Dante Cpp (DanteCpp)

UNA LIBRERIA C PER LE MATRICI

31 January 2014

Non penso, che al giorno d'oggi una libreria come quella che sto per presentarvi, possa avere la benché minima utilità. D'altronde tra Matlab, Octave, Scilab etc, etc... nei nostri desk sono ormai innumerevoli gli ambienti che ci supportano nel calcolo. Poi c'è python con il suo pacchetto scipy, che non ha nulla da invidiare alle piattaforme ideate ad-hoc, per il calcolo numerico.
Malgrado la mole di software già a disposizione, per far le stesse cose, che la qui presente fa, ho deciso comunque di scrivere questa collezione di funzioni, principalmente perché fuori pioveva e m'annoiavo!
Cosi vagando per il portale, mi sono imbattuto in quest'articolo: Una libreria C per i numeri razionali e mi son detto: be' dalla noia non si esce pensando a cosa fare, ma facendo!

Tipi e inizializzazioni

In questa libreria ci sono due tipi fondamentali definiti nell'header **matrix.h**, il primo rappresenta i numeri reali:

```
typedef double Real;
```

Il tipo Real consente di cambiare la precisione con cui rappresentiamo i reali. È sufficiente sostituire **double** con **float** per dimezzare la precisione o con **long double** per raddoppiarla.

Il secondo tipo di dato, altro non è che il fulcro di tutta la libreria, si chiama Matrix ed è così definito:

```
typedef struct
{
    int rows;  int columns;
    Real ** values;
}
Matrix;
```

Bravo Matrix, un'applauso d'incoraggiamento!!!
Se pur si descrive da se, voglio dedicargli due righe, rows e columns indicheranno rispettivamente il numero di righe e di colonne della matrice, puntata da values.

Ora abbaiamo i contenitori, è tempo di riempirli. Per farlo abbiamo a disposizione una vasta gamma di funzioni, vediamole:

```
Matrix * matrix_new( int, int );
Matrix * matrix_new_identity( int );
Matrix * matrix_new_zero( int, int );
Matrix * matrix_new_scalar( int, Real );
Matrix * matrix_new_values( int, int, ... );
```

La prima funzione inizializza semplicemente una variabile di tipo Matrix, allocando dinamicamente una matrice di dimensioni, rows x columns, in fine restituisce un puntatore alla struttura creata. Se andassimo ad'esplorare la matrice cosi creata potremmo trovarci di tutto poiché la matrice appena allocata non è ancora inizializzata.

Il seguente programma d'esempio, potrebbe stamparci una matrice 2x3 con ogni entrata a zero oppure con valori del tutto aleatori in maniera indeterminata.

```
#include <stdio.h>
#include <matrix.h>
int main()
{
   Matrix * m = matrix_new( 2,3 );
   for( int row = 0; row < m->rows; row++)
       for( int column = 0; column < m->columns; column++)
           printf("%.2f",m->values[row][column])
               ;
   matrix_destroy(m);
   return 0;
}
```

Quando si esplora una matrice in maniera cosi diretta (con i cicli for) senza ricorrere alle funzioni di libreria, fare molta attenzione ai limiti dell'interazione; poiché si potrebbe finire in zone di memoria non allocate. E ritrovarci nello stdout, un seg fault anziché la soluzione del nostro problema! Comunque, a si la funzione matrix_destroy() fa esattamente il lavoro inverso alla matrix_new(), essa dealloca la matrice che gli si passa come argomento dalla memoria, per ogni chiamata di matrix_new() ci deve essere la rispettiva chiamata mi matrix_destroy(). Più in generale per ogni chiamata di funzione che restituisce un'oggetto di tipo Matrix *, è necessario chiamare la matrix_destroy(). Le altre funzioni matrix_new_*() oltre che creare il nuovo oggetto lo inizializzano in svariate maniere, che non spiegherò per filo e per segno, comunque i nomi sono intuitivi, per che è un po arrugginito con le matrici può far riferimento a questo articolo molto chiaro, Una passeggiata tra le matrici.

Manipolazione delle matrici

Le funzioni di manipolazione definite in matrix_manipolation.c, permettono di ottenere particolari sub matrici. Vale la pena esaminarne alcune, come ad'esempio:

```
Matrix * mat = matrix_get_minor( r, M, c )
```

mat sarà la matrice ottenuta dalla matrice M saltando le la riga r e la colonna c. Mente la funzione:

```
Matrix * mat = matrix_get_sub(x1,x2,M,y1,y2)
```

consente di selezionare a propria discrezione la sottomatrice da estrapolare.

```
Matrix * matrix_get_minor( int, Matrix *, int );
Matrix * matrix_get_sub( int, int, Matrix *, int, int );
Matrix * matrix_get_row( Matrix *, int );
Matrix * matrix_get_column( Matrix *, int );
Matrix * matrix_get_diagonal( Matrix * );
void matrix_put_sub( Matrix *, int, int, Matrix * );
void matrix_put_row( Matrix *, int, Matrix * );
void matrix_put_column( Matrix *, int, Matrix * );
void matrix_put_diagonal( Matrix *, Matrix * );
void matrix_filter( Matrix *, Real (*)(Real) );
```

