h1**

$$Q_2 = C_2 \rho h_1 g$$

Para que no rebalse, cuando el líquido llega a una altura **H** existe una segunda descarga a un tanque inferior.

$$Q_3 = \begin{cases} 0; & \text{si } h_1 \leq H; \\ C_3(h - H); & \text{si } h_1 > H; \end{cases}$$

El tanque inferior descarga con un caudal proporcional a la altura

$$Q_4 = C_4 \rho h_2 g \tag{1}$$

Para cada tanque la ecuación de evolución de su altura es

$$A\dot{h} = \sum Q_j$$

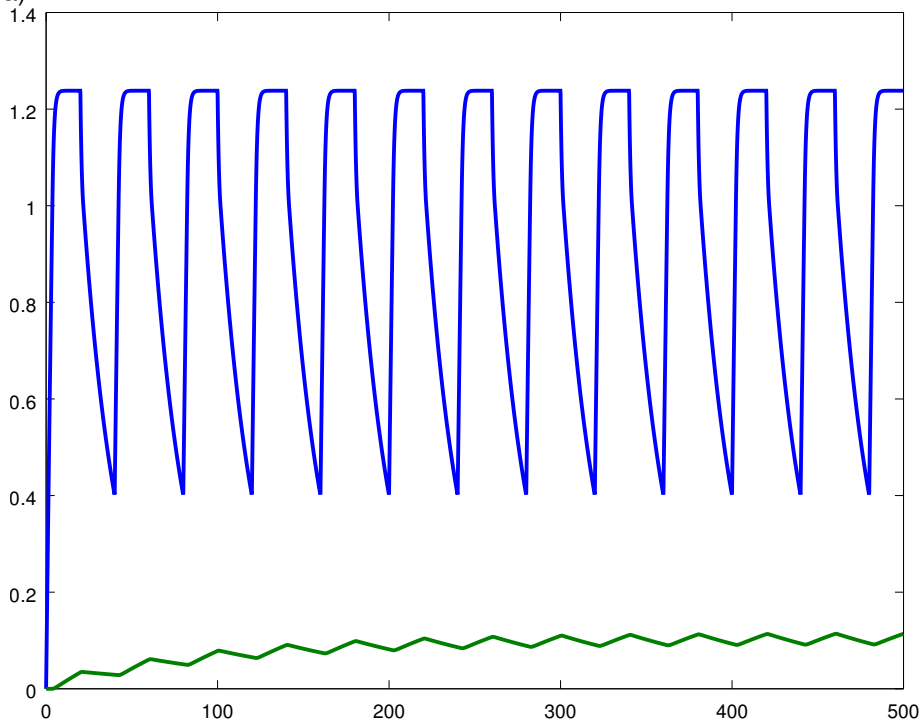
donde **A** es el área del tanque y $\sum Q_j$ la sumatoria (con signo!) de los caudales que ingresan/egresan al tanque.

Consigna: Hallar la altura promedio del tanque inferior luego del transitorio inicial, para los siguientes parámetros:

```
A1 = 1;
C2 = 0.5e-5;
rho = 1000;
g = 10;
C3 = 1;
```

```
H = 1;
Q10 = 0.3;
T = 40;
A2 = 100;
C4 = 1e-4;
```

Ayuda: El ejercicio consiste en escribir la función o clase correspondiente al sistema. El estado del sistema son las dos alturas h_1, h_2. Resolver para un tiempo prudencial y encontrar el promedio de h_2 (en forma aproximada).



Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado **`program.cpp`**. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si **`f`** es la función a evaluar tenemos

  ```
  ev.eval1(f,vrbs);
  ```

  `ev.eval1(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario **`f`** y compara la salida del usuario (**`user`**) con respecto a la esperada (**`ref`**). Si la verbosidad (el argumento

vrbs) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

Curso de Programación en C++.
TPLR-2015. Recuperatorio de Trabajo Práctico de Laboratorio. [2015-09-17]

Ejercicios

[Ej. 1] **[contains]** Escribir una función
`bool contains(list<int> &L1,list<int> &L2);`
que determina si la lista `L1` contiene a la lista `L2`. Es decir cada elemento de `L1` debe estar en `L2`, teniendo en cuenta la cantidad de veces que ocurre. Por ejemplo si `L1=(3,2,1,2,3)`, entonces

- Para `L2=(3,3) -> true`
- Para `L2=(1,1) -> false`

Ayuda:

- Escribir una función
  `void list2map(list<int> &L1,map<int,int> &M);`
  que deja en `M[x]` la cantidad de veces que aparece `x` en `L`. Por ejemplo:
  - `L1=(3,2,1,2,3) -> M={1->1,2->2,3->2}`
  - `L2=(3,3) -> M={3->2}`
  - `L2=(1,1) -> M={1->2}`
- Contruir los `M1` y `M2` correspondientes a `L1` y `L2`.
- Para cada elemento en `L2` chequear que la el conteo de veces que aparece en `L2` sea menor o igual que las veces que aparece en `M1`.

[Ej. 2] **[getjaco]** Consideremos funciones vectoriales de tipo

$$f : \mathbb{R}^m \to \mathbb{R}^n \tag{1}$$

donde m, n son enteros positivos. Queremos escribir una función `GetJaco()` que calcule por diferencias el gradiente de estas funciones, es decir

$$J : \mathbb{R}^m \to \mathbb{R}^{n \times m} \tag{2}$$

donde

$$J(j,k) = \frac{\partial f_j}{\partial x_k}. \tag{3}$$

La aproximación por diferencias es

$$J(:,k) = \frac{f(x + \epsilon \hat{\mathbf{e}}_k)) - f(x - \epsilon \hat{\mathbf{e}}_k))}{2\epsilon} \tag{4}$$

donde $\hat{\mathbf{e}}_k)$ es un vector con todos 0's menos un 1 en la posición k.

Las funciones f las representamos con una clase polimórfica

```
class vecfun_t {
  public:
  virtual VectorXd eval(VectorXd &x)=0;
};
```

donde **eval()** representa la evaluación de la función.

Consigna 1: Escribir una función
void GetJaco(vecfun_t &F,VectorXd &x,MatrixXd &J,double eps=1e-5); que cumple la tarea indicada, devolviendo por **J** el Jacobiano de **F** en **x**.

Consigna 2: Escribir funciones de prueba para **GetJaco()**

- **Id(x)=x** debe retornar **J=I** (matriz identidad)
- **phi(x)=||x||^2** debe retornar **J=**$\nabla(\phi) = 2x$
- Dado un vector Ω definir la función vectorial de $\mathbb{R}^3$ en $\mathbb{R}^3$: $x \to \Omega \times x$ usando la función **cross()** de Eigen. Calcular el Jacobiano con **GetJaco()** que debe retornar

$$\begin{bmatrix} 0 & -\Omega_z & \Omega_y \\ \Omega_z & 0 & -\Omega_x \\ -\Omega_y & \Omega_x & 0 \end{bmatrix} \tag{5}$$

mstorti@galileo/texstuff-1.4.0-67-ge52a7899/Tue Jun 22 09:06:33 2021 -0300

Curso de Programación en C++.
TPL0-2017. Trabajo Práctico de Laboratorio. [2017-09-11]

PASSWD PARA EL ZIP: YK9L KL5G SKBT

Ejercicios

[Ej. 1] **[separate]** Escribir una función
`void separate(vector<int> &v,vector<int> &even,vector<int> &odd);`
que toma un vector de enteros `v` y separa los pares y los impares, preservando el orden en el que estaban. Por ejemplo:

```
v=[0,1,2,3,4,5,6,7,8], => even=[0,2,4,6,8], odd=[1,3,5,7]
v=[5,9,3,9,16,14,11,11,2,4,1] => even=[16,14,2,4], odd=[5,9,3,9,11,11,1]
v=[0,7,17,5,15,5,16] => even=[0,16], odd=[7,17,5,15,5]
```

[Ej. 2] **[is-greater-than]** Escribir una función
`bool is_greater_than(vector<int> &v1,vector<int> &v2);` Que retorna `true` si la secuencia `v1` es **mayor** que `v2`, en el siguiente sentido:

- Si `v1` y `v2` tienen igual longitud entonces `v1>v2` si para todo `j`: `v1[j]>=v2[j]` y al menos para algún `j` `v1[j]>v2[j]`. Por ejemplo:

  ```
  v1=[1,2,3], v2=[0,1,2] -> true
  v1=[0,1,3], v2=[0,1,2] -> true
  v1=[0,1,3], v2=[0,1,3] -> false
  v1=[3,2,1], v2=[0,1,3] -> false
  ```

- Si los vectores no tienen la misma longitud entonces se considera que se completan con $-\infty$, por lo tanto

  ```
  v1=[0,1,2,0], v2=[0,1,2] -> true
  v1=[0,1,2], v2=[0,1,2,0] -> false
  ```

  Nota: Los elementos de `v1,v2` son todos `>=0` de manera que se puede usar `-1` en vez de $-\infty$.

Ayuda: Se sugiere el siguiente algoritmo.

- Escribir una función `int elem(vector<int> &v,int j);` que retorna el elemento correspondiente `v[j]` o bien `-1` si la posición no existe.
- Sea `n=max(n1,n2)` la máxima longitud de ambos vectores. Recorrer todas las posiciones en `0<=j<n` y chequear si `elem(v1,j)<elem(v2,j)` para algún `j`, en dicho caso retornar `false`.
- Recorrer nuevamente todos los `j`, si `elem(v1,j)>elem(v2,j)` para algún `j` retornar `true` caso contrario retornar `false`.

[Ej. 3] **[max-sum]** Escribir una función
`int max_sum(vector<vector<int» &vv);` Que toma un vector de vector de enteros `vv` y retorna el índice `j` en `vv` tal que la suma de los elementos de `vv[j]` es mayor o igual que la de todos los otros. Si hay varios que tienen la misma suma máxima debe retornar el primero. Por ejemplo:

```
vv=[[0,1,2],[4],[3,2]] => 2
vv=[[0,1,2],[5],[3,2]] => 1
```

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado `program.cpp`. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si `f` es la función a evaluar tenemos

  ```
  ev.eval1(f,vrbs);
  ```

  `ev.eval1(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario `f` y compara la salida del usuario (`user`) con respecto a la esperada (`ref`). Si la verbosidad (el argumento `vrbs`) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

mstorti@galileo/texstuff-1.4.0-69-gb3543f11/Tue Jun 22 10:41:21 2021 -0300

Curso de Programación en C++.
TPL0. Trabajo Práctico de Laboratorio. [2017-09-11]

PASSWD PARA EL ZIP: YK9L KL5G SKBT

Ejercicios

[Ej. 1] **[separate]** Escribir una función
`void separate(vector<int> &v,vector<int> &even,vector<int> &odd);`
que toma un vector de enteros `v` y separa los pares y los impares, preservando el orden en el que estaban. Por ejemplo:

```
v=[0,1,2,3,4,5,6,7,8], => even=[0,2,4,6,8], odd=[1,3,5,7]
v=[5,9,3,9,16,14,11,11,2,4,1] => even=[16,14,2,4], odd=[5,9,3,9,11,11,1]
v=[0,7,17,5,15,5,16] => even=[0,16], odd=[7,17,5,15,5]
```

[Ej. 2] **[is-greater-than]** Escribir una función
`bool is_greater_than(vector<int> &v1,vector<int> &v2);` Que retorna `true` si la secuencia `v1` es **mayor** que `v2`, en el siguiente sentido:

- Si `v1` y `v2` tienen igual longitud entonces `v1>v2` si para todo `j`: `v1[j]>=v2[j]` y al menos para algún `j` `v1[j]>v2[j]`. Por ejemplo:

  ```
  v1=[1,2,3], v2=[0,1,2] -> true
  v1=[0,1,3], v2=[0,1,2] -> true
  v1=[0,1,3], v2=[0,1,3] -> false
  v1=[3,2,1], v2=[0,1,3] -> false
  ```

- Si los vectores no tienen la misma longitud entonces se considera que se completan con $-\infty$, por lo tanto

  ```
  v1=[0,1,2,0], v2=[0,1,2] -> true
  v1=[0,1,2], v2=[0,1,2,0] -> false
  ```

  Nota: Los elementos de `v1,v2` son todos `>=0` de manera que se puede usar `-1` en vez de $-\infty$.

Ayuda: Se sugiere el siguiente algoritmo.

- Escribir una función `int elem(vector<int> &v,int j);` que retorna el elemento correspondiente `v[j]` o bien `-1` si la posición no existe.
- Sea `n=max(n1,n2)` la máxima longitud de ambos vectores. Recorrer todas las posiciones en `0<=j<n` y chequear si `elem(v1,j)<elem(v2,j)` para algún `j`, en dicho caso retornar `false`.
- Recorrer nuevamente todos los `j`, si `elem(v1,j)>elem(v2,j)` para algún `j` retornar `true` caso contrario retornar `false`.

