Interpolazione Polinomiale

Esperienze di Programmazione

Gioacchino Mirko Matonti

Settembre 2019

1 Introduzione al problema

Sia $p(x)$ il polinomio di grado n interpolante i dati

x	x_o	x_1	$\cdots$	$\cdots$	x_n
y	y_0	y_1	$\cdots$	$\cdots$	y_n

forniti da una funzione $y = f(x)$ con $y_i = f(x_i)$, $x_i \in [a, b]$ per $i = 0, ..., n$, $x_i \neq x_j$, $i \neq j$. Siamo interessati a capire l'accuratezza con cui $p(x)$ interpola la funzione $f(x)$ per tutti i punti x. Formuliamo matematicamente il problema

$$p(x_i) = \sum_{k=0}^{n} \alpha_k \phi_k(x_i) = y_i, \quad 0 \leq i \leq n \tag{1}$$

dove $\phi = \{\phi_0(x), ..., \phi_n(x)\}$ è una base dello spazio vettoriale dei polinomi a coefficienti reali di grado minore od uguale ad n. L'esistenza ed unicità del polinomio $p(x)$ sono garantite dalla scelta di nodi distinti [3]. Di seguito sono analizzate in dettaglio le diverse scelte della base ϕ per il calcolo dei coefficienti a_i, rispetto alla funzione

$$f(x) = \frac{1}{1 + x^2} \tag{2}$$

2 Algoritmi

2.1 Base dei monomi

Scegliamo come base $\phi(x_j) = x^j, 0 \leq i \leq n$, la matrice associata prende il nome di matrice di Vandermonde che sappiamo essere non singolare[1]. I coefficienti di $p(x)$ possono essere trovati risolvendo il sistema

$$\begin{pmatrix} 1 & x_0 & \cdots & x_0^n \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ \vdots \\ \vdots \\ f(x_n) \end{pmatrix} \tag{3}$$

Tale metodo di risoluzione non è in genere preferibile, la matrice dei coefficienti può risultare mal condizionata ed inoltre la risoluzione richiede al piu $O(n^3)$ operazione matematiche.

2.2 Forma di Newton

Se scegliamo come base $\phi_0(x) = 1$, $\phi_j(x) = \prod_{i=0}^{j-1} x - x_i$, $1 \leq j \leq n$ la matrice associata è triangolare inferiore, il calcolo dei coefficienti avviene risolvendo il sistema (4) utilizzando il metodo di sostituzione in avanti con costo computazionale di $O(n^2)$.

$$\begin{pmatrix} 1 & & & & \\ 1 & x_1 - x_0 & & & \\ 1 & x_2 - x_0 & (x_2 - x_0)(x_2 - x_1) & & \\ \vdots & \vdots & \vdots & \ddots & \\ 1 & x_n - x_0 & (x_n - x_0)(x_n - x_1) & \cdots & \prod_{i=0}^{n-1} x_n - x_i \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ \vdots \\ \vdots \\ f(x_n) \end{pmatrix} \tag{4}$$

La particolarità della forma di Newton è la possibilità di aumentare il grado del polinomio interpolante senza dover ricalcolare i nodi precedenti.

2.3 Forma di Lagrange

Nella forma di Lagrange il polinomio assume la forma

$$p(x) = \sum_{j=0}^{n} f(x_j) L_j(x)$$

dove

$$L_j(x) = \prod_{k=0, k \neq j}^{n} \frac{x - x_k}{x_j - x_k}$$

Nei punti $x = x_i$ il calcolo risulta essere particolarmente efficiente e non richiede operazioni matematiche poiché

$$L_j(x_i) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

e quindi $p(x_i) = y_i \;\; 0 \leq i \leq n$. La forma di Lagrange è molto utile in caso siamo interessati alla valutazione del polinomio in determinati punti piuttosto che al calcolo dei coefficienti in se, difatti nel caso di $x \neq x_i$ il costo computazione per il calcolo di $p(x)$ è $O(n^2)$

3 Analisi dei risultati

Analizziamo ora il caso della funzione $f(x) = \frac{1}{1+x^2}$, troviamo il polinomio di interpolazione di grado 4 a partire dal set di dati nel intervallo [-2,2]:

x	-2	-1	0	1	2
f(x)	0.2	0.5	1	0.5	0.2

Utilizzando uno degli algoritmi sopra proposti (ricordiamo che $p(x)$ è unico) otteniamo :