La funzione:

```
matrix_filter()
```

prende come secondo argomento una funzione da Real a Real e la applica ad'ogni entrata della matrice che come primo argomento. Per esempio se volessimo calcolare la radice quadrata d'ogni elemento della matrice m scriveremmo:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <matrix.h>
int main()
{
   srand(time(NULL));
   Matrix * m = matrix_new(4,5);
   matrix_fill_random(m);
   matrix_print(m);
   printf("\n");
```

```
   matrix_filter(m,sqrt);
   matrix_print(m);
   matrix_destroy(m);
   return 0;
}
```

fico no!!!
La funzione matrix_fill_random(), funziona al meglio se inseminiamo il rand() con srand(/*valore non determinabile*/). Essa appartiene a una famiglia definita in matrix.c, queste funzioni servono a inizializzare matrici precedentemente allocate, ecco una panoramica della famiglia matrix_fill_*():

```
void matrix_fill( Matrix *, ... );
void matrix_fill_random( Matrix *);
void matrix_fill_real( Matrix *, Real );
void matrix_fill_triangular_up( Matrix * );
void matrix_fill_triangular_down( Matrix * );
```

Avremmo modo di esaminarle nei prossimi esempi, molto utile è la funzione matrix_print() che stampa semplicemente la matrice nello standard output. Possiamo modificare la precisione della stampa grazie alla direttiva di preprocessing:

```
#define PRINT_FORMAT "%.2f "
```

in questo caso, ogni oggetto di tipo Real verrà stampato sino alla seconda cifra decimale. Di seguito un algoritmo macchinoso ma istruttivo, per creare una matrice identità 4x4:

```
int main()
{
    Matrix * m = matrix_new_zero(4,4);
    Matrix * d = matrix_new(1,4);
    matrix_fill_real(d,1.0);
    matrix_put_diagonal(d,m);
    matrix_print(m);
    matrix_destroy(m);
    matrix_destroy(d);
    return 0;
}
```

Operazioni

Le operazioni matriciali supportate da libmatrix sono dichiarate in matrix_operations.h, anche le operazioni non hanno bisogno di commenti:

```
Matrix * matrix_add( Matrix *, Matrix * );
Matrix * matrix_scalar_mul( Real, Matrix * );
Matrix * matrix_trasposition( Matrix * );
Matrix * matrix_multiplication( Matrix *, Matrix * );
Real matrix_trace( Matrix * );
Real matrix_determinant( Matrix * );
Matrix * matrix_inverse( Matrix * );
Matrix * matrix_inverse_gauss( Matrix * );
Matrix * matrix_cofactors( Matrix * );
Matrix * matrix_adjunct( Matrix * );
```

Quindi passiamo subito ad un'esempio, il programmino che segue tanto per cambiare è un'altro modo macchinoso per ottenere un'identità. La matrice mat vien riempita con la funzione matrix_fill(), a quest'ultima si passano gli elementi con cui riempire per mezzo di stringhe, che rappresentano le righe della matrice da riempire. Una volta inseriti i valori bisogna passarli NULL come ultimo parametro (attenti a non mettere più spazzi del dovuto). La funzione matrix_inverse_gauss() restituisce l'inversa del suo argomento, utilizzando l'eliminazione di Gauss, però difetta per matrici di grandi dimensioni, quindi io consiglio l'uso di matrix_inverse() che lavora usando la matrice aggiunta.

```
int main()
{
    Matrix * mat = matrix_new(3,3);
    matrix_fill( mat, "1 1 2",
                      "2 1 2",
                      "1 -2 1", NULL );
    Matrix * inv = matrix_inverse(mat);
    Matrix * idt = matrix_multiplication(mat,inv);
    matrix_print(idt);
    matrix_destroy(mat);
    matrix_destroy(inv);
    matrix_destroy(idt);
    return 0;
}
```

Dopo l'inizializzazione di mat ne calcoliamo l'inversa che chiamiamo inv, in fine moltiplichiamo le due matrici per ottenere un'identità. Non scordiamoci di distruggere gli oggetti creati!

Interrogare le matrici

Queste funzioni son dichiarate in matrix_query.h, prendono come valore una o due matrici è restituiscono un valore booleano in risposta all'affermazione esplicitata come nome della funzione.

