Progetto sviluppato: **Classe di Quaternioni**

Lorenzo Bellettini

Introduzione

Lo scopo di tale progetto è quello di descrivere degli oggetti matematici detti “quaternioni” attraverso l'utilizzo della sintassi c++.
In particolare è stata creata e implementata una classe che crei degli oggetti di tipo “Quaternione” e che li manipoli tramite varie operazioni.
Le operazioni più utili e significative che si possono svolgere con i quaternioni sono sicuramente le rotazioni spaziali e il prodotto di Hamilton, per cui sono stati rispettivamente ridefiniti gli operatori binari "&&" (operatore booleano and) e "||" (operatore booleano or) all'interno della classe.

Ho scelto questo tema poiché ho davvero apprezzato la dinamicità del linguaggio c++ nel suo aspetto object oriented, e ammetto che sviluppare tale progetto mi ha dato parecchie soddisfazioni.
Ho inoltre deciso di sviluppare il codice in modo modulare, ovvero di separare la dichiarazione della classe e l'implementazione dei suoi metodi in due file distinti, al fine di rendere il codice più preciso e facilmente consultabile o riutilizzabile in futuro.
La relazione è strutturata in vari paragrafi, uno per ciascuna proprietà dei quaternioni. In tali paragrafi verranno trattate le proprietà matematiche di tali oggetti, quindi, evitando eccessiva pedanteria, illustrate le maniere con cui tali proprietà sono state tradotte nel linguaggio c++.

I Quaternioni: struttura generale

I quaternioni sono una generalizzazione a quattro dimensioni dei numeri complessi, introdotti dal matematico W. R. Hamilton a metà 'Ottocento.
Proprio come i numeri complessi descrivono un piano, i quaternioni descrivono uno spazio vettoriale quadridimensionale.
Un quaternione è formato, in ulteriore analogia con i complessi, da una parte reale e da una parte immaginaria.

Possiamo rappresentare un quaternione con questa struttura:

ai + bj + ck + d.

Dove a, b, c sono i coefficienti della parte immaginaria, (caratterizzata dalle unità immaginarie i, j, k) mentre d è il coefficiente della parte reale.
Possiamo quindi ridurre la rappresentazione di un quaternione a :

v = parte immaginaria (v *C* R^3) che possiamo anche chiamare “parte vettoriale”
r = parte reale (r *C* R)

quindi q = (**v**, r)

Per rappresentare in modo efficiente questi quattro elementi ho posto nella classe Quaternione un array di double come attributo privato.
Tale array contiene i valori numerici dei quattro coefficienti del quaternione; esso viene inizializzato ad array nullo quando un Quaternione viene creato tramite costruttore di default (da me ridefinito) mentre viene inizializzato con dei valori specifici utilizzando un costruttore parametrico.
Inoltre l'attributo, non essendo stato dichiarato const, rimane sempre accessibile ai metodi della classe, perciò i suoi valori possono essere modificati anche dopo la creazione del Quaternione tramite dei metodi “Setter”.
La scelta di utilizzare un array è avvenuta solo dopo aver già implementato vari metodi della classe. Mi sono infatti accorta che un array di double sarebbe potuto essere molto più facilmente manipolabile rispetto a quattro singoli attributi, non solo per le operazioni di input-output ma anche per altri metodi in cui dei cicli “for” avrebbero reso il codice più snello.

Le operazioni più significative: alcuni operatori in ridefinizione

Prodotto:

Essendo la matrice identità:

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

I coefficienti immaginari i, j, k sono identificati con le seguenti matrici:

$$i = \begin{bmatrix} i & 0 \\ 0 & -i \end{bmatrix}$$

$$j = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

$$k = \begin{bmatrix} 0 & i \\ i & 0 \end{bmatrix}$$

Di conseguenza, svolgendo il prodotto righe per colonne, avremo:

$$ii = i^2 = -1$$
$$jj = j^2 = -1$$
$$kk = k^2 = -1$$

$$ij = -ji = k$$
$$jk = -kj = i$$
$$ki = -ik = j$$

Queste proprietà mostrano come il prodotto di due quaternioni sia non commutativo; i quaternioni risultano perciò formare un gruppo non abeliano rispetto al prodotto.