[Ej. 3] **[max-sum]** Escribir una función
`int max_sum(vector<vector<int>> &vv);` Que toma un vector de vector de enteros `vv` y retorna el índice `j` en `vv` tal que la suma de los elementos de `vv[j]` es mayor o igual que la de todos los otros. Si hay varios que tienen la misma suma máxima debe retornar el primero. Por ejemplo:

```
vv=[[0,1,2],[4],[3,2]] => 2
vv=[[0,1,2],[5],[3,2]] => 1
```

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado `program.cpp`. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si `f` es la función a evaluar tenemos

  ```
  ev.eval1(f,vrbs);
  ```

  `ev.eval1(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario `f` y compara la salida del usuario (`user`) con respecto a la esperada (`ref`). Si la verbosidad (el argumento `vrbs`) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

mstorti@galileo/texstuff-1.3.2-122-ga5e21a43/Sun Sep 10 16:29:08 2017 -0300

Curso de Programación en C++.
TPL1-2017. Trabajo Práctico de Laboratorio. [2017-10-02]

PASSWD PARA EL ZIP: FFP1 K25Q 5D4B

Ejercicios

[Ej. 1] **[occurm]**. Dada un vector `v`, y un entero `m`, escribir una función `void occurm(vector<int> &v,int m,vector<int> &vm);` que genere una lista `vm` con todos los elementos que aparecen exactamente `m` veces en `v`. Por ejemplo, si `v=[0,1,2,3,4,5,1,2,4,2]` entonces `occurm(v,2,vm)` debe dejar `vm=[1,4)]` ya que son los únicos dos elementos que están exactamente 2 veces. `occurm(v,1,vm)` debe retornar `vm=[0,3,5]`.

Nota: Los elementos en `vm` deben aparecer en el orden que estaban originariamente (la primera vez) en `v` (concepto de estabilidad).

Ayuda1: Recorrer los elementos de `v`. Para cada elemento de `v` chequear si está en `vm`. Si NO está, contar cuantas veces está en `v` y si el número de veces es el indicado (es decir coincide con `m`) entonces agregarlo al final de `vm`.

Ayuda2: Puede ser conveniente escribir una función `int count(vector<int> &v,int x)` que retorna cuántas veces `x` está contenido en `v`.

[Ej. 2] **[subvmax]** Programar una función `void subvmax(vector<int> &v,vector<int> &vmax);` la cual recibe una lista de enteros `v` y encuentra y retorna el subvector consecutivo `vmax` que obtenga la mayor suma entre todos sus elementos. Notar que debido a que algunos elementos pueden ser negativos el problema no se resuelve simplemente tomando todos los elementos. También es posible que el subvector resultado no contenga ningún elemento, en el caso de que todos los elementos de `v` sean negativos.

Por ejemplo:

```
[1, 2, -5, 4, -3, 2] -> [4]
[5, -3, -5, 1, 7, -2] -> [1,7]
[4, -3, 11, -2] -> [4, -3, 11]
[4  -4  2 2] -> [4]
```

Si hay varios subvectores que den la misma suma máxima debe retornar el primero y el más corto. En el último ejemplo de arriba hay tres subvectores que dan la máxima suma de 4: a saber `[4]`, `[4 -4 2 2]`, `[2 2]`. Por este criterio debe retornar `[4]` ya que es el que empieza primero y el más corto.

Ayuda:

- Escribir una función `int sumsubv(vector<int> &v,int j1,int j2);` que calcula la suma de los elementos del vector `v` en el rango `[j1,j2)`.
- Recorrer todos los pares `[j1,j2)` con `j1<=j2` y calcular el par que tiene la suma máxima.
- Una vez determinado el par `[J1,J2)` que tiene la suma máxima copiar el rango `[J1,J2)` a `vmax`.

[Ej. 3] **[subvector]** Escribir una función predicado `bool subvector(vector<int> &v1, vector<int> &v2);` que determina si `v2` es un subvector contiguo de `v1` es decir si existe un rango `[j1,j2)` tal que `v2=v1[j1,j2)`. Por ejemplo

```
v1=[1 3 2 5 6] v2=[3 2 5] -> true
v1=[1 3 2 5 6] v2=[3 5 6] -> false // No es contiguo
```

Ayuda: Para cada posicicón `j` en `v1` recorrer `v2` y chequear si los elementos de `v2` están en `v1[j,end)`. Si el tamaño de `v1[j,end)` es menor que el de `v2` retornar `false`, si está contenido retornar `true`.

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado `program.cpp`. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si `f` es la función a evaluar tenemos

  ```
  ev.eval1(f,vrbs);
  ```

  `ev.eval1(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario `f` y compara la salida del usuario (`user`) con respecto a la esperada (`ref`). Si la verbosidad (el argumento `vrbs`) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

Curso de Programación en C++.
TPL2-2017. Trabajo Práctico de Laboratorio. [2017-10-23]

PASSWD PARA EL ZIP: 2VRL XZDN 57Z6

Ejercicios

[Ej. 1] **[separate]** Dada una **función de mapeo** `int f(int)` y un conjunto de enteros `S` queremos construir la **función inversa** consideremos el conjunto de los valores `yj=f(xj)` para todos los `xj` en `S`. Queremos construir una estructura `y2x` que nos permita obtener para cada valor de `y` los valores `x` tales que `f(x)=y`. Como `f` no es necesariamente **biyectiva** puede haber varios `x` que cumplen la condición, por eso usaremos para esta estructura un `map<int,set<int>` es decir, para cada valor de la imagen `y` el conjunto `y2x[y]` es el conjunto de los valores de `x` tales que `y=f(x)`.

Consigna: escribir una función
`void separate(set<int> &S,mapfun_t f,map<int,set<int> > &y2x);` que realiza la tarea indicada.

Ejemplo:

- `S : [-4,-3,0,1,2,3,4,5,6,7], f=sign(x),`
  `y2x={-1=>[-4,-3], 0=>[0], 1=>[1,2,3,4,5,6,7]}`
- `S: [-4,-3,0,1,2,3,4,5,6,7], f=abs(x),`
  `y2x={0=>[0], 1=>[1], 2=>[2], 3=>[-3,3], 4=>[-4,4], 5=>[5], 6=>[6], 7=>[7]}`
- `S: [-4,-3,0,1,2,3,4,5,6,7], f=mod(x,3),`
  `y2x={0=>[-3,0,3,6], 1=>[1,4,7], 2=>[-4,2,5]}`

[Ej. 2] **[min-path]** (**Problema del Agente Viajero**) Dado un grafo `graph_t G` escribir una función `int min_path(graph_t &G,vector<int> &pathmin);` en el grafo `G` que determina el camino de longitud mínima `pathmin`. La clase `graph_t` está definida por la siguiente interfaz virtual pura,

```
class graph_t {
  public:
  virtual int size()=0; // cantidad de vertices
  virtual int dist(int i,int j)=0; // distancia entre los vertices i,j
};
```

El camino está representado por un `vector<int> path`. Los caminos son cíclicos es decir vuelven al vértice de partida. Es decir que si `path=[0,1,2,3]` entonces hay que sumar las aristas `0-1,1-2,2-3,3-0`

El algoritmo consiste en recorrer todos los caminos posibles (hay `N!` posibilidades) se provee la función `next_path(p)` la cual dado un camino `p` lo actualiza al siguiente camino. La función retorna `false` cuando llega al último es decir cuando no lo puede actualizar más. Para recorrer todos los caminos hay que entonces inicializar el camino con `[0 1 2 ... N-1]` y luego aplicar sucesivamente `next_path()` hasta que retorne `false`.

```
vector<int> path(N);
for (int j=0; j<N; j++) path[j] = j;
```

```
while (1) {
  // Analizar 'path'
  if (!next_path(path)) break;
}
```

Consigna 1: Escribir la función `int min_path(graph_t &G,vector<int> &pathmin);`

Ejemplos: Si el grafo es un círculo `circle_t` donde los `dist(j,k)=1` si `j==(k+1)%N` o viceversa, caso contrario `dist(j,k)=Inf` (usaremos `Inf=10000`). El camino mínimo tiene longitud `N` y debe ser `[0,1,...,N-1]` o cualquier rotación del mismo o inversión, e.g. `[2,...,N-1,0,1]` o `[N-1,N-2,...,1,0]`.

Consigna 2: Escribir los siguientes grafos (clases derivadas):

- `full_t`: Todos los vértices están conectados por la misma distancia `dist(i,j)=1`. El camino mínimo puede ser cualquiera y es de longitud `N`. Calcular el camino mínimo para grafos `full_t` de 4 y 7 vértices.
- `grid_t`: Consiste en una grilla 2D de `Nx x Ny` vértices. El vértice en la posición 2D `(j,k)` le corresponde el índice `n=k*Nx+j`. La distancia entre los nodos `n1=(j1,k1)` y `n2=(j2,k2)` es la norma L_1 de la distancia: `dist(n1,n2)=|j2-j1|+|k2-k1|`. Calcular el camino mínimo para grafos `grid_t` 3x3 y 4x2 vértices.

[Ej. 3] **[mkorth]** (Eigen) Dada una matriz $\mathbf{Q}$ buscar una matriz *cercana* $\mathbf{O}$ tal que $\mathbf{O}$ es ortogonal, es decir $\mathbf{O}\mathbf{O}^T = \mathbf{I}$. Para ello asumir que la corrección es $\Delta\mathbf{Q}$ de manera que queremos que

$$(\mathbf{Q}+\Delta\mathbf{Q})(\mathbf{Q}+\Delta\mathbf{Q})^T = \mathbf{Q}\mathbf{Q}^T + \mathbf{Q}\Delta\mathbf{Q}^T + \Delta\mathbf{Q}\mathbf{Q}^T + \Delta\mathbf{Q}\Delta\mathbf{Q}^T = \mathbf{I} \tag{1}$$

Despreciando el último término y asumiendo $\mathbf{Q}\Delta\mathbf{Q}^T = \Delta\mathbf{Q}\mathbf{Q}^T$ queda

$$\begin{aligned} \Delta\mathbf{Q} &= \tfrac{1}{2}(\mathbf{I} - \mathbf{Q}\mathbf{Q}^T)(\mathbf{Q}^T)^{-1}, \\ \mathbf{Q} &\leftarrow \mathbf{Q} + \Delta\mathbf{Q}; \end{aligned} \tag{2}$$

Consigna: Escribir una función `void mkorth(MatrixXd &Q);` que realiza la tarea explicada aplicando iterativamente (**??**) hasta que $||\Delta\mathbf{Q}|| <$`tol=1e-7`

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado `program.cpp`. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si `f` es la función a evaluar tenemos

```
ev.eval1(f,vrbs);
```

`ev.eval1(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario **`f`** y compara la salida del usuario (**`user`**) con respecto a la esperada (**`ref`**). Si la verbosidad (el argumento **`vrbs`**) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

mstorti@galileo/texstuff-1.4.0-69-gb3543f11/Tue Jun 22 10:42:55 2021 -0300

Curso de Programación en C++.
TPL3-2017. Trabajo Práctico de Laboratorio. [2017-11-27]

PASSWD PARA EL ZIP: DW2N 11XD LS3B

Ejercicios

[Ej. 1] **[splitmono]** Dado un `vector<int> v` escribir una función
`void splitmono(vector<int> &v,vector< vector<int> > &vmono);`
que separa a `v` en subsecuencias monótonas crecientes. Ejemplos:

```
v:[1,2,3,1,2,1]}, vmono:[[1,2,3],[1,2],[1]]
v:[0,4,3,4,1,4,1,1,2]}, vmono:[[0,4],[3,4],[1,4],[1,1,2]]
v:[1,4,1,3,3,0,3,3,4,0,0,0]}, vmono:[[1,4],[1,3,3],[0,3,3,4],[0,0,0]]
v:[2,3,3,2,4,1,0,4,0,4]}, vmono:[[2,3,3],[2,4],[1],[0,4],[0,4]]
```

[Ej. 2] **[filtersets]** Escribir una función
`void filtersets(vector< set<int> > &vset,vector< set<int> > &vsout,pred_t pred);`
que dado un vector de conjuntos y un predicado **`pred`**, deja en **`vsout`** sólo los conjuntos que satisfacen el predicado **`pred`**.

```
pred:sumzero  vset:[[1,-1],[1,2],[2,3]]           => vsout:[[-1,1]]
pred:hassize2 vset:[[1,-1,5],[1,2],[2,3]]         => vsout:[[1,2],[2,3]]
pred:hassize2 vset:[[-1,1,2],[-1],[-2,3],[-2,-1,0]]=> vsout:[[-2,3]]
pred:hassize2 vset:[[2,3],[0,3],[-2,-1,1],[-2,1]] => vsout:[[2,3],[0,3],[-2,1]]
pred:sumzero  vset:[[-2,-1,3],[2],[1,3]]          => vsout:[[-2,-1,3]]
```

Adicionalmente: escribir los predicados:

- **`sumzero`**: La suma de los elementos da 0.
- **`hassize2`**: El tamaño del conjunto es 2.
- **`hasodd`**: Tiene al menos un elemento impar
- **`allpositive`**: Todos los elementos son positivos.

[Ej. 3] **[inside]** Escribir una función
`bool inside(MatrixXd &xvrtx, VectorXd &xprobe);` que determina si el punto **`xprobe`** está dentro del **símplice** definido por sus vértices **`xvrtx`**. (Nota: Los símplices son los polígonos con menor número de vértices (**`ndim+1`**) en dimensión **`ndim`**, es decir en 1D son segementos, en 2d son triángulos y en 3D son tetrahedros). Los vefrtices del símplice están en las columnas de **`xvrtx`** que por lo tanto es de tamaño **`ndim x (ndim+1)`**. **`xprobe`** es de dimensión **`ndim`**.

La función debe funcionar en dimensión 2 o 3. Se puede determinar la dimensión del espacio a partir de la longitud de **`xprobe`**.