$$p(x) = 1 - 0.6x^2 + 0.1x^4$$

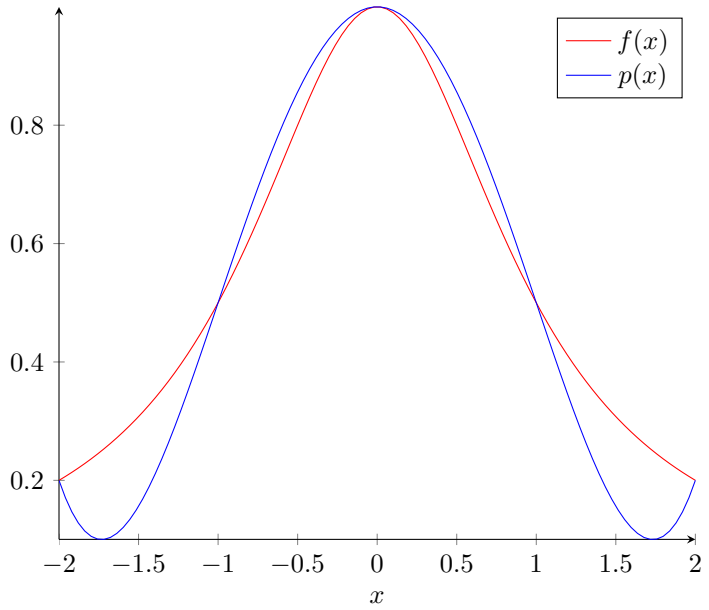

Figura 1: interpolazione della funzione con 5 nodi equidistanti.

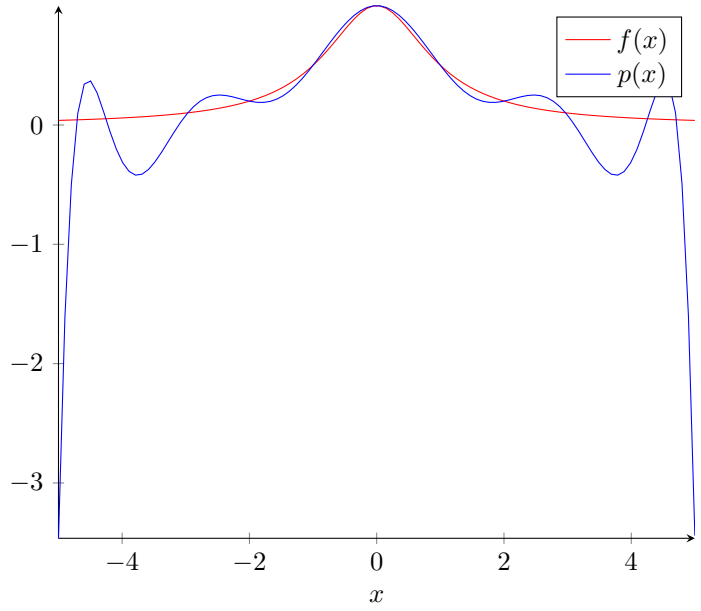

Figura 2: interpolazione della funzione con 11 nodi equidistanti.

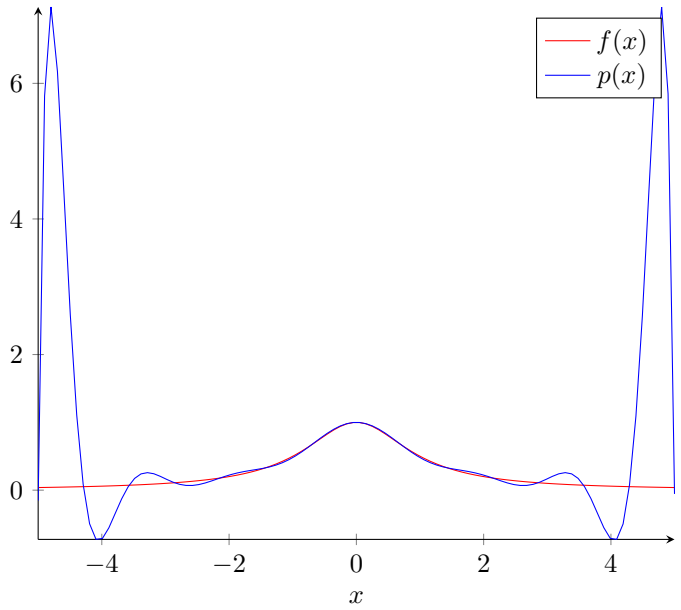

Figura 3: interpolazione della funzione con 15 nodi equidistanti.

Si potrebbe pensare che più nodi corrispondano ad un approssimazione migliore ma non è sempre cosi, dai risultati precedenti notiamo che all'aumentare del numero dei nodi l'interpolazione risultante oscilla in ampiezza verso gli estremi dell'intervallo, questo fenomeno è dovuto alla scelta di nodi equidistanti e viene definito Fenomeno di Runge.

3.1 Resto dell'interpolazione

Definiamo il resto dell' interpolazione di $f(x)$ con il polinomio $p(x)$ come la funzione

$$r(x) = f(x) - p(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i) \tag{5}$$

Si puo dimostrare come nel caso di nodi equi distanti

$$x_i = a + ih \quad con \quad h = \frac{b-a}{n}, \qquad i = 0, 1, ..., n$$

tale errore nel caso della funzione f(x) tende all'infinito all'aumentare del grado del polinomio.

$$\lim_{n->\infty} \left(\max_{x \in [a,b]} |f(x) - p(x)| \right) = +\infty$$

3.2 Scelta dei nodi

Dalla formula del resto (5) notiamo che l'errore è composto dal prodotto di due fattori, sul primo, $\frac{f^{(n+1)}(\xi)}{(n+1)!}$ non possiamo intervenire (sostanzialmente perché ξ non è noto a priori) di conseguenza l'unica maniera per intervenire sull'errore indipendentemente dalla f è fornire una maggiorazione alla componente $\prod_{i=0}^{n}(x - x_i)$ cosa che si può fare scegliendo nodi di Chebyshev.
Tali nodi vengono anche definiti come zeri dei polinomi di Chebyshev[6], polinomi che sono le componenti della seguente successione polinomiale:

$T_0(x) = 1$
$T_1(x) = x$
$T_2(x) = 2x^2 - 1$
$T_3(x) = 4x^3 - 3x$
$T_4(x) = 8x^4 - 8x^2 + 1$
$\dots$
$\dots$
$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$

In formula, questi nodi per un intervallo [-1,1] sono calcolabili come:

$$x_i = \cos(\frac{2i-1}{2n}\pi), \quad 0 \leq i \leq n$$

mediante una trasformazione possiamo calcolarli per un generico intervallo [a,b]

$$x = \frac{b+a}{2} - \frac{b-a}{2}t$$

ottenendo,

$$x_i = \frac{b+a}{2} - \frac{b-a}{2}\cos(\frac{2i-1}{2n}\pi), \quad 0 \leq i \leq n \tag{6}$$

Analizzando nuovamente l'interpolazione utilizzando i nodi di Chebyshev:

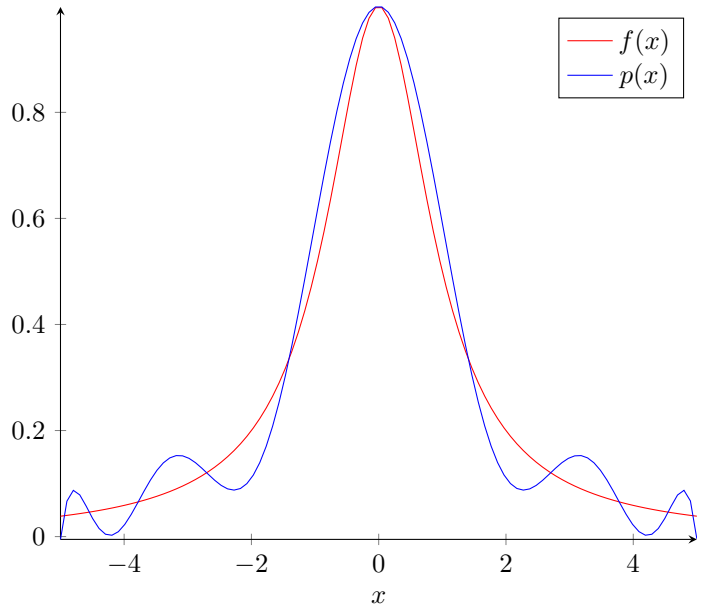

Figura 4: interpolazione della funzione con 11 nodi di Chebyshev.

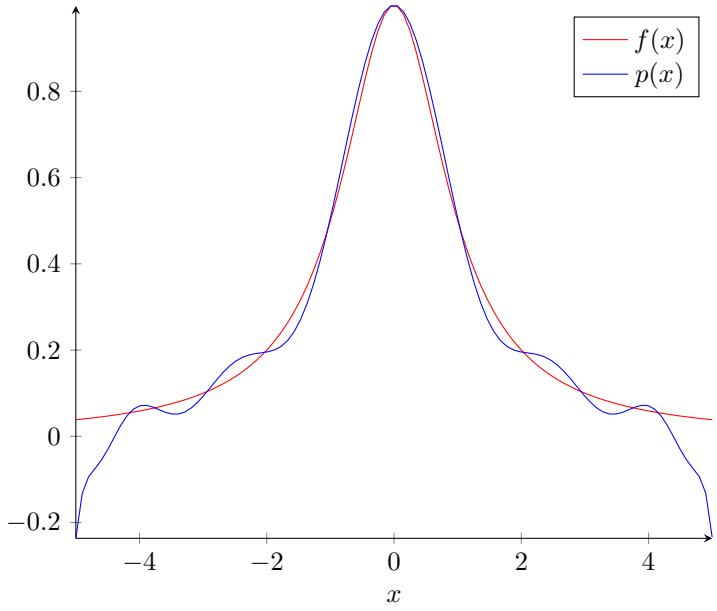

Figura 5: interpolazione della funzione con 15 nodi di Chebyshev.

Notiamo come le oscillazioni agli estremi dell'intervallo vengono arginate con la scelta dei nodi di Chebyshev, il fenomeno è meglio visibile nel caso del resto r(x) calcolato prima con i nodi equidistanti poi con quelli di Chebyshev.