Il prodotto fra due quaternioni è stato da me implementato tramite ridefinizione dell'operatore "*", in modo da poter svolgere tale operazione in modo agile.
Ho quindi posto un overload di tale operatore per poter descrivere anche il prodotto di un quaternione per un numero reale. La potenza del linguaggio c++ permette al programma che utilizza i metodi e gli oggetti della mia classe di riconoscere quando utilizzare un metodo piuttosto che l'altro in base agli argomenti forniti:

Quaternione * Quaternione	nel primo caso
Quaternione * double	nel secondo caso.

(Gran linguaggio!)

Quoziente:

Dato che il prodotto fra due quaternioni non è commutativo, l'operatore "Quoziente" si declina in due definizioni: quoziente destro e quoziente sinistro.

Il quoziente destro di un quaternione p per un quaternione h risulta essere il quaternione q così definito:

$$q = p * *h/||h||^2$$

Il quoziente sinistro di un quaternione p per un quaternione h risulta essere il quaternione q così definito:

$$q = *h/||h||^2 * p$$

per descrivere tali operazioni ho compiuto una ridefinizione di due operatori binari:

"/" : quoziente destro
"%" : quoziente sinistro

Vorrei far notare che l'espressione:

*h/||h||^2

indica il quaternione inverso di h*;* è stato quindi utile scrivere un metodo per tale operazione (ridefinendo l'operatore ~ -tilde), per poi chiamarlo all'interno dei metodi per il quoziente, al fine di snellire il programma.

Coniugato:

Ogni quaternione, proprio come ogni numero complesso, ha il proprio coniugato.
Nel passaggio da quaternione a suo coniugato, si conserva la parte reale, mentre la parte immaginaria cambia di segno.
Volendo indicare il coniugato con un asterisco avremo:

$$q = ai + bj + ck + d$$
$$q^* = - ai - bj - ck + d$$

Anche per ottenere il quaternione coniugato ho ritenuto opportuno ridefinire un operatore.
Ho ridefinito quindi l'operatore unario "!" (not - negazione booleana).

Prodotto di Hamilton:

Ogni quaternione la cui parte reale sia nulla è detto quaternione immaginario puro.
All'interno della classe ho implementato un costruttore apposito che richiede solamente tre argomenti di tipo double, che andranno a riempire i primi tre spazi dell'array attributo.
Il quarto spazio verrà invece inizializzato a zero dal costruttore stesso.

Il prodotto di Hamilton è appunto un prodotto fra due quaternioni immaginari puri.
Tale prodotto dà come risultato un terzo quaternione (NON immaginario puro).
Per implementare il metodo (con l'operatore binario “&&” in ridefinizione) non ho perciò dovuto creare un'ulteriore operazione, poiché ho semplicemente sfruttato l'operatore "*" già implementato.
Ho dovuto tuttavia creare un messaggio di errore nel caso in cui i quaternioni forniti non fossero immaginari puri.
Inizialmente sono stata tentata di creare un metodo “Quaternione Amputa()” per eliminare la parte reale di un quaternione già creato, qualora l'utente volesse utilizzarlo come operando del prodotto di Hamilton, tuttavia, dando ascolto ad una severa reprimenda del professor Servizi, mi sono limitata a rigettare i quaternioni non privi di parte reale che pretendessero di essere operandi di tale operazione.

Reputo degna di nota la seguente proprietà:

Considerando i quaternioni immaginari puri come vettori in R^3, notiamo che il quaternione risultante dal prodotto di Hamilton fra i due presenta come parte reale il loro prodotto scalare, e come parte immaginaria il loro prodotto vettoriale.

Provare per credere!

Rotazioni:

Tramite i quaternioni è possibile definire rotazioni di vettori nello spazio.
In particolare, ad ogni quaternione unitario è associata una matrice di rotazione 3x3.
Sfruttando quindi la corrispondenza dei quaternioni immaginari puri con dei vettori in R^3, possiamo ottenere un quaternione la cui parte vettoriale è il vettore di partenza ruotato grazie ad un quaternione unitario u.

In altre parole, avendo un quaternione unitario u, che corrisponde alla nostra rotazione, e un quaternione immaginario puro x, possiamo definire il prodotto:

ux*u = y

dove *u è il coniugato di u, e dove y è il quaternione la cui parte vettoriale è il vettore x (o quaternione immaginario puro) ruotato rispetto al quaternione unitario u.