Ayuda: basta con determinar si existen **ndim+1** escalares $0 \leq \alpha_j \leq 1$ tales que

$$\mathbf{x}_p = \sum_j \alpha_j \mathbf{x}_j \tag{1}$$

Con la restricción que $\sum_{j=1}^{n_d+1} \alpha_j = 1$. O sea que debemos resolver un sistema de $(n_d+1) \times (n_d+1)$ de la siguiente manera:

$$\begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \mathbf{x}_4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_p \\ 1 \end{bmatrix} \tag{2}$$

$$\mathbf{C}\boldsymbol{\alpha} = \mathbf{b}$$

Una vez resuelto el sistema, debe verificarse que todos los α_j estén en el rango $[0, 1]$.

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado **`program.cpp`**. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si **`f`** es la función a evaluar tenemos

```
ev.eval1(f,vrbs);
```

`ev.eval1(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario **`f`** y compara la salida del usuario (**`user`**) con respecto a la esperada (**`ref`**). Si la verbosidad (el argumento **`vrbs`**) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

mstorti@galileo/texstuff-1.4.0-69-gb3543f11/Tue Jun 22 10:43:31 2021 -0300

Curso de Programación en C++.
TPL4-2017. Trabajo Práctico de Laboratorio. [2018-05-09]

PASSWD PARA EL ZIP: WQQR RPF3 V8HM

Ejercicios

[Ej. 1] **[mapvec]** Dado un vector `v`, y una correspondencia `M`, devolver el vectotr `w` que contiene el resultado de aplicar la correspondencia `M` a esos elementos. Es decir si `x` es un elemento de `v`, entonces `w` debe contener el elemento `y=M[x]` (en el mismo orden que están en `v`). Adicionalmente, de todos los elementos de `w` dejar solo los elementos que satisfacen el predicado `pred`, es decir aquellos `y` tales que `pred(y)` retorna verdadero.

Consigna: Escribir la función `void mapvec(map<int,int> &M,bool (*pred)(int),vector<int> &v, vector<int> &w);` que realiza la tarea indicada.

Ejemplos:

```
M: {0=>1,1=>2,2=>3}, pred: even, v:[0,1,2]     ->     output:[2]
M: {0=>1,1=>2,2=>3}, pred: odd,  v:[0,1,2]     ->     output:[1,3]
M: {1=>0,2=>1,3=>2}, pred:odd,   v:[1,2,3]     ->     output:[1]
M: {1=>0,2=>1,3=>2}, pred:even,  v:[1,2,3]     ->     output:[0,2]
```

Ayuda: Recorrer los elementos de `v`, aplicar `M` y verificar si la imagen satisface el predicado o no. Si lo satisface insertar ese elemento imagen en `w`.

[Ej. 2] **[reciprocal]** Dadas dos correspondencias `M1` y `M2` determinar si una es la inversa de la otra, es decir si para cada par de asignación `y=M1[x]` debe haber un par `x=M2[y]` y viceversa (para todo par en `M2` existe el inverso en `M1`).

Consigna: Escribir la función `bool reciprocal(map<int,int> &M1,map<int,int> &M2);` que realiza la tarea indicada.

Ejemplos:

```
M1:{0=>1,1=>2,2=>3},       M2:{1=>0,2=>1,3=>2},       -> true
M1:{0=>1,1=>2,2=>3},       M2:{1=>0,2=>4,3=>2},       -> false
M1:{0=>1,1=>2,2=>2},       M2:{1=>0,2=>1},            -> false
M1:{0=>1,1=>0,2=>3,3=>2},  M2:{0=>1,1=>0,2=>3,3=>2},  -> true
```

[Ej. 3] **[only1]** Dadas dos conjuntos `A` y `B` la differencia simétrica `C` es el conjunto de los elementos que están en `A` o `B` pero no en ambos a la vez. El nombre de diferencia simétrica, proviene por contraposición con la diferencia `A-B` que son los elementos que están en `A` pero no en `B`.

Otra forma de verlo es que si, dado un elemento `x` contamos en cuantos conjuntos está, la diferencia simétrica son aquellos que están en sólo 1 conjunto. Esta definición se puede extender a cualquier número de conjuntos.

Consigna: Escribir una función `void only1(vector< set<int> >&VS,set<int> &DS);` que realiza la tarea indicada.

Ejemplos:

```
VS=[{0,1,2},{1,2,3}]           -> DS={0,3}
VS=[{0,1,2},{1,2,3},{2,4,5}], -> DS=[0,3,4,5]
```

Ayuda: Recorrer todos los elementos `x` de todos los conjuntos y mantener una correspondencia `map<int,int> M` que almacena para cada elemento `x` la cantidad de conjuntos en los que está. Finalmente recorrer las asignaciones de `M` e insertar en `DS` sólo los elementos que tienen un conteo exactamente igual a 1.

[Ej. 4] **[trust-game]** Consideremos el siguiente juego. Hay dos jugadores de ambos lados de una máquina. En cada movida cada uno de los jugadores puede poner 1 moneda en la máquina. Si el jugador

- Si `p1` pone una moneda el otro `p2` recibe 3 monedas,
- Si `p1` no pone la moneda no recibe nada.

Ninguno de los dos sabe cuál es la jugada del otro, pero tienen acceso a toda la historia de jugadas entre ellos.

Por lo tanto, si ambos "colaboran" los dos ganan ambos 2 monedas, pero si uno de ellos "hace trampa" (no pone la moneda) tiene una ganancia adicional ya que no puso la moneda, es decir gana 3 (mientras que el oponente pierde 1). Finalmente si ambos "hacen trampa" (no ponen la moneda) ninguno de los dos gana nada (ganancia 0 para ambos).

La cuestión es cuál es la mejor estrategia de juego, es decir que de la máxima ganancia, en un entorno en el cual los otros jugadores pueden tener diferentes estrategias.

```
enum move_t {NONE, COLLAB, CHEAT};
class player_t {
  public:
  virtual move_t move(vector<move_t> &me,vector<move_t> &you)=0;
  virtual string state() {return string("unknown"); }
};
```

Consigna1: Escribir instancias (clases derivadas) de la clase `player_t` con las siguientes características:

- `collabr_t`: Siempre colabora.
- `cheater_t`: Siempre hace trampa.
- `copycat_t`: (Copión) Copia lo que hizo el otro jugador en la jugada previa.
- `grudger_t`: (Resentido) Inicialmente colabora, pero apenas es traicionado una sola vez, no colabora nunca más.
- `copysmart_t`: (Copión astuto) Mira las dos jugadas previas del oponente. Si fue traicionado las dos veces traiciona, caso contrario colabora.
- `random_t`: (Aleatorio) Tira la moneda y colabora o traiciona con probabilidades 50 %.

Nota: El método **`state()`** retorna un string que describe al jugador, por ejemplo **`cheater_t`** puede retornar **`"siempre traiciona"`**.

Consigna2: Escribir una función
`void game(player_t &p1,player_t &p2,int nrounds,int &score1,int &score2);` que toma dos jugadores **`p1`** y **`p2`** y los hace jugar **`nrounds`** jugadas. Retorna también (score, cantidad de monedas ganadas) de cada jugador.

Referencia: Al hacer jugar los diferentes jugadores entre si debe dar lo siguiente (para 5000 jugadas cada par):

```
collaborator: score 10000 (2.000 per move), | collaborator  score: 10000 (2.000 per move)
collaborator: score -5000 (-1.000 per move), | cheater  score: 15000 (3.000 per move)
collaborator: score 10000 (2.000 per move), | copycat  score: 10000 (2.000 per move)
collaborator: score 10000 (2.000 per move), | copysmart  score: 10000 (2.000 per move)
collaborator: score 10000 (2.000 per move), | grudger  score: 10000 (2.000 per move)
collaborator: score 2356 (0.471 per move), | random  score: 12548 (2.510 per move)
===== Total score collaborator: 37356 (1.245/move)
cheater: score 15000 (3.000 per move), | collaborator  score: -5000 (-1.000 per move)
cheater: score 0 (0.000 per move), | cheater  score: 0 (0.000 per move)
cheater: score 3 (0.001 per move), | copycat  score: -1 (-0.000 per move)
cheater: score 6 (0.001 per move), | copysmart  score: -2 (-0.000 per move)
cheater: score 3 (0.001 per move), | grudger  score: -1 (-0.000 per move)
cheater: score 7371 (1.474 per move), | random  score: -2457 (-0.491 per move)
===== Total score cheater: 22383 (0.746/move)
copycat: score 10000 (2.000 per move), | collaborator  score: 10000 (2.000 per move)
copycat: score -1 (-0.000 per move), | cheater  score: 3 (0.001 per move)
copycat: score 10000 (2.000 per move), | copycat  score: 10000 (2.000 per move)
copycat: score 10000 (2.000 per move), | copysmart  score: 10000 (2.000 per move)
copycat: score 10000 (2.000 per move), | grudger  score: 10000 (2.000 per move)
copycat: score 4943 (0.989 per move), | random  score: 4947 (0.989 per move)
===== Total score copycat: 44942 (1.498/move)
copysmart: score 10000 (2.000 per move), | collaborator  score: 10000 (2.000 per move)
copysmart: score -2 (-0.000 per move), | cheater  score: 6 (0.001 per move)
copysmart: score 10000 (2.000 per move), | copycat  score: 10000 (2.000 per move)
copysmart: score 10000 (2.000 per move), | copysmart  score: 10000 (2.000 per move)
copysmart: score 10000 (2.000 per move), | grudger  score: 10000 (2.000 per move)
copysmart: score 3709 (0.742 per move), | random  score: 8857 (1.771 per move)
===== Total score copysmart: 43707 (1.457/move)
grudger: score 10000 (2.000 per move), | collaborator  score: 10000 (2.000 per move)
grudger: score -1 (-0.000 per move), | cheater  score: 3 (0.001 per move)
grudger: score 10000 (2.000 per move), | copycat  score: 10000 (2.000 per move)
grudger: score 10000 (2.000 per move), | copysmart  score: 10000 (2.000 per move)
grudger: score 10000 (2.000 per move), | grudger  score: 10000 (2.000 per move)
grudger: score 7412 (1.482 per move), | random  score: -2468 (-0.494 per move)
===== Total score grudger: 47411 (1.580/move)
random: score 12444 (2.489 per move), | collaborator  score: 2668 (0.534 per move)
random: score -2479 (-0.496 per move), | cheater  score: 7437 (1.487 per move)
random: score 5085 (1.017 per move), | copycat  score: 5081 (1.016 per move)
random: score 8753 (1.751 per move), | copysmart  score: 3693 (0.739 per move)
random: score -2453 (-0.491 per move), | grudger  score: 7375 (1.475 per move)
random: score 5109 (1.022 per move), | random  score: 5017 (1.003 per move)
===== Total score random: 26459 (0.882/move)
```

Nota de color:

- Otro nombre para este probema es el del **dilema del Prisionero** (goo.gl/iEXtMF, youtu.be/UkXI-zPcDIM).
- Es estudiado en **Teoría de Juegos**, la conclusión es que el juego tiene un **punto de equilibrio de Nash**: a saber **todos traicionan**, es decir que cada jugador si cambia su estrategia (esto es

colabora) pierde y por lo tanto retorna al punto de equilibrio (traicionar). Sin embargo la ganancia en este punto individual y colectiva es baja (nula).
Notablemente, la ganancia total e individual máxima se obtiene si todos colaboran, ya que en promedio ganan 2 unidades por jugada, bastante más que en el punto de equilibrio.

- En el sitio ncase.me/trust este concepto es explicado en detalle, con animaciones muy interesantes e implicancia en temas sociales como la confianza y la cooperación.

[Ej. 5] **[is-inside-cc]** Dados 3 puntos $\mathbf{x}_j \in \mathbb{R}^2$, la **circunferencia circunscripta** (CC) (en inglés *circumcircle*) es el círculo que pasa por los 3 puntos al mismo tiempo, por lo tanto el centro del círculo $\mathbf{x}_c$ es equidistante de los 3 puntos, es decir $\|\mathbf{x}_j - \mathbf{x}_c\| = R$. Dados los 3 puntos, podemos encontrar el centro de la CC planteando que dados dos de los puntos $\mathbf{x}_0, \mathbf{x}_1$ el centro $\mathbf{x}_c$ debe ser equidistante de ambos, por lo tanto debe estar en el plano bisector del segmento que los une, es decir

$$\begin{aligned} (\mathbf{x}_c - \mathbf{x}_0) \cdot (\mathbf{x}_1 - \mathbf{x}_0) &= (\mathbf{x}_1 - \mathbf{x}_c) \cdot (\mathbf{x}_1 - \mathbf{x}_0), \\ \implies \quad (\mathbf{x}_c - \mathbf{x}_{c,01}) \cdot \Delta\mathbf{x}_{01} &= 0, \end{aligned} \tag{1}$$