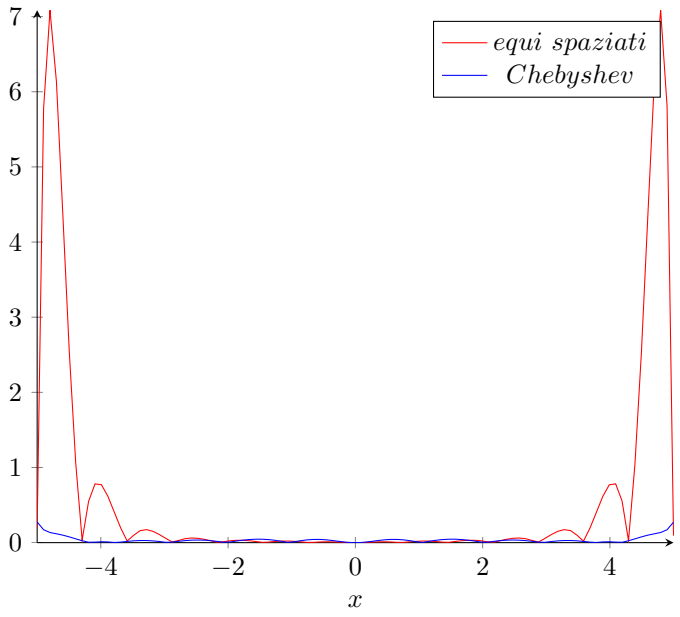

Figura 6: Errore di interpolazione con nodi equispaziati e di Chebyshev.

3.3 Interpolazione spline

Un altro modo per ovviare al fenomeno di Runge è quello di interpolare la funzione $f(x)$ utilizzando dei polinomi di grado basso, in opportuni sottointervalli, cioè utilizzando le funzioni cosiddette polinomiali a tratti, tra le quali le più comunemente utilizzate sono le funzioni spline.
Si definisce spline di grado m, relativa ai punti $x_0, x_1, \ldots, x_k$ una funzione $S_m(x) : [x_0, x_k] \to R$ tale che:

1. $S_m(x)$ è un polinomio di grado non superiore ad m in ogni intervallo $[x_{i-1}, x_i]$ con $i = 1, 2, \ldots, k$;
2. $S_m(x_i) = y_i$ con $i = 0, 1, \ldots, k$;
3. $S_m(x_i) \in C^{m-1}([x_0, x_k])$.

Prendiamo in considerazione solo la funzione spline naturale cubica, ovvero con m=3, questa funzione spline è a tratti cubica e due volte differenziabile nell'intero intervallo. Inoltre, la relativa

derivata seconda è zero nei punti estremi. La funzione assume la forma:

$$s_3(x) = \begin{cases} a_0 + b_0(x - x_0) + c_0(x - x_0)^2 + d_0(x - x_0)^3 & [x_0, x_1] \\ a_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 & [x_1, x_2] \\ \dots & \dots \\ a_{n-1} + b_{n-1}(x - x_{n-1}) + c_{n-1}(x - x_{n-1})^2 + d_{n-1}(x - x_{n-1})^3 & [x_{n-1}, x_n] \end{cases} \tag{7}$$

I vantaggi sono evidenti nella Figura 7 dove notiamo come la funzione spline non soffra del fenomeno di Runge; Inoltre, l'interpolazione risultante è più *liscia* rispetto ai metodi descritti in precedenza.