Ho implementato perciò il metodo "Quaternione Unitario()" che ha il compito di dividere ogni coefficiente del quaternione chiamante per la sua norma.
Il valore della norma di un quaternione è calcolata tramite la funzione "double Norma()".

INPUT-OUTPUT

Ho ritenuto opportuno ridefinire anche le operazioni di INPUT e OUTPUT per i quaternioni.
Gli operatori binari >> (estrazione) e << (inserimento) sono stati dichiarati “friend” in quanto essi non appartengono alla classe Quaternione. Con questo espediente ho fatto sì che essi potessero accedere a membri privati della classe.

Essi ricevono operatori di stream, in particolare:

istream cin;
ostream cout;

Affinché la ridefinizione avesse successo ho dovuto perciò includere l'header <iostream> nei file Quaternione.h e Quaternione.cxx.

La ridefinizione dell'operatore di estrazione più complessa di quella dell'operatore di inserimento, in quanto ho reputato necessario l'inserimento di un sistema di controllo.
Una volta che il fruitore ha inserito i coefficienti del quaternione, lo stesso operatore di estrazione, prima di assegnare tali coefficienti al nuovo quaternione, controlla se l'INPUT è andato a buon fine.
A questo scopo ho utilizzato la funzione bool istream::fail() definita nell'header <iostream>: quando INPUT è accettabile (es: non è un carattere, non è un simbolo ecc) allora la funzione prosegue nel chiedere in INPUT i coefficienti, in caso contrario la funzione termina.

Appendice

Di seguito è riportata una legenda degli operatori ridefiniti e il codice che descrive la classe. Riporto infine il codice del programma di test che serve appunto a verificare il funzionamento delle operazioni più rilevanti per gli oggetti della classe Quaternione.
In tale programma ho scelto di temporizzare l'output, al fine di rendere più leggibili le informazioni stampate a schermo.

Legenda per l'utilizzo della classe – operatori ridefiniti:

/ : Quoziente destro
% : Quoziente sinistro
&& : Rotazioni
|| : Prodotto di Hamilton
~ : Quaternione inverso
! : Coniugato del quaternione fornito

Codice:

_____Quaternione.h_____

```
#ifndef QUATERNIONE_H
#define QUATERNIONE_H
#include <iostream>

class Quaternione {

 public:

  //Costruttori
  Quaternione(); //default
  Quaternione(double, double, double, double); //parametrico
  Quaternione(const Quaternione&); //costruttore di copia
  Quaternione(double, double, double); //quaternione immaginario puro

  //Getters
  double Geta() const;
  double Getb() const;
  double Getc() const;
  double Getd() const;
```

```
//Setters
void Seta(double);
void Setb(double);
void Setc(double);
void Setd(double);


//ridefinizione di operatori

//binari

Quaternione operator+(Quaternione&);
Quaternione operator-(Quaternione&);
Quaternione operator*(Quaternione&); //non simmetrico
Quaternione operator*(double); //simmetrico
Quaternione operator/(Quaternione &); //quoziente destro
Quaternione operator%(Quaternione &); //quoziente sinistro

Quaternione operator+=(Quaternione&);
Quaternione operator-=(Quaternione&);
Quaternione operator*=(Quaternione&);

Quaternione operator&&(Quaternione&);//Rotazione
Quaternione operator||(Quaternione&);//Prodotto di Hamilton


//unari

Quaternione operator!(); //Coniugato
Quaternione operator~(); //Inverso


//friends

//input-output
friend std::ostream& operator<<(std::ostream&, Quaternione);
friend std::istream& operator>>(std::istream&, Quaternione&);

//confronto
friend bool operator == (Quaternione, Quaternione);
friend bool operator != (Quaternione, Quaternione);


//Altro
Quaternione Unitario(); //Unitario
double Norma(); //Norma del quaternione


private:

double C_[4];//coefficienti del quaternione
```

```
 };

#endif
```