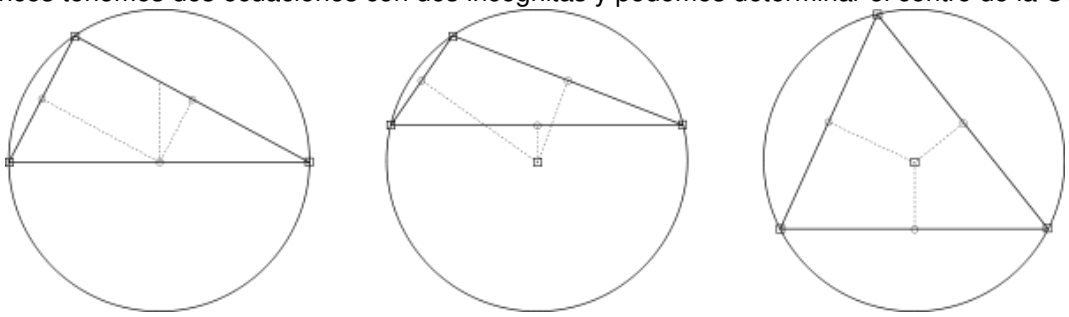
donde

$$\begin{aligned} \mathbf{x}_{c,01} &= (\mathbf{x}_0 + \mathbf{x}_1)/2, \\ \Delta\mathbf{x}_{01} &= (\mathbf{x}_1 - \mathbf{x}_0), \end{aligned} \tag{2}$$

son el punto medio del segmento $\mathbf{x}_0 - \mathbf{x}_1$ y el vector orientado a lo largo del mismo. Notar que esto representa una ecuación lineal para las coordenadas de $\mathbf{x}_c$. Si planteamos lo mismo para el segmento $\mathbf{x}_0 - \mathbf{x}_2$

$$\begin{aligned} \implies \quad (\mathbf{x}_c - \mathbf{x}_{02}) \cdot \Delta\mathbf{x}_{02} &= 0, \\ \mathbf{x}_{c,02} &= (\mathbf{x}_0 + \mathbf{x}_2)/2, \end{aligned} \tag{3}$$

entonces tenemos dos ecuaciones con dos incógnitas y podemos determinar el centro de la CC $\mathbf{x}_c$.



$$\begin{aligned} \mathbf{A}\mathbf{x}_c &= \mathbf{b}, \\ \mathbf{A} &= \begin{bmatrix} \Delta\mathbf{x}_{01}{}^T \\ \Delta\mathbf{x}_{02}{}^T \end{bmatrix}, \\ \mathbf{b} &= \begin{bmatrix} \Delta\mathbf{x}_{01} \cdot \mathbf{x}_{c,01} \\ \Delta\mathbf{x}_{02} \cdot \mathbf{x}_{c,02} \end{bmatrix}, \end{aligned} \tag{4}$$

El concepto se puede extender a 3 dimensiones. En 3D tendremos 4 puntos y planteamos que el centro de la esfera circunscripta CC debe pertenecer a los planos bisectores de los 3 segmentos orientados

desde el primero $\mathbf{x}_0$

$$\begin{aligned} \mathbf{A}\mathbf{x}_c &= \mathbf{b}, \\ \mathbf{A} &= \begin{bmatrix} \Delta\mathbf{x}_{01}{}^T \\ \Delta\mathbf{x}_{02}{}^T \\ \Delta\mathbf{x}_{03}{}^T \end{bmatrix}, \\ \mathbf{b} &= \begin{bmatrix} \Delta\mathbf{x}_{01} \cdot \mathbf{x}_{c,01} \\ \Delta\mathbf{x}_{02} \cdot \mathbf{x}_{c,02} \\ \Delta\mathbf{x}_{03} \cdot \mathbf{x}_{c,03} \end{bmatrix}, \end{aligned} \tag{5}$$

En general, en $\mathbb{R}^n$ tendremos $n+1$ puntos $\mathbf{x}_j$. Tomando uno arbitrario de ellos, por ejemplo el $\mathbf{x}_0$ y considerando los n segmentos $\mathbf{x}_0 - \mathbf{x}_j$ con $j = 1, n$ y para cada uno de ellos podemos plantear una ecuación y por lo tanto podemos construir un sistema lineal con $\mathbf{A} \in \mathbb{R}^{n \times n}$ y $\mathbf{b} \in \mathbb{R}^n$ para determinar el centro de la **Hiper-Esfera Circunscripta** (HEC).

Dado un punto $\mathbf{x} \in \mathbb{R}^n$ y $n+1$ puntos $\mathbf{x}_j \in \mathbb{R}^n$ podemos determinar si $\mathbf{x}$ se encuentra dentro de la HEC correspondiente a los $\{\mathbf{x}_j\}$ con el siguiente algoritmo

- Construir la matriz $\mathbf{A}$, y el miembro derecho $\mathbf{b}$ de acuerdo a (**??**).
- Resolver el sistema y determinar el centro $\mathbf{x}_c$ de la HEC.
- Determinar el radio de la HEC $R = \|\mathbf{x}_0 - \mathbf{x}_c\|$.
- Determinar si el punto está en la HEC $\|\mathbf{x} - \mathbf{x}_c\| < R$

Consigna: Escribir una función **`bool is_insidecc(MatrixXd &X,VectorXd &xprobe);`** que determina si el punto **`xprobe`** está contenido dentro de la HEC determinada por los puntos **`X`**.

Notas:

- La función debe poder utilizarse para dimensiones **`ndim>=2`**.
- **`X`**: es una matriz **`MatrixXd`** de tamaño **`[ndim,(ndim+1)]`** donde **`ndim`** es la dimensión del problema. Cada columna representa uno de los **`xj`**.
- **`xprobe`**: es un **`VectorXd`** de longitud **`ndim`**.

[Ej. 6] **[lsqfit]** Dada un serie de valores $\mathbf{x}_j$ $j \in [0, N)$ y una serie de valores y_j que representan aproximaciones a la aplicación de una función $y_j = f(x_j)$ queremos hallar una aproximación en una base ϕ_k, para una serie de términos m es decir

$$y_j = \sum_k a_k \, \phi(x_k). \tag{6}$$

Los ϕ_k pueden ser potencias ($\phi_k(x) = x^k$), series trigonométricas $\{1, \sin(x), \cos(x), \sin(2x), \cos(2x), ...\}$ Para ello elegimos los a_k por el método de **mínimos cuadrados** (LSQ). Es decir, en forma matricial podemos poner

$$\mathbf{y} = \mathbf{\Phi}\mathbf{a} \tag{7}$$

donde $\mathbf{y} \in \mathbb{R}^N$, $\mathbf{\Phi} \in \mathbb{R}^{N \times m}$, y $\mathbf{a} \in \mathbb{R}^m$. Los elementos de $\mathbf{\Phi}$ son

$$\Phi_{jk} = \phi_k(x_j). \tag{8}$$

Asumiendo que tenemos más puntos que términos de la expansión ($N \geq m$), el sistema está sobredeterminado. En este caso puede demostrarse que LSQ corresponde a premultiplicar el sistema (**??**) por $\mathbf{\Phi}^T$ es decir

$$\begin{aligned} (\mathbf{\Phi}^T\mathbf{\Phi})\,\mathbf{a} &= (\mathbf{\Phi}^T\mathbf{y}), \\ \mathbf{H}\mathbf{a} &= \mathbf{b}, \end{aligned} \tag{9}$$

es decir que se puede obtener $\mathbf{a}$ resolviendo el sistema lineal.

Consigna: Escribir una función
`void lsqfit(VectorXd &X,VectorXd &Y,basis_t basis, int nterms,VectorXd &a);` que toma un vector de abscisas **`X`** de tamaño **`N`** y un vector de valores de la función **`Y`** del mismo tamaño, y determina los coeficientes de la expansión **`a`** en **`nterms`** términos de la base **`basis`**.

La base basis es un puntero a función, cuyo prototipo es
`typedef double (*basis_t)(double x,int j);`

Ejemplos:

```
1) X=[0,1,2,3],Y=[0,1,2,3], basis=poly {1,x,x^2...},nterms=2   =>     a=[0,1]
2) X=[0,1,2,3],Y=[1,2,3,4], basis=poly [1,x,x^2...],nterms=2   =>     a=[1,1]
3) X=[0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1],
      Y=[1,1.094,1.178,1.250,1.310,1.357,1.389,1.409,1.414,1.404,1.381] (y=sin(x)+cos(x)),
      basis=trigo{1,sin(x),cos(x),sin(2*x),cos(2*x),....}, nterms=3 =>  a=[0,1,1]
```

Ayuda:

- Construir la matriz **`Phi`**.
- Construir las matrices **`H`** y **`b`**.
- Resolver el sistema (usar los métodos **`lu()`** y **`solve()`** de Eigen.

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado **`program.cpp`**. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Hay una función de evaluación, por ejemplo si **`f`** es la función a evaluar tenemos

  ```
  ev.eval1(f,vrbs);
  ```

  `ev.eval1(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario **`f`** y compara la salida del usuario (**`user`**) con respecto a la esperada (**`ref`**). Si la verbosidad (el argumento **`vrbs`**) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

mstorti@galileo/texstuff-1.4.0-70-g09a0bdc4/Tue Jun 22 10:44:39 2021 -0300

Curso de Programación en C++.
TPL5-2017. Trabajo Práctico de Laboratorio. [2018-06-08]

PASSWD PARA EL ZIP: `3W25 6CWX R556`

Ejercicios

[Ej. 1] **[findsum]** Implemente la función `vector<int> findsum(vector<int>& v,int S);` que dado un vector de enteros `v` y un entero `S`, encuentre y retorne un subvector contiguo cuya suma sea `S`. Si no existe ningún subvector con dicha suma, retorne un vector vacía. En caso de haber varios vectores que cumplan retorne el primero y más corto.

Ejemplos:

```
v: [1,2,6,5,8,3,4,6] y S=13 debe retornar [2,6,5]
v: [1,2,6,5,8,3,4,6] y S=15 debe retornar [8,3,4]
v: [1,2,6,5,8,3,4,6] y S=17 debe retornar []
```

[Ej. 2] **[xinyect]** Dada un vector de conjuntos `VS_IN` de enteros, y una función `int f(int)`, copiar en otro vector de conjuntos `VS_OUT` solo aquellos conjuntos para los cuales la función `f` es inyectiva (esto es, que genera imágenes diferentes para todos los elementos del conjunto).

Ejemplo: Si `VS_IN=[ (1,2,3), (2,-2,10), (0,5) ]` y `f` es `f(x)=x^2` entonces `VS_OUT` debería quedar `[ (1,2,3), (0,5) ]`, ya que para el segundo conjunto del vector de entrada hay dos elementos (`2` y `-2`) que generan la misma imagen (`4`), razón por la cual ese conjunto se omite.

Ayuda:

- Por cada conjunto de `S` de `VS_IN`
  - Generar un conjunto auxiliar con las imágenes de la función aplicada a sus elementos
  - Comparar los tamaños de ambos conjuntos para determinar si es debe insertar `S` en `VS_OUT`.

[Ej. 3] **[getjaco]** Consideremos una clase virtual `lin_transform_t` que representa transformaciones lineales afines en un espacio de dimension `n`.

```
class lin_transform_t {
  public:
  virtual int dim()=0;
  virtual void transform(VectorXd &x,VectorXd &y)=0;
};
```

El método `dim()` retorna la dimensión del espacio, y `void transform(VectorXd &x,VectorXd &y);` retorna el resultado `y` de aplicar la transformación a un vector `x`.

Consigna 1: Escribir implementaciones `shift_t`, `ident_t`, `scale_t`, `rotate_t` de acuerdo a las siguientes características

- **shift_t(n,c)**: $y_j = x_j + c$
- **ident_t(n)**: $y_j = x_j$
- **scale_t(n,c)**: $y_j = cx_j$
- **rotate_t(n)**: $y_j = x_k$, con $k = \mathrm{mod}(j-1, n)$, es decir si **x=[3,5,2]** entonces **y=[5,2,3]**.

En todos los casos **int n** es la dimensión del espacio y c es un **double**. Estos valores deberían estar almacenados en el objeto y ser definidos en el constructor.

Consigna 2: Escribir una función **void getjaco(lin_transform_t &lt,MatrixXd &A,VectorXd &b);** que calcula en **A** y **b** el Jacobiano y término constante de la transformación **lt**; es decir **y=A*x+b**.

Para los ejemplos de la **Consigna 1** **getjaco** debe retornar:

- **shift_t**: $A = I, b = [c, c, c, c....]^T$.
- **ident_t**: $A = I, b = 0$.
- **scale_t**: $A = cI, b = 0$.
- **rotate_t**: $b = 0$,

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Por ejemplo si hacemos

```
void check1(lin_transform_t &lt) {
  cout << "================\n";
  MatrixXd A;
  VectorXd b;
  getjaco(lt,A,b);
  cout << "A: " << endl;
  cout << A << endl;
  cout << "b: " << endl;
  cout << b << endl;
}

void check_getjaco() {
  shift_t shift(10,3.0); check1(shift);
  ident_t ident(5); check1(ident);
  scale_t scale(4,8.3); check1(scale);
  rotate_t rotate(5); check1(rotate);
}
```

Debemos obtener:

```
================
A:
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
b:
3
```

```
3
3
3
3
3
3
3
3
3
================
A:
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
```

```
0 0 0 1 0
0 0 0 0 1
b:
0
0
0
0
0
================
A:
8.3   0   0   0
0 8.3   0   0
0   0 8.3   0
0   0   0 8.3
b:
0
```

```
0
0
0
================
A:
0 0 0 0 1
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
b:
0
0
0
0
0
```

Curso de Programación en C++.
TPL0-2019. Trabajo Práctico de Laboratorio. [2019-09-11]

PASSWD PARA EL ZIP: `RFP7 MCW6 J43P`

Ejercicios

[Ej. 1] **[contained]** Dados dos arreglos de enteros **v1,v2** determinar si **v1** está contenido en **v2** o viceversa: **`bool contained(vector<int> &v1,vector<int> &v2);`** Por ejemplo

```
v1:[1,2,3],    v2:[1,2,3,4]   => true
v1:[5,1,2,3],  v2:[2,5]       => true
v1:[5,1,2,3],  v2:[2,5,5]     => false
v1:[1,2,3,4],  v2:[0,1,2,3]   => false
v1:[1,2,3],    v2:[3,2,1]     => true
```

Si un valor está repetido **n** veces en **v1** debe estar repetido al menos **n** veces en **v2** y viceversa (respetar multiplicidad). Por ejemplo en el 3er caso arriba los valores de **v2** están contenidos en **v1** pero no con su multiplicidad, ya que 5 está dos veces en **v2**pero una sola vez en **v1**.