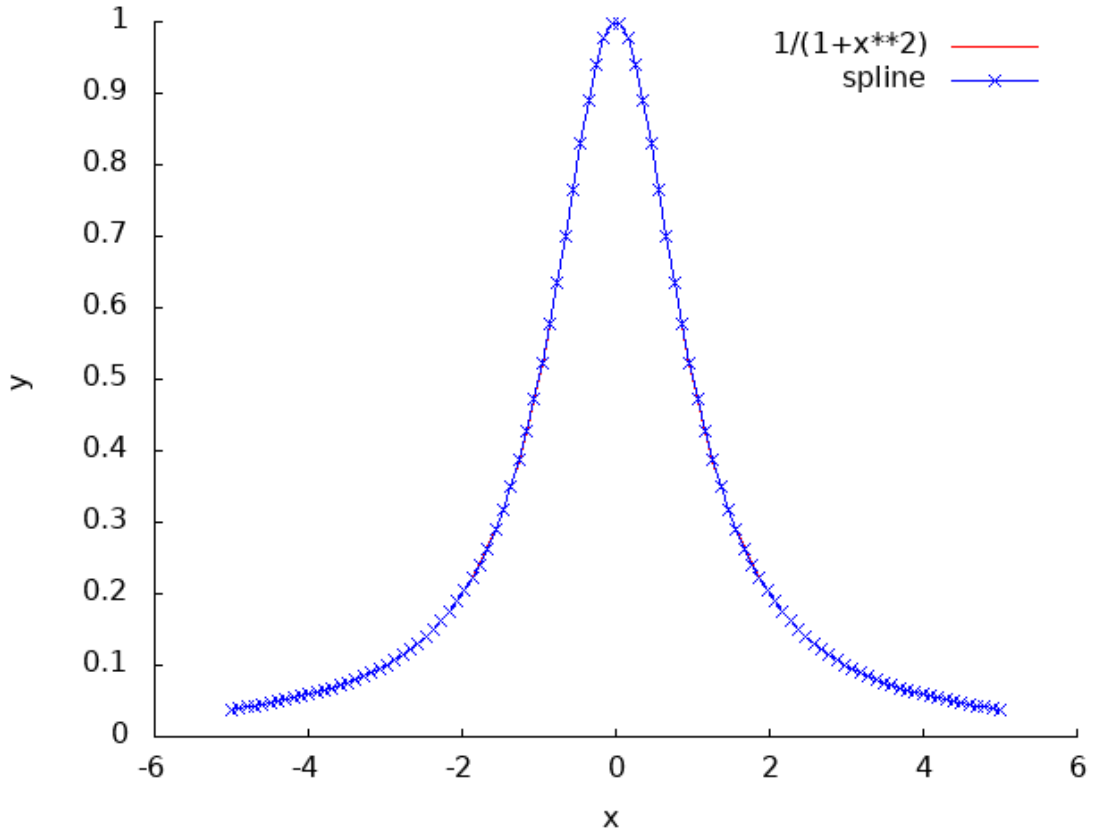

Figura 7: Interpolazione con funzione spline e 15 nodi

4 Implementazione degli algoritmi

4.1 Newton

Piuttosto che risolvere il sistema(4) risulta più pratico utilizzare le differenze divise che possono essere rappresentate come una tabella:

$$\begin{array}{llll} x_0 & f(x_0) & & \\ & & f[x_0,x_1]=\frac{f(x_1)-f(x_0)}{x_1-x_0} & \\ x_1 & f(x_1) & & f[x_0,x_1,x_2]=\frac{f[x_0,x_1]-f[x_1,x_2]}{x_2-x_1} \\ & & f[x_1,x_2]=\frac{f(x_2)-f(x_1)}{x_2-x_1} & \\ x_2 & f(x_2) & & \\ \vdots & \vdots & \vdots & \vdots \\ x_n & f(x_n) & & \end{array}$$

Di conseguenza

$$p(\hat{x}) = y_0 + \sum_{k=1}^{n} F[x_0, ..., x_k](\hat{x} - x_0)...(\hat{x} - x_{k-1}) \tag{8}$$

Il calcolo delle differenze divise richiede al più $O(n^2)$ operazioni aritmetiche, la valutazione di $p(\hat{x})$ con l'algoritmo di Horner ne richiede al più $O(n)$.

4.2 Lagrange

Per l'interpolazione con la forma di Lagrange è consigliabile utilizzare la forma Baricentrica che sotto opportune ipotesi rende la valutazione dei nodi computazionalmente efficiente.

$$p(\hat{x}) = \sum_{j=0}^{n} y_j L_j(\hat{x}) = \sum_{j=0}^{n} y_j \prod_{i=0, i\neq j}^{n} \frac{\hat{x} - x_i}{x_j - x_i} = \prod_{i=0}^{n} (\hat{x} - x_i) \sum_{j=0}^{n} \frac{y_j}{w_j(\hat{x} - x_j)}, \tag{9}$$

dove

$$w_j = \prod_{i=0, i\neq j}^{n} (x_j - x_i) \;\; per \;\; 0 \leq j \leq n$$

In caso i pesi w_i siano già stati precomputati in precedenza, allora la valutazione di $p(\hat{x})$ richiede al più $O(n)$ operazioni aritmetiche.

4.3 Spline

Su ciascun sottointervallo viene costruito un polinomio di interpolazione di terzo grado in modo che la funzione globale $v(x)$ sia data da

$$v(\hat{x}) = s_i(\hat{x}) = a_i + b_i(\hat{x} - x_i) + c_i(\hat{x} - x_i)^2 + d_i(\hat{x} - x_i)^3, \;\; x_i \leq \hat{x} \leq x_{i+1} \tag{10}$$

per $0 \leq i \leq n-1$. Per ogni sottointervallo ci sono quattro incognite da ricavare (a_i, b_i, c_i, d_i). Poiché gli intervalli sono $n-1$, il totale delle incognite da determinare è dato da $4(n-1)$. Per ricavarle, si impongono non solo le condizioni di interpolazione, ma anche la continuità delle derivate prime e seconde nei punti di raccordo tra un sottointervallo e il successivo. Imponendo le condizioni $S_0^{''}(x_0) = S_{n-1}^{''}(x_n) = 0$ (spline naturale) ricaviamo i coefficienti c_i risolvendo il sistema $Ac = b$

$$A = \begin{pmatrix} 1 & 0 & 0 & \cdots & \cdots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & \ddots & & \\ 0 & h_1 & 2(h_1+h_2) & h_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \cdots & \cdots & 0 & 0 & 1 \end{pmatrix} \tag{11}$$

dove $h_i = x_{i+1} - x_i$, mentre

$$c = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix} \quad e \quad b = \begin{pmatrix} 0 \\ \frac{3}{h_1}(y_2 - y_1) - \frac{3}{h_0}(y_1 - y_0) \\ \vdots \\ \frac{3}{h_{n-1}}(y_n - y_{n-1}) - \frac{3}{h_{n-2}}(y_{n-1} - y_{n-2}) \\ 0 \end{pmatrix}$$