_____Quaternione.cxx_____

```
#include "Quaternione.h"
#include <iostream>
#include <cmath>
#include <ctype.h>

using namespace std;

//Constructors


//costruttore di default
Quaternione::Quaternione() {
 for (int i=0; i<4; i++) C_[i] = 0;
}
//costruttore parametrico
Quaternione::Quaternione(double a, double b, double c, double d) {
 C_[0] = a;
 C_[1] = b;
 C_[2] = c;
 C_[3] = d;
}


//costruttore di copia
Quaternione::Quaternione(const Quaternione& n) {
 for (int i=0; i<4; i++) C_[i] = n.C_[i];
}


//immaginario puro
Quaternione::Quaternione(double a, double b, double c) {
 C_[0] = a;
 C_[1] = b;
 C_[2] = c;
 C_[3] = 0;
}



//Getters


double Quaternione::Geta() const {
 return C_[0];
}

double Quaternione::Getb() const {
```

```
 return C_[1];
}

double Quaternione::Getc() const {
 return C_[2];
}

 double Quaternione::Getd() const {
 return C_[3];
}



//Setters

void Quaternione::Seta(double a) {
 C_[0] = a;
}

void Quaternione::Setb(double b) {
 C_[1] = b;
}

void Quaternione::Setc(double c) {
 C_[2] = c;
}

void Quaternione::Setd(double d) {
 C_[3] = d;
}



//Ridefinizione di operatori

Quaternione Quaternione::operator+ (Quaternione& n) {
 Quaternione somma;

 for (int i=0; i<4; i++) somma.C_[i] = C_[i] + n.C_[i];

 return somma;
}

Quaternione Quaternione::operator- (Quaternione& n) {
 Quaternione differenza;

 for (int i = 0; i<4; i++)
 differenza.C_[i] = C_[i] - n.C_[i];

 return differenza;
}
```

```
Quaternione Quaternione::operator* (Quaternione& n) {

 Quaternione prodotto;

 prodotto.C_[0] = C_[0]*n.C_[3] + C_[1]*n.C_[2] - C_[2]*n.C_[1] + C_[3]*n.C_[0];
 prodotto.C_[1] = -C_[0]*n.C_[2] + C_[1]*n.C_[3] + C_[2]*n.C_[0] +
C_[3]*n.C_[1];
 prodotto.C_[2]= C_[0]*n.C_[1] - C_[1]*n.C_[0] + C_[2]*n.C_[3] + C_[3]*n.C_[2];
 prodotto.C_[3] = -C_[0]*n.C_[0] -C_[1]*n.C_[1] -C_[2]*n.C_[2] +C_[3]*n.C_[3];

 return prodotto;
}


Quaternione Quaternione::operator *(double x) {
 Quaternione p(*this);
 for (int i=0;  i<4; i++) p.C_[i] *= x;
 return p;
}


//quoziente destro
Quaternione Quaternione::operator /(Quaternione & h) {

 Quaternione s = ~h;

 Quaternione quoziente = *this * s;
 return quoziente;
}
//quoziente sinistro
Quaternione Quaternione::operator %(Quaternione & h) {

 Quaternione s = ~h;

 Quaternione quoziente = s * *this;
 return quoziente;
}



//assegnamento composto a somma
Quaternione Quaternione::operator+=(Quaternione &r) {
 *this = (*this) + r;
 return *this;
}


//assegnamento composto a differenza
Quaternione Quaternione::operator-=(Quaternione &r) {
 *this = (*this) - r;
 return *this;
}
```

```
//assegnamento composto a prodotto
Quaternione Quaternione::operator*=(Quaternione& r) {
 *this = (*this) * r;
 return *this;
}


//Rotazione spaziale
Quaternione Quaternione::operator&&(Quaternione &p) {

 Quaternione ruotato = Quaternione(*this);
 if(Getd() != 0) ruotato.Setd(0);
 Quaternione u;
 u = p.Unitario();
 Quaternione pcon;
 pcon = !ruotato;

 Quaternione l;
 l = ruotato * u;

 Quaternione r = l * pcon;

 return r;
}


//Prodotto di Hamilton : crea corrispondenza fra quat. immaginari puri e vettori in R3.
Quaternione Quaternione::operator||(Quaternione &r) {

 Quaternione prodotto;

 //check:
 if (Getd() == 0 && r.Getd() == 0) {

 //parte immaginaria: prodotto vettoriale
 //parte reale: prodotto scalare

 prodotto = *this * r;

 return prodotto; //attenzione! il prodotto fra due q. immaginari puri non è un immaginario puro!
 }


  else {
   cout << "uno dei quaternioni forniti non è immaginario puro." << '\n'
    << "Non è possibile creare una corrispondenza con vettori in R3" << endl;
 }
}
```

```
//Coniugato
Quaternione Quaternione::operator!() {

 Quaternione coniugato;
 for (int i=0; i<3; i++) coniugato.C_[i] = -C_[i];
 coniugato.C_[3] = C_[3];
 return coniugato;
}


//Inverso del quaternione dato
Quaternione Quaternione::operator~() {
 Quaternione q;
 Quaternione c = !*this;
 for (int i=0; i<4; i++) q.C_[i] = c.C_[i]/pow(c.Norma(), 2);

 return q;
}




//friends

//input-output
ostream& operator<< (ostream& c, Quaternione r) {

 c << endl;
 if (r.Geta()==0 && r.Getb()==0 && r.Getc()==0 && r.Getd()==0) cout <<
"Quaternione nullo" << endl;
 else if (r.Getd()==0) {
  cout  << "(immaginario puro)" << '\n'
   << r.C_[0] << "i + " << r.C_[1] << "j + " << r.C_[2] << "k " << endl;
 }
 else cout << r.C_[0] << "i + " << r.C_[1] << "j + " << r.C_[2] << "k + " <<
r.C_[3] << endl;

 return c;

}


istream& operator >>(istream &g, Quaternione &q) {

 double digit;
 for (int i=0; i<4; i++) {
  cout << "Elemento " << i+1 << ": " ;
  cin >> digit;
  if(cin.good()) q.C_[i] = digit;
  else {
```

```
   cerr << "Solo valori numerici, per favore." << endl;
   break;
  }
 }
 return g;
}



//Confronto
bool operator == (Quaternione r, Quaternione q) {

 Quaternione s = r - q;
 if(s.Geta()==0 && s.Getb()==0 && s.Getc()==0 && s.Getd()==0)
   return true;
 else return false;

}



bool operator != (Quaternione r, Quaternione q) {

 Quaternione s = r - q;
 if(s.Geta()!=0 || s.Getb()!=0 || s.Getc()!=0 || s.Getd()!=0)
   return true;
 else return false;

}


//Altro

//Norma del quaternione
double Quaternione::Norma() {

 double tot;
 for (int i=0; i<4; i++) tot += pow(C_[i], 2);
 return sqrt(tot);

}


//Trasforma il quaternione dato nell'unitario corrispondente
Quaternione Quaternione::Unitario() {

 Quaternione u;
 double norma = Norma();
 for (int i =0; i<4; i++) u.C_[i] = C_[i]/norma;
 return u;

}
```