Ayuda:

- Escribir una función **`bool isin(int x,vector<int> &v);`** que busca el elemento **x** en el vector **v**. Si lo encuentra lo eliminar de **v** y retorna **`true`**.
- Escribir una función auxiliar **`bool contained2(vector<int> x,vector<int> y);`** que realiza la tarea indicada pero sólo si **x** está contenido en **y** (es decir no si **y** está incluida en **x**).Para hacer el **`contained2`** recorre los elementos de **v1** y se fija si están en **v2** con **`isin`** y eliminándolo. Si todos los elementos de **v1** están en **v2** entonces retorna **`true`**, si no retorna **`false`**.
  Nota: Para borrar el **j**-ésimo elemento de un vector debe hacer **`v.erase(v.begin()+j)`**.
- Escribir **`contained`** como un *wrapper* que llama a **`contained2`** con **`(v1,v2)`** y **`(v2,v1)`**.

[Ej. 2] **[findsum]** Escribir una función **`bool findsum(vector<int> &v,int sumtarget);`** que dado un vector de enteros **v** y un entero **`sumtarget`** determina si existe alguna tripleta de números **`(x1,x2,x3)`** en **v** tal que la suma de sus cubos $(x_1^3 + x_2^3 + x_3^3)$ es igual a **`sumtarget`**.

Atención: Los enteros **`(x1,x2,x3)`** deben ser distintos.

Ejemplos:

```
sumtarget:36,    v:[1,2,3,4,5,6]    => true   36=1^3+2^3+3^3
sumtarget:37,    v:[1,2,3,4,5,6]    => false
sumtarget:36,    v:[6,5,4,3,2,1]    => true   36=3^2+2^3+1^3
sumtarget:856,   v:[6,8,1,7]        => true   856=8^3+1^3+7^3
sumtarget:293,   v:[6,5,7,4,9,2]    => false
sumtarget:91,    v:[5,7,3,0,6,4]    => true   91=3^3+0^3+4^3
```

Ayuda:

- Hacer un triple lazo sobre los enteros del vector, evitando los valores repetidos. Una posibilidad es hacer los lazos anidados sobre índices `j,k,l` de la siguiente manera
  - `j` en `[0,n-2)`,
  - `k` en `[j+1,n-1)`,
  - `l` en `[k+1,n)`.

  Si los valores `(v[j],v[k],v[l])` satisfacen la condición retorna verdadero.
- Si ninguna tripleta satisface la condición retorna falso.
- Otra posibilidad para evitar los valores repetidos es hacer los tres lazos anidados sobre `j,k,l` evitando las repeticiones con condicionales del tipo `if (j!=k) { ... }`
- *Nota:* Para hacer el cubo de un entero conviene escribir un macro o función auxiliar. No se recomienda usar la función `pow(x,y)` ni tampoco usar el operador `x^3` ya que en C++ tiene otro significado (es el `xor` bit a bit).

[Ej. 3] **[findperm]** Escribir una función `bool findperm(vector<string> vs,string target);` que determina si existe alguna combinación de strings de `vs` que forme el string `target`.

Ejemplos:

```
target=abc         vs=[c,b,a]             => true
target=abcd        vs=[c,b,a]             => false
target=abc         vs=[e,c,b,a]           => true
target=brabraaca   vs=[a,bra,ca,da,bra]   => true
target=bracabra    vs=[a,bra,ca,da,bra]   => true
target=cadabraboom vs=[a,bra,ca,da,bra]   => false
target=adabrabra   vs=[a,bra,ca,da,bra]   => true
target=caada       vs=[a,bra,ca,da,bra]   => true
```

Ayuda:

- Recorremos por fuerza bruta todos las posibles combinaciones de los strings en `vs` y comprobamos si coincide con `target`. Si lo encontramos retornamos `true` caso contrario retornamos `false`.
- Para recorrer las combinaciones utilizamos una estrategia recursiva. Supongamos que tenemos una de las combinaciones `partial` de `m` de los strings, y `rest` es el resto de los strings. Por ejemplo si `vs=[a,bra,ca,da,bra]` y `partial="dabra"`, entonces `rest=[a,ca,bra]`. Podemos generar las combinaciones de 3 strings combinando `partial` con cada uno de los strings en `rest`, es decir `[dabraa,dabraca,dabrabra]`.
- Es decir, escribimos una función recursiva auxiliar `bool findperm(string partial,vector<string> rest,string target);` que
  - Verifica si `partial` coincide con `target`, si es así retorna true.
  - Caso va llamando recursivamente a `findperm` con la concatenación de `partial` con `rest[j]` y pasando el vector `rest` con `rest[j]` eliminado. Si en algún caso retorna `true` entonces `findperm` retorna true. Si se recorren todos los `rest[j]` y ninguna retorna true, entonces retorna `false`.

```
findperm(partial=dabra,rest=[a,ca,bra])
   -> findperm(partial=dabra+a,  rest=[ca,bra])
   -> findperm(partial=dabra+ca ,rest=[a,bra])
   -> findperm(partial=dabra+bra,rest=[a,ca])
```

Curso de Programación en C++.
TPL1-2019. Trabajo Práctico de Laboratorio. [2019-10-03]

PASSWD PARA EL ZIP: VCP5 QRF6 R6QJ

Ejercicios

[Ej. 1] **[int-set]** Queremos representar subconjuntos de los números enteros a través de una clase polimórfica de la forma:

```
class int_set_t {
  public:
  virtual bool isin(int x)=0;
};
```

El método **predicado** `s.isin(x)` indica si `x` pertenece al conjunto o no. Notar que de esta forma se pueden representar conjuntos infinitos, por ejemplo los números impares se pueden representar de la forma

```
class odd_t : public int_set_t {
  public:
  bool isin(int x) {
    return modulo(x,2)==1;
  }
};
```

donde `modulo(x,2)` es el resto de dividir `x` por 2 (como el operador `%` pero siempre retorna 0/1).

Consigna: Instanciar las siguientes clases derivadas de `int_set_t`:

```
class even_t : public int_set_t { ... };
class all_t : public int_set_t { ... };
class empty_t : public int_set_t { ... };
class positive_t : public int_set_t { ... }
class stride_t : public int_set_t {
  int start,end,step;
  ...
};
class vector_t : public int_set_t {
  vector<int> w;
  ...
};
class intersection_t : public int_set_t {
  unique_ptr<int_set_t> s1,s2;
```

```
  ...
};
class union_t : public int_set_t {
  unique_ptr<int_set_t> s1,s2;
  ...
};
class difference_t : public int_set_t {
  unique_ptr<int_set_t> s1,s2;
  ...
};
class complement_t : public int_set_t {
  unique_ptr<int_set_t> s1;
  ...
};
```

- **stride_t**, dados los valores **start,end,step** debe retornar los elemenots **k=start+j*step** tales que **k>=start && k<end**, por ejemplo si **start=10,step=3,end=18** los valores son **[10,13,16]**.
- **vector_t** simplemente contiene un **vector<int> w** que contiene los elementos aceptables. En este caso la función **isin(x)** simplemente debe buscar el elemento **x** en **w**.
- **intersection_t** contiene dos punteros a subobjetos **s1,s2** de la clase **int_set_t**. Esta clase debe hacer la intersección de los subobjetos. Los punteros son guardados en **unique_ptr<int_set_t> s1,s2** recordar que lo punteros subyacentes se pueden obtener haciendo **s1.get()**.
- Las clases **union_t** y **difference_t** también hacen las correspondientes operaciones de conjuntos y son binarias (contienen dos subobjetos **s1** y **s2**).
- La clase **complement_t** corresponde al complemento del conjunto **s1** es decir todos los elementos del conjunto universal que no está en **s1**.
- Todos los campos dato (**w**, **s1**, **s2**, **start**, **step**, **end**) deben ser públicos.
- Al definir una dada clase **XYZ** hay que activar su evaluación definiendo el macro correspondiente **HAS_XYZ** por ejemplo

```
class positive_t : public int_set_t { ... }
#define HAS_POSITIVE_T
```

[Ej. 2] **[nilpot]** Queremos representar permutaciones de los elementos de un vector de enteros. Por ejemplo la permutación de tipo *shift* desplaza los elementos de un vector una posición hacia el final **shift1([0,1,2,3,4]) => [4,0,1,2,3]**. Estas operaciones las representamos a través de una clase virtual

```
class perm_t {
  public:
  virtual void mix(vector<int> &w) { }
};
```

El método virtual **mix(w)** debe retornar la permutación correspondiente en el vector de enteros **w**.

Consigna 1: Implementar las siguientes clases

```
class shiftm_t : public perm_t {
  int m;
  ...
};
class oddeven_t : public perm_t { };
class reflex_t : public perm_t { };
class shift2_t : public perm_t {
  int m1,m2;
  ...
};
```

- **shiftm_t** hace un shift pero de salto **m** por ejemplo si **m=2** **shiftm([0,1,2,3,4]) => [3,4,0,1,2]**.
- **oddeven_t** intercambia las posiciones sucesivas:0 con 1, 2 con 3, **2*j** con **2*j+1**. Por ejemplo **oddeven([0,1,2,3,4]) => [1,0,3,2,4]**. (Nota: Si el número de elementos es impar el último queda en su lugar.)
- **reflex_t** hace una reflexión con respecto a la posición media. Por ejemplo si **reflex([0,1,2,3,4,5]) => [5,4,3,2,1,0]**. (Nota: si el número de elementos es impar el del centro queda en su lugar.)
- **shiftm1m2_t**: Asumiendo que **m1+m2<=n** (donde **n** es la longitud del vector) hace un *shift* de una posición en los primeros **m1** posiciones y un shift de una posición en los siguientes **m2**. Si **m1+m2<n** entonces los elementos restantes quedan en su posición.
  Por ejemplo: Si **m1=3,m2=2** entonces **shift2([0,1,2,3,4,5,6,7,8,9])** **=>[2,0,1,4,3,5,6,7,8,9]**

Nota: Después de implementar las clases deben activar su evaluación definiendo el macro correspondiente **HAS_XYZ** por ejemplo

```
class reflex_t : public perm_t { ... };
#define HAS_REFLEX_T
```

Nilpotencia: Puede verse que una permutación es una operación *nilpotente*, es decir si es aplicada repetidamente, existe un cierto entero **np>=1** para el cual la aplicación de **np** veces es equivalente a la identidad. Por ejemplo: la permutación **oddeven** es de nilpotencia 2, y el **shift** de una posición para **N** objetos es de nilpotencia **N**. La identidad tiene nilpotencia 1.

Consigna 2: Implementar una función **int nilpot(perm_t &perm,int N);** que dada una permutación **perm** y una longitud del vector **N** retorna su nilpotencia.

Ayuda: Construir un vector de longitud **N** inicialmente con la identidad **[0,1,2,...,N-1]** y aplicar la permutación repetidamente hasta que el vector vuelva a ser la identidad.

Curso de Programación en C++.
TPL2-2019. Trabajo Práctico de Laboratorio. [2019-10-23]

PASSWD PARA EL ZIP: 3QW6 GPCH F9CV

Ejercicios

[Ej. 1] **[surface]** Una superficie en el espacio puede representarse como una transformación de una región en el plano $(u, v) \in \mathbb{R}^2$ al espacio $(x, y, z) \in \mathbb{R}^3$. Por ejemplo la esfera puede escribirse como el mapeo de las coordenadas $u = \phi$ (coordenada *azimutal* o *longitudinal*), $v = \theta$ llamada *elevación* o *latitud*,

$$\begin{aligned} x &= R\cos(u)\ \cos(v), \\ y &= R\cos(u)\ \sin(v), \\ z &= R\sin(u). \end{aligned} \tag{1}$$

Esta transformación mapea el rectángulo $-\pi \leq u \leq \pi$, $-\pi/2 \leq v \leq \pi/$. Otras ejemplos de superficies son,

- **Elipsoide** de ejes a, b, c:

$$\begin{aligned} x &= a\cos(u)\ \cos(v), \\ y &= b\cos(u)\ \sin(v), \\ z &= c\sin(u). \end{aligned} \tag{2}$$

- **Cono** con eje alineado con el eje z y apertura angular θ. Usamos coordenadas polares en el plano: $u = \rho$ y $v = \phi$

$$\begin{aligned} x &= u\ \cos(v), \\ y &= u\ \sin(v), \\ z &= \frac{u}{\tan(\theta)}. \end{aligned} \tag{3}$$

El objetivo del ejercicio es escribir una clase virtual pura **`surface_t`** que representa una superficie genérica y una función **`area(surface_t &surf...)`** que calcula el área en una región de la superficie.

```
class surface_t {
  public:
  virtual void generate(VectorXd &uv,VectorXd &xyz)=0;
};

double area(surface_t &surf,int N=100,
double umin=0.0,double umax=1.0,
double vmin=0.0,double vmax=1.0);
```

- La clase **`surface_t`** tiene un método **`void generate(VectorXd &uv,VectorXd &xyz);`** que es el que representa el mapeo entre las coordenadas bidimensionales **`uv`** y las tridimensionales **`xyz`**.