Essendo la matrice A a predominanza diagonale possiamo risolvere il sistema $Ac = b$ attraverso fattorizzazione LU e quindi mediante la sequenza di sistemi triangolari

$$\begin{cases} Lz = b \\ Uc = z \end{cases}$$

I restanti coefficienti si calcolano nel seguente modo per $0 \leq i \leq n-1$

$$a_i = y_i, \quad b_i = \frac{1}{h_i}(a_{i+1} - a_i) - \frac{h_i}{3}(2c_i + c_{i+1}), \quad d_i = \frac{c_{i+1} - c_i}{3h_i}$$

4.4 Libreria

Il seguente programma implementa una libreria statica dimostrativa dei principali algoritmi di risoluzione del problema dell'interpolazione polinomiale. I metodi forniti dalla suddetta libreria sono:

- **double *vandermonde(Point *points, int n)** – restituisce i coefficienti del polinomio interpolante, utilizzando come algoritmo la risoluzione della matrice di Vandermonde.
- **double *newton(Point *interp_points, int n, double *eval_points,int n_evalpoints)** – effettua l' interpolazione dei nodi forniti in input utilizzando la forma di Newton. Restituisce un vettore di punti (x,p(x)) valutati attraverso il polinomio ottenuto in precedenza.
- **Point *lagrange(Point *interp_points, int n, double *eval_points,int n_evalpoints)** – effettua l' interpolazione dei nodi forniti in input utilizzando la forma di Lagrange. Restituisce un vettore di punti (x,p(x)) valutati attraverso il polinomio ottenuto in precedenza.

- **Point *spline(Point *interp_points,int n,double *eval_points,int n_evalpoints)** – effettua l' interpolazione dei nodi forniti in input utilizzando la funzione spline cubica. Restituisce un vettore di punti (x,sp(x)) valutati attraverso le funzioni ottenute in precedenza.
- **Point *gen_point(double (*fun)(double),int a,int b, int n)** – restituisce un vettore di n punti equispaziati del tipo (x,f(x)) nell' intervallo [a,b].
- **Point *chebyshev(double(*fun)(double),int a,int b,int n)** – restituisce un vettore contente n nodi di Chebyshev applicati ad una funzione passata come parametro.
- **double *linspace(int a,int b, int n)** – restituisce un vettore contente n nodi equispaziati nell' intervallo [a,b].

Il tipo Point utilizzato nella libreria viene definito nel seguente modo:

```
typedef struct{
    double x;
    double y;
}Point;
```

4.5 Precisione

Nel seguente programma vengono utilizzati numeri in virgola mobile (double) da 64 bit con precisione alla quindicesima cifra decimale.

5 Appendice

5.1 Vandermonde

```
1  #include "stdio.h"
2  #include "stdlib.h"
3  #include "math.h"
4  #include "interpolazione.h"
5
6  double *vandermonde(Point *points, int n)
7  {
8      //output
9      double *coeffpolinomio= malloc(n * sizeof(double));
10     //matrice A - Vandermonde
11     double **a= (double **) malloc(n*sizeof(double *));
12     //1 x0 x0^2 .... x0^n
13     //. .   .   .... .
14     //. .   .   .... .
15     //1 xn xn^2 .... xn^n
16
17     //Allocazione in memoria & prima riga
18     for (int i=0; i < n;i++)
19     {
```

```
20          a[i] = (double *) malloc((n+1)*sizeof(double));
21          a[i][0] = 1;
22      }
23
24      //Costruzione matrice
25      for (int i=0; i < n ;i++)
26      {
27          int j;
28          for (j=1; j < n ;j++)
29          {
30              a[i][j] =pow(points[i].x, (double)j);
31          }
32          a[i][j] = points[i].y;
33      }
34
35
36      for (int i=0;i < n;i++)
37      {
38          //Cerco il massimo in questa colonna
39          double maxE=abs(a[i][i]);
40          int maxRow= i;
41          for (int k=i+1 ; k< n;k++)
42          {
43              if( abs(a[k][i]) > maxE)
44              {
45                  maxE=abs(a[k][i]);
46                  maxRow=k;
47              }
48          }
49
50          //scambio la riga massima con la riga corrente
51          for(int k=i;k <= n;k++)
52          {
53              double tmp = a[maxRow] [k];
54              a[maxRow][k]=a[i][k];
55              a[i][k] =tmp;
56          }
57
58          //mosse di gauss
59          for (int k=i+1; k < n;k++)
60          {
61              double c = - a[k][i] / a[i][i];
62              for (int j=i; j <= n; j++)
63              {
64                  if (i==j)
65                      a[k][j] = 0;
66                  else
67                      a[k][j] += c * a[i][j];
```

```
            }
        }


    }
    //risolvo sistema triangolare
    for(int i = n-1 ; i >= 0; i--)
    {
        coeffpolinomio[i]=a[i][n] / a[i][i];
        for(int k = i - 1;k>=0;k--)
        {
            a[k][n] -= a[k][i] * coeffpolinomio[i];
        }
    }
    return coeffpolinomio;
}
```