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include "Quaternione.h"

using namespace std;
int main () {

 time_t t;
 time_t a;


 cout << '\n' << "Inserire i coefficienti del quaternione." << '\n';

 time(&a);
 while (time(&t) < a + 0.5);
 cout << " NB:" << '\n' << " I primi tre numeri inseriti saranno i coefficienti
della parte immaginaria del quaternione, " << endl;

 time(&a);
 while (time(&t) < a + 1);
 cout << " mentre il quarto sarà il coefficiente della sua parte reale." << endl;


 Quaternione q;
 cin >> q;
 if (cin.fail()) return 0;


 cout << '\n' << "Questo è il quaternione inserito" << q << '\n';


 time(&a);
 while (time(&t) < a + 3);
 cout << "Questo è il suo corrispondente coniugato: " << !q << '\n';
 time(&a);
  while (time(&t) < a + 3);
  cout << "questo è il suo corrispondente unitario: " << q.Unitario() << '\n';

 time (&a);
  while (time(&t) < a + 3);
   cout  << "Questa è la sua norma: " << '\n' << q.Norma() << endl;

 time (&a);
 while (time(&t) < a + 3);
 cout << '\n' << "Inserire i coefficienti di un secondo quaternione." << endl;
 Quaternione r;
 cin >> r;
```