- La función **`area()`** debe calcular la superficie mapeada por el rectángulo **`umin<=u<=umax`**, **`vmin<=v<=vmax`** en el plano **`(u,v)`** en la superficie 3D. Para esto debe dividir el rectángulo en **`N*N`** rectángulos, mapear los vértices de cada uno de estos rectángulos usando el método **`generate()`** y acumula la superficie en 3D. Recordar que para calcular el área de un triángulo en 3D definido por los 3 vértices x_0, x_1, x_2 conviene calcular las aristas a, b y utilizar la norma del producto vectorial

$$a = x_1 - x_0, \quad b = x_2 - x_0, \quad A = \tfrac{1}{2}\|b \times a\| \tag{4}$$

Nota1: Recordar que Eigen sólo acepta el producto vectorial **`a.cross(b)`** cuando **`a,b`** son **`Vector3d`** es decir de longitud fija. Si hacen los cálculos con vectores dinámicos entonces deben copiar estos en temporarios y utilizar el producto vectorial sobre éstos. La otra posibilidad es calcular el producto vectorial en forma explícita con las componentes de los **`VectorXd`**.
Nota2: Se provee el código fuente **`eigen6.cpp`** desarrollado en clase donde se calcula el área de un elipsoide como base para el ejercicio.

Verificaciones:

- **Plano:** Generado como $x = u, y = v, z = c + au + bv$. El área debe dar $\sqrt{1 + a^2 + b^2} \cdot A_{uv}$, donde A_{uv} es el área del rectángulo en el plano u, v: $A_{uv} = (u_{\max} - u_{\min})(v_{\max} - v_{\min})$.
- **Esfera:** Debe dar $4\pi R^2$.
- **Elipsoide:** Para un elipsoide oblongo con $a = b = 1, c = 2$ debe dar $A = 21.4784$. Para un elipsoide oblato con $a = b = 1, c = 0.5$ debe dar $A = 8.67185$. La fórmula analítica para el área se puede consultar aquí:
  https://en.wikipedia.org/wiki/Ellipsoid
- **Cono:** El área lateral del cono es $\pi R^2 / \sin(\theta)$.
- **Catenoide:** Es una figura de revolución alrededor del eje z definida así

$$x = a\cosh(v/c)\,\cos(u), \quad y = a\cosh(v/c)\,\sin(u), \quad z = v. \tag{5}$$

  Esta figura es muy interesante ya que representa la figura de área mínima cuando $a = c$ (en realidad es un **extremo** (**máximo** o **mínimo** local)). Calcular el área de la catenoide entre dos círculos de radio $R = 1$ a una distancia de $2L$ con $L = 0.6$ y para varios valores de c que controla la curvatura de la catenoide. Si variamos c entre 0.1 y 2 manteniendo R =cnst (es decir $a = R/\cosh(L/c)$) entonces tenemos los siguientes valores,

c	a(neck rat)	Area/Area(cilindro)
0.100	0.0049570	0.8624530
0.195	0.0920060	0.9159020
0.290	0.2486650	0.9507090
0.385	0.4030730	0.9548920
0.480	0.5295420	0.9452070
0.575	0.6267030	0.9351750
0.670	0.7000330	0.9295590
0.765	0.7554770	0.9284290
0.860	0.7978250	0.9304480
0.955	0.8306050	0.9342700
1.050	0.8563430	0.9389260
1.145	0.8768330	0.9438070
1.240	0.8933620	0.9485730
1.335	0.9068580	0.9530470

```
1.430   0.9180020   0.9571540
1.525   0.9272980   0.9608760
1.620   0.9351260   0.9642250
1.715   0.9417740   0.9672260
1.810   0.9474650   0.9699110
1.905   0.9523710   0.9723130
2.000   0.9566280   0.9744640
```

- Notar que el área está normalizada por el área correspondiente del cilindro $4\pi RL$.
- El área tiene extremos locales para $c = 0.385$ y $c = 0.765$ (un máximo y un mínimo, respectivamente).
- El valor de a en este caso coincide con el **neck ratio** es decir la relación entre el radio mínimo en el angostamiento ($z = 0$) y los extremos ($z = \pm L$).
- Además el área decrece uniformemente para $c \to 0$ ya que en ese caso el área tiende a la de dos discos en $z = \pm L$ es decir $A = 4\pi$.

[Ej. 2] **[optimize]** Escribir una clase **`double optimize(objfun_t &F,VectorXd &X);`** que toma una función objetivo **`F`** y busca el mínimo **`X`**, es decir tal que **`F(Y)>=F(X)`** para todo **`Y`**. La función objetivo está definida a través de una clase virtual pura

```
class objfun_t {
  public:
  virtual double eval(VectorXd &X)=0;
};
```

El algoritmo de optimización es muy simple, basado en gradiente. Dado un $X \in \mathbb{R}^N$ calcular el gradiente $g \in \mathbb{R}^N$ de F en X por diferencias finitas centradas

$$g_j = \frac{F(X + \epsilon e_j) - F(X - \epsilon e_j)}{2\epsilon} \tag{6}$$

donde ϵ es un parámetro a elegir por el usuario y e_j es el versor en la dirección j, es decir ($e_{jk} = \delta_{jk}$. El algoritmo consiste en avanzar una cierta distancia en la la dirección del gradiente con relajación ω, $X' = X - \omega g$. Esto se realiza iterativemente hasta que el gradiente del funcional g esté por debajo de una tolerancia especificada $\|g\| < \mathrm{tol}$.

Nota: El algoritmo puede ser divergente para ω grande, por lo tanto se recomienda probar con un ω pequeño (por ejemplo **`1e-3`**) e incrementarlo mientras el algoritmo sea estable.

Consigna: Escribir una función
`double optimize(objfun_t &F,VectorXd &X,double epsln=1e-3,double tol=1e-3);` que realiza la tarea indicada.

Verificaciones: Buscar el mínimo de los funcionales indicados a continuación

- Dado un **`X0`** de dimensión **`n`** podemos formar el funcional cuadrático

$$F = c + a\|X - X_0\|^2 \tag{7}$$

El mínimo se obtiene por supuesto en $X = X_0$ y el valor mínimo del funcional es c.
Probar en 3D con $X0 = (1, 1, 1)$, $a = 1$, $c = 2$ y otras variantes. **Consigna:**

- **Ley de refracción:** Queremos encontrar el tiempo mínimo para que una partícula recorra la distancia entre dos posiciones A y B (ver figura). La zona entre estos puntos está dividida en N bandas de ancho dado d_j ($N = 3$ en la figura) donde en cada banda la velocidad es constante e igual a c_j. Encontrar el recorrido óptimo, es decir las distancias L_0, L_1, L_2 tales que el tiempo sea mínimo. La distancia en la dirección paralela a las bandas $L = x_b - x_a$ es un dato.
  Ayuda: EL funcional a minimizar en este caso es el tiempo total de recorrido

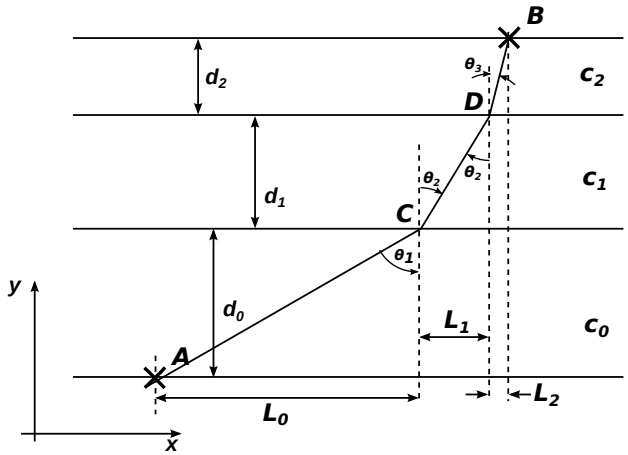
$$F = \sum_j \frac{(\text{distancia})_j}{(\text{velocidad})_j} = \sum_j \frac{\sqrt{d_j^2 + L_j^2}}{c_j} \tag{8}$$

  Las incógnitas son un vector de longitud $N - 1$,

$$X = (L_0, L_1, \ldots, L_{N-2}) \tag{9}$$

  ya que la longitud en la última capa está determinada por la suma, es decir

$$L_{N-1} = L - \sum_{j=0}^{N-2} L_j. \tag{10}$$

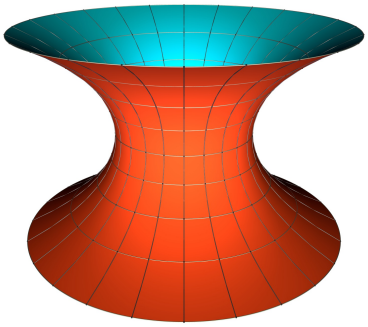


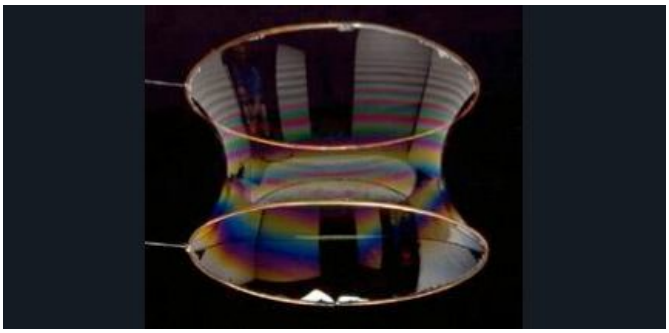
  Puede demostrarse que la condición que debe verificarse para que la solución sea un extremo es que se cumplan las **leyes de refracción de Snell**,

$$\frac{\sin(\theta_j)}{c_j} = \frac{\sin(\theta_{j+1})}{c_{j+1}} \tag{11}$$

  Consigna: Comprobar que, por ejemplo, si tenemos dos capas y $d_1 = d_2 = 1$, y $c_1 = 1, c_2 = 2$, $L = 1$. La solución es $L_1 = 0.29947$ $L_2 = 0.70053$.
- **Superficie mínima:** Dados dos círculos de radio R ubicados a una distancia $2L$ encontrar la superficie de área mínima que los vincula.
  Nota: Las superficies mínimas se pueden visualizar físicamente usando pompas de jabón, ya que la energía de una configuración está dada por su área y como el sistema busca la configuración de energía mínima, resulta ser que las pompas de jabón toman la forma que da área mínima. En el caso de dos círculos paralelos como se describe en este ejemplo la superficie de área mínima resulta ser la catenoide. Para más información ver
  - Catenoide: https://youtu.be/UuS2qcpeJ6c

- Superficies mínimas: https://youtu.be/dAyDi1aa40E, https://youtu.be/NscxxwDqq5s





Para representar la superficie vamos a utilizar una expresión de tipo superficie de revolución pero con una expansión en cosenos

$$\begin{aligned} x &= \rho(v)\ \cos(u), \\ y &= \rho(v)\ \sin(u), \\ z &= v, \end{aligned} \tag{12}$$

con

$$\begin{aligned} \rho(v) &= R(1 - a\cos(\zeta) - b\cos(3\zeta) - c\cos(5\zeta)\dots) \\ \zeta &= \pi v/2L, \end{aligned} \tag{13}$$

Los valores incógnitas para optimizar son $X = (a, b, c...)$

Consigna: Para $R = 1.0, L = 0.6$ y $m = 1$ términos en el desarrollo la superficie mínima se obtiene con los parámetros $X = (0.262601)$ y la superficie mínima es $A = 7.00032$. Con $m = 2$ parámetros obtenemos una superficie de $A = 6.99125$ y los parámetros óptimos son $X = (0.266887, -0.0115996)$.

Curso de Programación en C++.
TPL0-2021. Trabajo Práctico de Laboratorio. [2021-04-28]

PASSWD PARA EL ZIP: VMRQ 6GWF 45PW

Ejercicios

[Ej. 1] **[isinverse]** Podemos representar una **permutación** de **n** objetos con un vector de enteros de longitud **n** cuyos elementos sean los enteros **[0,n)** reordenados de alguna forma. Por ejemplo si **n=4** entonces el vector sin permutar **[0,1,2,3]**, entonces la permutación **p=[1,2,3,0]** representa que el objeto 0 fue movido a la posición 1, el 1 al 2 y así siguiendo (una rotación hacia la derecha). Es decir el objeto **j** es movido a la posición **p[j]**. La rotación **inversa** es aquella que vuelve los objetos a su lugar, es decir en el caso anterior sería **q=[3,0,1,2]** porque manda el 1 al 0, el 2 al 1... (una rotación a la izquierda).
Otro ejemplo: La rotación inversa de **p=[1,0,2,3]** (intercambia los dos primeros objetos entre sí, y los dos últimos quedan inalterados) es ella misma **q=p=[1,0,2,3]** ya que sólo hay que volver a invertir los dos primeros.

Consigna: Escribir un predicado `bool isinverse(vector<int> &p,vector<int> &q);` que determina si las permutaciones **p,q** son una la inversa de la otra.

Ejemplos:

```
p=[1,2,3,0],   q=[3,0,1,2]   => true
p=[1,0,2,3],   q=[1,0,2,3]   => true
p=[1,2,0],     q=[2,0,1]     => true
p=[1,2,0],     q=[1,2,0]}    => false
p=[1,4,0,2,3], q=[2,0,3,4,1] => true
```

Ayuda: Recorrer los valores de **p** y verificar que si en la posición **j** de **p** está el valor **k**, entonces en la posición **k** de **q** debe estar el valor **j**.