5.2 Newton

```
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "interpolazione.h"

double ** divided_difference(Point * interp_points,int n)
{
    double **mdd= (double **) malloc(n*sizeof(double *));

    //set prima colonna f(x0),f(x1).....f(xn-1)
    for (int i=0; i < n;i++)
    {
        mdd[i] = (double *) malloc((n+1)*sizeof(double));
        mdd[i][0]=interp_points[i].y;
    }

    //matrice delle differenze divise
    for(int j=1; j < n;j++)
        for (int k = 1;k <= j; k++)
            mdd[j][k]= ((mdd[j][k-1] -  mdd[j-1][k-1]) / (interp_points[j].x -interp_points[j-k].x));

   return mdd;
}


//Valutazione polinomio in x con metodo di Horner
double eval_point_newton(Point * interp_points,int n,double ** mdd,double x)
{
```

```
    double result = mdd[n-1][n-1];

    for (int i=n-2; i>=0; i--)
        result = (result*(x - interp_points[i].x) + mdd[i][i]);

    return result;
}



Point *newton(Point *interp_points, int n, double *eval_points,int n_evalpoints)
{
    double **mdd = divided_difference(interp_points,n);
    Point *result = malloc(n_evalpoints * sizeof(Point));
    for(int i=0; i < n_evalpoints; i++)
    {
        result[i].x = eval_points[i];
        result[i].y = eval_point_newton(interp_points,n,mdd,eval_points[i]);

    }

    //pulizia
    for (int i=0; i < n;i++)
    {
        free(mdd[i]);
    }
    free(mdd);
    return result;
}
```

5.3 Lagrange

```
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "interpolazione.h"

//Calcolo della componente prod(x_j - x_i)
double *denominatore(Point *points,int n)
{
    double *den= malloc(n * sizeof(double));
    for(int j=0; j < n; j++)
    {
        den[j]=1;
        for(int i=0;i < n;i++)
        {
            if(i != j)
```

```
                den[j]*=(points[j].x-points[i].x);
        }
    }
    return den;
}
//Valutazione polinomio in x
double eval_point_lagrange(Point *interp_points, int n, double *den_coeff, double x)
{
    double prod=1;
    double sum=0;

    //Produttoria
    for(int i=0;i < n ; i++)
        {
            if (x == interp_points[i].x)
                return interp_points[i].y;
            prod *= (x-interp_points[i].x);
        }
    // Sommatoria
    for(int j=0;j < n;j++)
        sum += interp_points[j].y / (den_coeff[j] * (x - interp_points[j].x));

    return sum * prod;
}

Point *lagrange(Point *interp_points, int n, double *eval_points,int n_evalpoints)
{
    double *den = denominatore(interp_points,n);
    Point *result = malloc(n_evalpoints * sizeof(Point));

    for(int i=0; i < n_evalpoints ; i++)
    {
        result[i].x=eval_points[i];
        result[i].y=eval_point_lagrange(interp_points,n,den,eval_points[i]);
    }
    free(den);
    return result;
}
```

5.4 Nodi di chebyshev

```
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "interpolazione.h"

//Generatore di nodi di Chebyshev
```

```c
Point *chebyshev(double (*fun)(double),int a, int b, int n)
{
    Point *points =malloc(n * sizeof(Point));
    double alpha=(b-a)/2;
    double beta = (a+b)/2;

    for(int i=0; i < n;i++)
    {
        points[i].x = (beta-alpha*cos((2*(i)+1)*M_PI/(2*n)));
        points[i].y=fun(points[i].x);
    }
    return points;
}
```

5.5 Punti

```c
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "interpolazione.h"

//Funzione generatrice di punti (x0,fun(x0))...(x_n-1,fun(x_n-1))
Point *gen_point(double (*fun)(double),int a,int b, int n)
{
    Point *points =malloc(n * sizeof(Point));

    for(int i=0;i<n;i++)
     {
      points[i].x= (a + (((double)(i) * (double)(b-a)) / (n-1)));
      points[i].y=fun(points[i].x);
     }

     return points;
}

//Funzione generatrice di n punti compresi nell'intervallo [a,b]
double *linspace(int a,int b, int n)
{
    double *x =malloc(n * sizeof(Point));

    for(int i=0;i<n;i++)
     {
      x[i]= (a + (((double)(i) * (double)(b-a)) / (n-1)));
     }

    return x;
}
```