```
 if (cin.fail()) return 0;
  cout << '\n' << "Questo è il quaternione inserito" << r << endl;

 time (&a);
 while (time(&t) < a + 3);

 if(q == r) cout << "Hai inserito due quaternioni uguali fra loro!" << endl;
 else cout << "I quaternioni inseriti sono diversi fra loro." << endl;

 time (&a);
 while (time(&t) < a + 3);
 cout << "Ora svolgerò varie operazioni fra i quaternioni inseriti." << '\n' <<
endl;

 time (&a);
 while (time(&t) < a + 3);
 cout  << "Somma: " << q+r << '\n';

 time (&a);
 while (time(&t) < a + 3);
 cout << "Differenza:" << (q-r) << '\n';

 time (&a);
 while (time(&t) < a + 3);
 cout << "Prodotto del primo per il secondo:" << (q*r) << '\n';

 time (&a);
 while (time(&t) < a + 4);
 cout << "Prodotto del secondo per il primo:" << (r*q) << '\n';
 cout << endl;

 time (&a);
 while (time(&t) < a + 4);
 cout << "Quoziente destro del primo rispetto al secondo:" << (q/r) << '\n';

 time (&a);
 while (time(&t) < a + 4);
 cout << "Quoziente sinistro del primo rispetto al secondo:" << (q%r) << '\n';
 cout << endl;

 time (&a);
 while (time(&t) < a + 3);
 cout  << "Rotazione spaziale, " << '\n';
 time (&a);
 while (time(&t) < a + 2);
     cout << " ponendo come vettore da ruotare la parte immaginaria del primo
quaternione inserito" << endl;
 cout << " e come quaternione unitario di rotazione il secondo quaternione
inserito:" << '\n';

 time (&a);
 while (time(&t) < a + 3);
```

```
 cout << (q&&r) << '\n';

 time (&a);
 while (time(&t) < a + 2);
 cout << "Ora svolgerò l'operazione opposta," << '\n';

 time (&a);
 while (time(&t) < a + 2);
 cout << " ovvero ruoterò il secondo quaternione in base alla rotazione
imposta dal primo quaternione reso unitario:" << '\n';

 time (&a);
 while (time(&t) < a + 3);
 cout << (r&&q) << '\n' << endl;

 time (&a);
 while (time(&t) < a + 3);
  cout << "Il Prodotto di Hamilton si può svolgere solo fra due quaternioni
immaginari puri. " << '\n' << endl;

 //controlliamo:
 double dq = q.Getd();
 double dr = r.Getd();

 time (&a);
 while (time(&t) < a + 4);
 if (dq == 0) cout << "Il primo quaternione inserito è già immaginario puro."
<< endl;
 else {
  cout << "Il primo quaternione non è immaginario puro." << endl;
  time(&a);
  while (time(&t) < a+2);
  cout << "Perciò creerò io per te un quaternione immaginario puro... Non
ringraziarmi." << endl;
  srand(time(NULL));

  q.Seta(rand()%10);
  q.Seta(rand()%10);
  q.Seta(rand()%10);
  q.Setd(0);

  time(&a);
  while (time(&t) < a+4);
  cout <<'\n' <<  "Ho creato questo quaternione: " << '\n' << q << endl;
 }


 time(&a);
 while (time(&t) < a+4);
 if (dr == 0) cout << "Il secondo quaternione inserito è già immaginario puro."
<< '\n' << endl;
 else {
```

```
  cout << "Il secondo quaternione non è immaginario puro." << endl;
  time(&a);
  while (time(&t) < a+4);
  cout << "Perciò creerò io per te un quaternione immaginario puro... Non
ringraziarmi." << endl;
  srand(time(NULL));

  r.Seta(rand()%10);
  r.Setb(rand()%10);
  r.Setc(rand()%10);
  r.Setd(0);

  time(&a);
  while (time(&t) < a+4);
  cout << '\n' <<  "Ho creato questo quaternione: " << '\n' << r << endl;

 }


 time (&a);
 while (time(&t) < a + 4);
 cout << "Questo è il prodotto di Hamilton fra i due quaternioni:" << (q||r) <<
'\n';

 time (&a);
 while (time(&t) < a + 3);
 cout  << "Viceversa:" << (r||q) << endl;

 time (&a);
 while (time(&t) < a + 3);
 cout << "Arrivederci!" << '\n' << endl;

 return 0;
}
```