[Ej. 2] **[mkmonotone]** Dado un vector de enteros **v** escribir una función `void mkmonotone(vector<int> &v,vector<vector<int> > &vv);` que particiona al vector en una serie de subvectores **vv** de tal forma que cada uno de ellos es monótono ascendente.

```
v:[1,2,0,1,2,3,2,3,4,5,3,4,5,6]},
=> vv:[[1,2],[0,1,2,3],[2,3,4,5],[3,4,5,6]]},

v:[5,9,3,9,6,4,1,1,2,4,1,9]},
=> vv:[[5,9],[3,9],[6],[4],[1,1,2,4],[1,9]]},

v:[1,6,4,6,8,4,7,2,2,1,0,0,3]},
=> vv:[[1,6],[4,6,8],[4,7],[2,2],[1],[0,0,3]]},
```

Ayuda:

- Mantener un índice `j` sobre `v` inicialmente en 0.
- Ir incrementando `j` mientras `v[j]<=v[j+1]` e ir insertando los elementos en un vector temporario `tmp`.
- Cuando ya `j` no se puede avanzar porque se viola la condición de monotonicidad insertar el temporario en `vv` y arrancar nuevamente desde la última posición.

[Ej. 3] **[incluido]** Dados dos vectores de enteros `v1,v2` escribir una función `bool incluido(vector<int> &v1,vector<int> &v2);` que determinar si el vector `v2` está incluido en el vector `v1`.

Ejemplos:

```
v1=[0,1,2,3,4,6]              v2=[1,2,3,4]    => true
v1=[0,1,2,3,4,6]              v2=[1,2,3,4,5]  => false
v1=[5,9,3,9,6,4,1,1,2,4,1,9]  v2=[3,9,6,4,5]  => false
```

Ayuda:

- Si la longitud de `v2` es mayor que la de `v1` entonces no es necesario hacer nada ya que claramente no puede estar contenido.
- Caso contrario, si las dimensiones son `n1,n2` entonces el vector `v2` debe estar contenido en `v1` en un cierto rango `[k,k+n2)`. La estrategia es ir buscando para cada `k` y chequear si a partir de `k` encontramos el vector `v2` o no.
- Las posiciones de `k` a buscar van desde 0 hasta `n1-n2` ya que más allá de esa posición no hay espacio para que esté contenido `v2`.
- Entonces, para cada una de esas posiciones `k` verificar si los `n2` elementos siguientes de `v1` (o sea el rango `[k,k+n2)`) es igual (elemento a elemento) a `v1`.

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más arriba; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Salvo indicación contraria pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado `program.cpp`. En principio sólo tienen que escribir el cuerpo de las funciones pedidas.
- Para cada ejercicio hay dos funciones de evaluación, por ejemplo si `f` es la función a evaluar tenemos

```
ev.eval<j>(f,vrbs);
hj = ev.evalr<j>(f,seed); // para SEED=123 debe dar Hj=170
```

`j` es el número de ejercicio, por ejemplo para el ejercicio 1 tenemos las funciones (`eval<1>` y `evalr<1>`). La primera `ev.eval<j>(f,vrbs);` toma una serie de casos de prueba de entrada, le aplica la función del usuario `f` y compara la salida del usuario (`user`) con respecto a la esperada (`ref`). Si la verbosidad (el argumento `vrbs`) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

```
m: 10, k: 3
T(ref): (10 (7 (4 1) 1) (4 1) 1)
T(user): (10 (7 (4 1) 1) (4 1) 1)
EJ1|Caso0. Estado: OK
```

- **ucase**: Además las funciones **eval()** tienen dos parámetros adicionales:
  Eval::eval(func_t func,int vrbs,int ucase);
  El tercer argumento **ucase** (caso pedido por el usuario), permite que el usuario seleccione uno solo de todos los ejercicios para chequear. Por defecto está en **ucase=-1** que quiere "hacer todos". Por ejemplo **ev.eval4(prune_to_level,1,51);** corre sólo el caso 51.
- **Archivo con casos tests JSON:** Los casos test que corre la función **eval<j>** están almacenados en un archivo **test1.json** o similar. Es un archivo con un formato bastante legible. Abajo hay un ejemplo. **datain** son los datos pasados a la función y **output** la salida producida por la función de usuario. **ucase** es el número de caso.

```
{ "datain": {
   "T1": "( 0 (1 2) (3 4 5 6) )",
   "T2": "( 0 (2 4) (6 8 10 12) )",
   "func": "doble" },
  "output": { "retval": true },
  "ucase": 0 },
```

- La segunda función **evalr<j>** es el chequeo que llamamos **SEED/HASH**. La clase evaluadora genera una serie de contenedores a partir de la semilla **seed**, se los pasa a la función del usuario **f()**. Las respuestas de la **f()** van siendo procesadas por la función interna de hash que genera un **checksum H** de las respuestas. Por ejemplo para el primer ejercicio si **seed=123** entonces el checksum es **H=523**. Una vez que el alumno termina su tarea se le pedirá que corra la función **evalr<j>()** de la clase evaluadora con un valor determinado de la semilla **seed** y se comprobará que genere el valor correcto del checksum **H**.

Curso de Programación en C++.
TPL1-2021. Trabajo Práctico de Laboratorio. [2021-06-02]

PASSWD PARA EL ZIP: 24C5 85V4 56FQ

Ejercicios

[Ej. 1] **[bjgame]** La consigna consiste en escribir jugadores y un simulador para el juego del BlackJack (o también llamado 21). Usaremos una versión muy simplificada de las reglas.

- Hay sólo dos jugadores **p1** y **p2**.
- El juego consiste en una secuencia de manos en la cual los jugadores van pidiendo cartas del mazo.
- Los valores de las cartas van del 1 al 10 y todos los valores del 1 al 10 son igualmente probables. Los valores son generados en forma aleatorio.
- En cada mano cada uno de los dos jugadores va pidiendo cartas del mazo tratando de sumar la cantidad lo más elevada posible pero que no exceda de 21. El jugador puede decidir **plantarse** (es decir no pedir más cartas) por ejemplo cuando la suma llega a 18. Si el jugador *"se pasa"* de 21 su mano es inválida.
- Una vez que cada uno de los jugadores hizo su mano, pueden pasar alguna de las siguientes situaciones.
  - Si las manos de ambos jugadores son inválidas entonces empatan.
  - Si la mano de **p1** es inválida y la de **p2** no, entonces gana **p2** y viceversa.
  - Si la mano de ambos es válida entonces gana el que tiene la suma de cartas más alta.
  - Si ambos tienen la misma suma es empate.
- El juego consiste en realizar una cierta cantidad alta de manos (por ejemplo 1000) y determinar cual es la probabilidad de que gane **p1** o **p2** o haya un empate.
- Los jugadores se implementan derivando la clase virtual pura siguiente,

```
class player_t {
public:
  virtual bool wants_card(vector<int> &hand,
       history_t &me,history_t &you)=0;
  virtual string description() { return "unknown"; }
};
```

- El método que determina la estrategia es **`bool wants_card(vector<int> &hand,...);`** que retorna **`true`** si el jugador quiere otra carta o no. El **`vector<int> &hand`** contiene las cartas en el orden que fueron saliendo.
- El método **`string description()`** es auxiliar y retorna un string que debe ser una descripción de la estrategia utilizada. Tiene una definición por defecto, por lo tanto no es obligatorio implementarla.
- La consigna consiste en implementar una serie de estrategias de jugadores que explicitaremos a continuación.
- La estrategia más simple en plantarse en un cierto número de cartas, por ejemplo 4. Como las cartas van del 1 al 10, en promedio cada carta vale 5, entonces típicamente con 4 cartas llegamos a 20, muy cerca del máximo 21.

Consigna1: implementar una clase derivada **`fixed_t`** con un constructor **`fixed_t(int ncards)`** que debe tener un parámetro interno **`ncards`** que es el número de cartas que pide el jugador.

- Los parámetros **`history_t &me,history_t &you`** de **`wants_card()`** serán explicados después. Contienen la historia de juego de este jugador y el adversario. Entonces por ejemplo si determinamos que el adversario tiende a plantarse en sumas bajas podemos diseñar una estrategia que se plante en sumas bajas también. Total de todas formas ganaremos la mano, sin correr riesgo innecesario de pasarnos.
- **Consigna2:** Escribir una función **`void playbjgame(player_t &p1,player_t &p2, vector<double> &stats,int nhands);`** que toma dos jugadores por referencia y un número da manos y va haciendo jugar a los dos jugadores entre sí **`nhands`** veces.
  - El valor de retorno **`stats`** es la probabilidad de
    - **`stats[0]`** probabilidad de que **`p1`** gane (**`win1`**)
    - **`stats[1]`** probabilidad de que empaten (**`draw`**)
    - **`stats[2]`** probabilidad de que **`p2`** gane (**`win2`**)
  - Para jugar cada mano la función debe generar la mano de **`p1`**, después la de **`p2`**. Para "pedir una carta" puede usar la función

    ```
    int get_card() { return rand()%10+1; }
    ```

  - O sea que para generar cada mano **`playbjgame()`** debe ir llamando a **`wants_card()`** del jugador y pasándole la mano actual **`vector<int> cards`**. El jugador decide en base a la mano actual si se planta o no. En el caso de **`fixed_t(3)`** se planta (retorna **`false`**) si el número de cartas actual ya es 3.
- **Consigna3:** Utilizar la función **`playbjgame()`** para determinar las probabilidades de ganar entre los jugadores **`fixed_t(n)`** con **`n=2,3,4,5,6`**. Como verificación el resultado de hacer jugar a **`fixed_t(3)`** con **`fixed_t(2,4,5,6)`** es,

  ```
  fixed3 vs. fixed2: win1 0.611, draw 0.043, win2 0.346
  fixed3 vs. fixed4: win1 0.565, draw 0.120, win2 0.313
  fixed3 vs. fixed5: win1 0.717, draw 0.147, win2 0.135
  fixed3 vs. fixed6: win1 0.801, draw 0.158, win2 0.040
  ```

  Quiere decir que la mejor estrategia (para los **`fixed_t`**) es **`ncards=3`** ya que le gana a todos los otros.
- **Nota:** Para los jugadores **`fixed_t`** las historias **`history_t h1,h2;`** son irrelevantes, de manera que por el momento no es necesario implementarlas. Sólo basta con declarar a los objetos y pasárselos al jugador.
- Una estrategia más elaborada (**`target_t`**) consiste en plantarse cuando la suma de la mano es mayor que un valor **`targetsum`** predefinido. A primera vista el valor más apropiado para **`targetsum`** parece estar cerca de 16 ya que si ya tenemos 16 y pedimos una carta en promedio saldrá 5 con lo cual justo tendríamos el valor máximo 21.
  Consigna4: implementar una clase derivada **`target_t`** con constructor **`target_t(int targetsum)`** que realiza esta estrategia, es decir se planta cuando la suma de la mano es mayor o igual que **`targetsum`**.
- Hacer competir todos las variantes de **`target_t(n)`** entre si. Verificar que el óptimo es **`target_t(18)`**. Es decir, si enfrentamos a todos **`target_t`** entre sí podemos ver que **`target_t(18)`** vence a todos los otros **`target`**,

  ```
  | PROB(P1=wins)      | PROB(draw) | PROB(P2=wins)    |
  | target(18)* 0.582 | draw 0.024 | target(11) 0.395 |
  ```

```
| target(18)* 0.555 | draw 0.033 | target(12) 0.412 |
| target(18)* 0.528 | draw 0.051 | target(13) 0.421 |
| target(18)* 0.504 | draw 0.072 | target(14) 0.424 |
| target(18)* 0.471 | draw 0.104 | target(15) 0.425 |
| target(18)* 0.440 | draw 0.143 | target(16) 0.417 |
| target(18)* 0.413 | draw 0.187 | target(17) 0.400 |
| target(18)* 0.384 | draw 0.263 | target(19) 0.353 |
| target(18)* 0.430 | draw 0.296 | target(20) 0.274 |
| target(18)* 0.512 | draw 0.332 | target(21) 0.156 |
```

(**Nota:** en la tabla un `*` al lado del jugador indica que gana).

Estrategia adaptativa: Podemos tratar de diseñar un jugador *adaptativo* que modifica su `targetsum` de acuerdo a la estrategia del oponente.
Nota: *Lo que sigue es la motivación. Si quieren pueden saltar directamente a la* ***Consigna5****, más abajo.*

- Si sabemos que el oponente es (por ejemplo) un `target_t(13)` entonces la ganancia máxima no la obtiene `target_t(18)` (cuya probabilidad de ganar contra `target_t(13)` es 0.528, sino que la máxima probabilidad la tiene `target_t(16)` y es 0.594.