5.6 Spline

```c
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "interpolazione.h"
//Algoritmo seguito : https://fac.ksu.edu.sa/sites/default/files/numerical_analysis_9th.pdf

//Ricerca binaria modificata per cercare indice
int search_index(Point *lst,double a,double b,double x)
{
    if( b >= a)
    {
        int md = a + (b-a) / 2;

        if(lst[md].x == x)
            return md;

        if (x < lst[md +1].x && x > lst[md].x)
        {
            return md;
        }

        if(x < lst[md].x )
            return search_index(lst,a,md -1,x);

        return search_index(lst,md +1,b,x);
    }
}

Point *spline(Point *interp_points,int n,double *eval_points,int n_evalpoints)
{
    double h[n],l[n],z[n],u[n],A[n],c[n],b[n],d[n];

    for(int i= 0; i < n-1;i++)
         h[i] = interp_points[i + 1].x - interp_points[i].x;

    for(int i= 1;i < n - 1;i++)
        A[i]=(3/h[i]) * (interp_points[i+1].y - interp_points[i].y ) - (3/h[i-1]) * (interp_points[i].y - interp_points[i-1

    //A[0,0]
    l[0]=1;
    u[0]=0;
    //b[0]
    z[0]=0;
```

```
44
45
46      //L non diagonali l[i,i-1] = h[i-1]
47      //l diag l[i,i] = a[i,i] - l[i,i-1] * u[i-1,i]
48      //u[i,i+1] = a[i,i+1] / l[i,i]
49
50      //Lz=b
51      for (int i = 1; i < n - 1; ++i) {
52          l[i] = 2 * (interp_points[i + 1].x - interp_points[i - 1].x) - h[i - 1] * u[i - 1];
53          u[i] = h[i] / l[i];
54          z[i] = (A[i] - h[i - 1] * z[i - 1]) / l[i];
55      }
56
57      z[n-1]=0;
58      c[n-1]=0;
59
60      // Uc = z e calcolo coeff b,d
61      for (int j = n - 2; j >= 0; --j) {
62          c[j] = z[j] - u[j] * c[j + 1];
63          b[j] = ((interp_points[j + 1].y - interp_points[j].y) / h[j]) - (h[j] * (c[j + 1] + 2 * c[j]) / 3);
64          d[j] = (c[j + 1] - c[j]) / (3 * h[j]);
65      }
66
67      //Valutazione punti
68      Point *result = malloc(n_evalpoints * sizeof(Point));
69      for(int i=0; i < n_evalpoints; i++)
70      {
71          int k = search_index(interp_points,0,n-1,eval_points[i]);
72          result[i].x=eval_points[i];
73          result[i].y =(interp_points[k].y + b[k]*(eval_points[i] - interp_points[k].x) + c[k] * pow((eval_points[i]-interp_p
74                  + d[k] * pow((eval_points[i]-interp_points[k].x),3));
75
76      }
77      return result;
78  }
```

5.7 test

```
1   void plot(char *name,Point *points,int n)
2   {
3       mkdir("plot", 0777);
4       FILE *f = popen("gnuplot", "w");
5
6       fprintf(f,
7       "set term png; "
8       "set output 'plot/%s.png'\n"
9       "set style line 1 lt 1 lw 6 \n"
```

```
10      "set xlabel 'x' \n"
11      "set ylabel 'y'\n"
12      "set border 3 \n"
13      "set xtics nomirror \n"
14      "set ytics nomirror \n"
15      "plot 1/(1+x**2) lt rgb 'red'  , '-' lt rgb 'blue'  title '%s' w linespoint \n",name,name);
16
17      for (int i=0; i< n; i++)
18        fprintf(f, "%g %g\n", points[i].x, points[i].y);
19
20      fclose(f);
21  }
22
23  void test(int a,int b,int n,int nxi)
24  {
25      Point *equi_points =gen_point(rounge,a,b,n);
26      Point *chb_points =chebyshev(rounge,a,b,n);
27      double *int_vande = vandermonde(equi_points,n);
28
29      fprintf(stdout,"Coefficienti ottenuti:\n");
30      for(int i=0;i<n;i++)
31      {
32          if(int_vande[i] >= 0 )
33              printf("+%.10fx^%d ",int_vande[i],i);
34          else
35              printf("%.10fx^%d ",int_vande[i],i);
36      }
37      fprintf(stdout,"\n");
38      double *xi = linspace(-5,5,nxi);
39      Point *int_newton = newton(equi_points,n,xi,nxi);
40      Point *int_lag = lagrange(equi_points,n,xi,nxi);
41      Point *int_lag_cheb = lagrange(chb_points,n,xi,nxi);
42      Point *int_spline = spline(equi_points,n,xi,nxi);
43      fprintf(stdout,"Plot delle funzioni in formato .png\n");
44
45      char newton_string[50];
46      sprintf(newton_string,"Newton-%d nodi",n);
47      char lagrange_string[50];
48      sprintf(lagrange_string,"Lagrange-%d nodi",n);
49      char lagrange_cheb_string[50];
50      sprintf(lagrange_cheb_string,"Lagrange-Cheb-%d nodi",n);
51      char spline_string[50];
52      sprintf(spline_string,"Spline-%d nodi",n);
53
54      plot(newton_string,int_newton,nxi);
55      plot(lagrange_string,int_lag,nxi);
56      plot(lagrange_cheb_string,int_lag_cheb,nxi);
57      plot(spline_string,int_spline,nxi);
```

```
58
59      //pulizia
60      free(equi_points);
61      free(chb_points);
62      free(int_vande);
63      free(xi);
64      free(int_newton);
65      free(int_lag);
66      free(int_lag_cheb);
67      free(int_spline);
68  }
```

Riferimenti bibliografici

[1] R. Bevilacqua and O. Menchi. *Appunti di calcolo numerico.*

[2] R. L. Burden. *Numerical Analysis.* 2010.

[3] L. Gemignani. *Lezioni di calcolo numerico.*

[4] D. Levy. *Interpolation.*

[5] M. Ragusa. *Interpolazione di funzioni.* 2007.

[6] Wikipedia. Polinomio di Čebyšëv.

[7] Wikipedia. Polynomial interpolation.