```
int matrix_same_size( Matrix *, Matrix *);
int matrix_is_square( Matrix * );
int matrix_is_vector( Matrix * );
int matrix_is_equal( Matrix *, Matrix * );
int matrix_is_sigular( Matrix * );
int matrix_can_mul( Matrix *, Matrix * );
```

devo mettervi in guardia dalla matrix_is_equal(), poiché come afferma Dirty nel punto 8 del suo articolo sui numeri razionali, scrivere funzioni per comparare dei numeri pseudo reali in maniera affidabile non è cosa affatto facile, quindi se noi arriviamo al medesimo risultato con due algoritmi diversi poi li compariamo con matrix_is_equal il risultato di quest'ultima può non essere attendibile, anzi non lo è senz'altro!

```
int main()
{
     Matrix * mat = matrix_new(3,3);
     matrix_fill( mat, "1 1 2",
                       "2 1 2",
                       "1 -2 1", NULL );
     Matrix * inv1 = matrix_inverse_i(mat);
     Matrix * adj = matrix_adjunct(mat);
     Matrix * inv2 = matrix_scalar_mul(1/matrix_determinant(mat),adj);
     if( matrix_is_equal(inv1,inv2) )
         printf("cool!");
     else
         printf("freez!");
     /*matrix_destroy(...)*/
     return 0;
}
```

In futuro, magari implementerò la comparazione eseguendo una differenza per vedere se appartiene a un range di tolleranza accanto a 0! E con queste vi ho esposto tutte le funzioni della libreria, ora vediamo come possono essere usate per il calcolo, risolvendo alcuni esercizi...

Esercizi

Per questa sezione ho raccolto un po d'esercizi dal portale, iniziamo con la risoluzione del circuito resistivo proposto nell'articolo Calcoli circuiti con vettori e matrici di Mathcad Express:

```
#include <matrix.h>
int main()
{
    Real R1 = 10.0 , R2 = 50.0 , R3 = 10.0 , R4 = 60.0 , R5 = 10.0 ;
    Real E1 = 15.0 , E2 = 12.0 ;
    Matrix * M = matrix_new_values( 5,5            , R1 , R2 , 0.0, 0.0, 0.0
                                                   , 0.0, -R2, R3 , R4 , 0.0
                                                   , 0.0, 0.0, 0.0, R4 , R5
                                                   , 1. , -1., -1., 0.0, 0.0
                                                   , 0.0, 0.0, 1. , -1., 1. );
   Matrix * V = matrix_new_values( 5,1  ,E1 ,0.0 ,E2 ,0.0 ,0.0 );
   Matrix * invM = matrix_inverse(M);
   Matrix * I = matrix_multiplication(invM,V);
   matrix_print(I);
   matrix_destroy(M);
   matrix_destroy(V);
   matrix_destroy(invM);
   matrix_destroy(I);
   return 0;
}
```

Ed ora Puro Kirchhoff da Uno sguardo sul ponte di Wheatstone:

```
#include <matrix.h>
int main()
{
    Real R1 = 470. , R2 = 330. , R3 = 220. , R4 = 680. , R5 = 1500. ;
    Real E = 24. ;
    Matrix * M = matrix_new_values( 6,6            , 0.0, 0.0, -1.0, -1.0, 0.0, 1.0
                                                   , -1.0, 0.0, 0.0, 1.0, -1.0, 0.0
                                                   , 0.0, -1.0, 1.0, 0.0 , 1.0 , 0.0
                                                   , 0.0 , 0.0, R3 , -R4 , -R5, 0.0
                                                   , -R1 , R2 , 0.0 , 0.0 , R5 ,0.0
                                                   , 0.0 , -R2, -R3 , 0.0 , 0.0 , 0.0);
   Matrix * V = matrix_new_values( 6,1  ,0.0 ,0.0 ,0.0 ,0.0 ,0.0 ,E );
   Matrix * invM = matrix_inverse(M);
   Matrix * I = matrix_multiplication(invM,V);
```

```
   matrix_print(I);
   matrix_destroy(M);
   matrix_destroy(V);
   matrix_destroy(invM);
   matrix_destroy(I);
   return 0;
}
```

Installazione e compilazione

Potete clonare il pacchetto con git:

`git clone https://github.com/DanteCpp/libmatrix`

Se siete sotto linux, potete compilare con makefile, spostatevi nella cartella libmatrix, poi:

```
sudo make
```

Il makefile sposta l'archivio con i file oggetto in /usr/bin e crea una coppia degli header in /usr/include, quindi volendo rimuovere la libreria è sufficiente digitare i seguenti comandi:

```
rm /usr/bin/libmatrix.a
rm /usr/include/matrix*h
```

se non volete usare git: potette scaricare libmatrix direttamente da browser al link: libmatrix

per utilizzare libmatrix è sufficiente includere <matrix.h> e in qualsiasi anfratto del filesystem potete compilare con:

```
gcc main.c -lmatrix -lm
```

Estratto da "http://www.electroyou.it/mediawiki/index.php?title=UsersPages:Dantecpp:una-libreria-c-per-le-matrici"