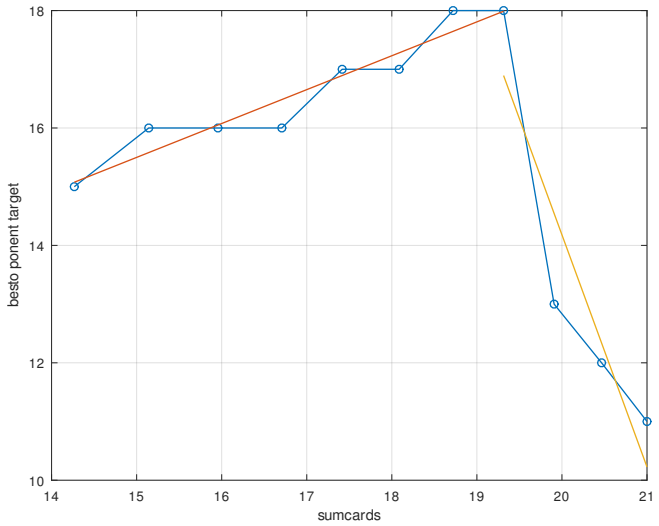
```
| PROB(P1=wins)     | PROB(draw) | PROB(P2=wins)     |
| target(13)* 0.632 | draw 0.092 | target(11) 0.276  |
| target(13)* 0.539 | draw 0.107 | target(12) 0.354  |
| target(13) 0.369  | draw 0.111 | target(14)* 0.520 |
| target(13) 0.328  | draw 0.095 | target(15)* 0.577 |
| target(13) 0.326  | draw 0.080 | target(16)* 0.594 |
| target(13) 0.356  | draw 0.065 | target(17)* 0.579 |
| target(13) 0.422  | draw 0.050 | target(18)* 0.528 |
| target(13)* 0.524 | draw 0.036 | target(19) 0.440  |
| target(13)* 0.655 | draw 0.027 | target(20) 0.318  |
| target(13)* 0.807 | draw 0.019 | target(21) 0.174  |
```

  Entonces esto sugiere que podemos desarrollar una estrategia **adaptativa**. Estudiando la historia de juego del adversario podemos tratar de adivinar su **target** y buscar cual es el target más adecuado, es el que tiene más probabilidad de ganar. Para esto relevamos la tabla siguiente, donde para cada player `target_t(n)` calculamos la probabilidad de pasarse `P(pasarse)` y la suma de cartas en el caso exitoso `sumcards`. Obviamente para un target `n`muy bajo (por ejemplo 11) la probabilidad de pasarse es muy baja, pero la suma de cartas también es en promedio baja: 14.27. Por el contrario contrario para un target alto (por ejemplo 21) la probabilidad de pasarse es muy alta (0.823) pero si no se pasa la suma de cartas es muy alta (justamente 21 en este caso).

```
target(n) P(pasarse) sumcards  BESTP2 P(win2)
   11     0.00000    14.26700   15    0.70700
   12     0.00000    15.14200   16    0.64900
   13     0.01700    15.95600   16    0.59400
   14     0.05100    16.70700   16    0.54200
   15     0.10400    17.41600   17    0.49400
   16     0.17500    18.08600   17    0.45500
   17     0.26900    18.72000   18    0.40900
```

18	0.41300	19.31400	18	0.42500
19	0.50900	19.90800	13	0.52300
20	0.66000	20.46600	12	0.66200
21	0.82300	21.00000	11	0.82200

Un jugador adaptativo **`adaptative_t p1`** puede diseñarse así, debe tratar de deducir la estrategia de su adversario (adivinar su **`p2.target`**) y en base a eso buscar el target más apropiado. Para adivinar el target del adversario, no podemos mirar su código, pero lo podemos tratar de adivinar a través de su historial de jugadas **`h2`**. Los historiales son simplemente **`vector<vector<int»`** donde cada entrada **`j`** son las cartas que jugó en esa mano. Entonces podemos esperar un cierto número de manos, por ejemplo 100, calcular la suma de cartas promedio **`sumcards`** de **`p2`** y con es valor el que más se aproxima en la 3ra columna de la tabla. A partir de ahí, deducimos cuál es el *target* del oponente (columna 1) y por lo tanto cuál es nuestro valor óptimo de *target* (columna 4). En la figura la curva azul de lineas y círculos muestra el target óptimo como función del sumcards.
Vemos que si el **`target`** del oponente es bajo (por ejemplo 11) **`sumcards`** será relativamente bajo también (14.27) entonces el óptimo es tomar **`target=15`** con lo cual probabilidad de ganar asciende a 0.707. Por el contrario si el oponente usa un target muy alto (por ejemplo 20) entonces tiene **`sumcards`** muy alto (20.4) pero falla muchas veces (66 %), entonces basta con tomar un target muy bajo (12) porque el 66 % de las veces el oponente se va a pasar.



Para evitar tener que cargar la tabla podemos aproximar el **`target`**, en función del **`sumcards`**, como dos tramos lineales (en rojo y anaranjado en la figura)

```
target=int(0.577*sumcards+6.842);   // si sumcards<19.5
target=int(-1.831*sumcards+49.462); // si sumcards>=19.5
```

- **Consigna5:** Implementar un jugador **`adaptative_t`** de acuerdo a la siguiente estrategia:
  - En las primeras 100 jugadas usa **`target=18`**.
  - A partir de las 100 jugadas calcula el **`sumcards`** del oponente y en base a ese valor usa las expresiones,
    ```
    target=int(0.577*sumcards+6.842);   // si sumcards<19.5
    target=int(-1.831*sumcards+49.462); // si sumcards>=19.5
    ```
    para adaptar su **`target`**. Por las dudas al finalizar el cálculo del **`target`** llevarlo al rango

`[11,18]`.

- *Cálculo del* **`sumcards`** *del oponente:* El jugador recibe en **`wants_card(hand,me,you);`** la historia de las jugadas en este partido, **`me`** son las mías, **`you`** las del oponente. Para implementar el **`adaptative_t`** sólo necesitamos las del oponente. Las historias son simplemente vectores de vectores de enteros:

```
typedef vector<vector<int>> history_t;
history_t h1,h2;
```

- Por supuesto para que los jugadores reciban las historias hay que modificar **`playbjgame()`** para que efectivamente mantenga las historias de ambos (hasta ahora sólo las declara pero están vacías). Para ello después de cada mano tiene que apendizar las manos de cada jugador a su historia. Notar que cuando llame a los jugadores tiene que pasarle en orden apropiado las historias. Es decir para
  - para **`p1`**: **`me`** es **`h1`** y **`you`** es **`h2`**
  - para **`p2`**: **`me`** es **`h2`** y **`you`** es **`h1`**

  Es decir que las llamadas deberían ser así,

```
history_t h1,h2;
// ... Dentro de un lazo le va preguntando a p1 y p2 si quieren
// más cartas
while (...) p1.wants_card(cards1,h1,h2);
while (...) p2.wants_card(cards2,h2,h1);
// Determina si gano p1 o p2 y actualiza stats[]...
// Apendiza cards1 a h1...
// Apendiza cards2 a h2...
```

- **Consigna6:** Realizar las tablas de probabilidad entre todos los jugadores: **`fixed_t(2-6)`**, **`target_t(11-21)`**, **`adaptative_t`**.

[Ej. 2] **[linearops]** Consideremos operaciones lineales homogéneas sobre vectores en $\mathbb{R}^n \to \mathbb{R}^n$, es decir están identificadas de la forma $y = Ax$, donde $A \in \mathbb{R}^{n \times n}$. Sin embargo, queremos representar estos operadores sólo por su acción sobre los vectores, es decir por la operación de aplicar el operador a un vector. Consideraremos entonces los operadores como clases derivadas de la clase virtual pura,

```
typedef vector<double> vd_t;
typedef vector<vd_t> vvd_t;

class linearop_t {
public:
  virtual void apply(vd_t &y,vd_t &x)=0;
  string description() { return "unknown"; }
};
```

A lo largo del ejercicio usaremos los **`typedef vd_t`** y **`vvd_t`** para vectores de dobles y vectores de vectores.

El método **`description()`** es opcional y debe retornar un string que describe a la acción de ese operador.

Implementar operadores:

- **Consigna1:** Implementar la clase **`zero_t`** que corresponde a la matriz $A = 0$, es decir por lo tanto que para todo **`x`** debe retornar el vector nulo.
- **Consigna2:** Implementar la clase **`identity_t`** que corresponde a la matriz identidad, es decir que $y = x$.
- **Consigna3:** Implementar la clase **`scale_t(a)`** que corresponde a la matriz $A = aI$, es decir un múltiplo escalar de la identidad. La operación corresponde a $y = ax$.
- **Consigna4:** Implementar la clase **`rot_t(int s)`** que hace un *shift* cíclico de los elementos del vector en **`s`** posiciones, es decir si **`x=[1,2,3]`** y **`s=1`** entonces debe dar **`y=[2,3,1]`**.
- **Consigna5:** Implementar la clase **`reflex_t(int axis)`** que cambia de signo la componente **`axis`**. Por ejemplo si **`x=[5,3,6,9]`** y **`axis=2`** entonces debe dar **`y=[5,3,-6,9]`**. Geométricamente corresponde a una reflexión según el plano perpendicular al eje **`axis`**.

Predicados:
Ahora nos podemos preguntar como implementar predicados, por ejemplo si es SPD (simétrica y positiva definida), si preserva la norma. También dados dos operadores si son iguales, etc...

- **Consigna6:** Implementar el predicado
  `bool is_spd(linearop_t &A,vvd_t &vectors,double tol=0.0);` que determina si el operador lineal **`A`** es SPD. Recordemos que una matriz se dice que es SPD si $(x, Ax) > 0,\ \forall x \in \mathbb{R}^n, x \neq 0$ donde $(x, y) = \sum_j x_j y_j$ denota el producto escalar. La verificación no se puede hacer sobre todos los vectores del espacio, porque son infinitos, pero lo hacemos sobre un cierto conjunto de vectores **`vvd_t &vectors`**. La función debe entonces recorrer los vectores de **`vectors`**, aplicarle el operador y luego hacer el producto escalar con **`x`** y verificar. Si es negativo al menos para un sólo **`x`** entonces retorna **`false`**. Caso contrario retorna **`true`**.
  - Verificar que la identidad es SPD.
  - Verificar que **`scale_t`** es SPD si **`a>0`**.
  - Verificar que **`scale_t`** no es SPD si **`a<0`**.
  - Verificar que **`rot_t(n)`** no es SPD a menos que **`n=0`** o **`n`** es un múltiplo de la dimensión, ya que en ese caso es la identidad.
  - Verificar que **`reflex_t`** no es SPD.

  Debido a errores de redondeo puede ser que el resultado de ligeramente negativo, por eso dejamos un argumento **`tol`** es decir que en el predicado se debe verificar que $(x, Ax) \geq -$**`tol`**. Si **`tol=0`** entonces estamos usando la versión exacta y si ponemos por ejemplo **`tol=1e-10`** le estamos tolerancia para el error de redondeo.
  Nota: En el **`program.cpp`** ya está implementada una función **`mkrandvectors()`** que genera los vectores aleatorios de prueba. Por ejemplo para verificar que **`scale_t(5.0)`** es SPD deberíamos hacer,

```
vvd_t vectors;
mkrandvectors(vectors);
scale_t smas(1.0),smenos(-1.0);
cout << "is_spd(smas) " << is_spd(smas,vectors) << endl;
cout << "is_spd(smenos) " << is_spd(smenos,vectors) << endl;
```

  y debe dar,

```
$ ./program.bin
is_spd(smas) 1
is_spd(smenos) 0
```

- **Consigna7:** Implementar el predicado
  `bool is_normpres(linearop_t &A,vvd_t &vectors,double tol=0.0);` que verifica si el operador *"preserva la norma"*, es decir que $|Ax| = |x|,\ \forall x$. Las reflexiones y rotaciones preservan la norma.
  - Verificar que la identidad preserva la norma.
  - Verificar que **`scale_t(a)`** no preserva la norma, salvo para **`a=-1,1`**.
  - Verificar que **`rot_t`** preserva la norma.
  - Verificar que **`reflex_t`** preserva la norma.

  La versión con tolerancia verifica que: $||y| - |x|| <$**`tol`**, donde $y = Ax$.
- **Consigna8:** Implementar el predicado **`bool are_equal(linearop_t &A1,linearop_t &A2, vvd_t &vectors,double tol=0.0);`** que toma dos operadores **`A1`**, **`A2`** y verifica si los operadores son iguales. Para ello recorre los vectores en **`vectors`** le aplica el operador **`A1`** y **`A2`** a cada uno de ellos y verifica si los resultados son iguales. Es decir si $y_1 = A_1x$, $y_2 = A_2x$ debe ser $y_2 = y_1$.
  La versión con tolerancia verifica que: $|y_2 - y_1| <$**`tol`**.
  Verificar que **`identity_t`** y **`scale_t(1)`** y **`rot_t(0)`** son todos iguales.

Composición de operadores:
Dada una serie de operadores $A_1, A_2...A_n$ podemos definir el operador compuesto A tal que $Ax = A_n(A_{n-1}(...(A_1x)))$, es decir vamos aplicando A_1 y luego A_2 hasta A_n.

Consigna9: Implementar la clase derivada **`compose_t`** que contiene un vector de punteros a operadores **`vector<linearop_t*> ops`** de manera que el método **`apply`** consiste en ir aplicando los operadores de **`ops`** (**Nota:** tal vez haga falta usar vectores temporarios.)

```
class compose_t : public linearop_t {
public:
  vector<linearop_t*> ops;
  void apply(vd_t &y,vd_t &x) {
    // Aplica los operadores de ops a x...
  }
};
```

- Verificar que la composición de un **`scale_t(5)`** y **`scale_t(0.2)`** es igual a un **`identity_t`**.
- Verificar que si la composición de **`n`** **`rot_t(1)`** es igual a un **`identity_t`**, donde **`n`** es la dimensión del vector.
  Nota: la dimensión de todos los vectores de prueba es 3, de manera que debe construir una composición de 3 **`rot_t(1)`** y chequear que ese operador es la identidad.
- Verificar que la composición de 2 **`reflex_t(axis=0)`** es igual a un **`identity_t`**, ya que dos reflexiones a través del mismo plano deja la identidad. Lo mismo para los otros planos, es decir 2 **`reflex_t(axis=1)`** o 2 **`reflex_t(axis=2)`** también deben dar la identidad.

mstorti@galileo/texstuff-1.4.0-67-ge52a7899/Tue Jun 22 08:48:50 2021 -0300