Elementi di Programmazione
Zero Robotics

Prof. Eliana Bottari, prof. Giovanni Rizzo
IIS "Verona Trento - Marconi - Majorana"
Messina

Definizione di Algoritmo

- Procedimento di risoluzione di un problema;
- Permette di ottenere un risultato eseguendo una sequenza finita di operazioni elementari;
- Esempi:
  1. Una ricetta di cucina.
  2. Il metodo per calcolare le radici di un'equazione di secondo grado.
  3. Le istruzioni per utilizzare il microonde.
- Ci possono essere più algoritmi in grado di risolvere lo stesso problema.

Algoritmo

Proprietà fondamentali dell'algoritmo:

- la sequenza di istruzioni deve essere finita (*finitezza*);
- la sequenza di istruzioni deve portare ad un risultato (*efficacia*);
- le istruzioni devono essere eseguibili materialmente (*realizzabilità*);
- e istruzioni devono essere espresse in modo non ambiguo (*non ambiguità*).

È importante valutare le risorse utilizzate (tempo, memoria, ...) perché un consumo eccessivo delle stesse può pregiudicare la possibilità stessa di utilizzo di un algoritmo.

Programmare

- Se esiste un procedimento che:
  1. può essere descritto in modo non ambiguo fino ai dettagli;
  2. conduce sempre all'obiettivo desiderato in un tempo finito;

  allora esistono le condizioni per affidare questo compito a un calcolatore;
- Si descrive l'algoritmo in questione in un programma scritto in un opportuno linguaggio comprensibile alla macchina.

Programma

- Il calcolatore sa eseguire molte operazioni di base: somma, sottrazione, AND, ecc.;
- Per risolvere un determinato problema si combinano queste operazioni in modo opportuno;
- Programmare significa determinare quali operazioni eseguire e in quale sequenza per raggiungere la soluzione.

Linguaggi

- Perché il programmatore possa istruire il calcolatore sulle operazioni da fare, serve un linguaggio noto ad entrambi
- Problema:
  1. Il calcolatore comprende solo sequenze di zeri e uno (ad es. la sequenza 1001001 potrebbe significare, per un ipotetico calcolatore, *fai la somma*): *linguaggio macchina.*
  2. Il programmatore comprende le parole *fai la somma*(mentre 1001001 non significa nulla per lui): *linguaggio umano.*

Soluzione 1. Il programmatore impara il *linguaggio macchina*, ma questo :

- ha un basso livello di astrazione (scende molto nei dettagli realizzativi e perde la visione di insieme del problema da risolvere);
- è difficile da ricordare (le istruzioni possono essere diverse centinaia);
- è diverso per ogni piattaforma hardware (ogni tipo di microprocessore ha il suo set di istruzioni).

Soluzione 2. Il programmatore impara l' *assembly*, simile al linguaggio macchina :

- ha un basso livello di astrazione;
- è più facile da ricordare (ad es. per avere una somma invece di scrivere 1001001 si scrive ADD);
- viene tradotto in linguaggio macchina da un programma relativamente semplice (assembler) che, in linea di massima, sostituisce le istruzioni assembly con le corrispondenti istruzioni macchina;
- è diverso per ogni piattaforma hardware (sebbene possano essere simili).

Linguaggi di Programmazione 3

Soluzione 3. Il programmatore impara *un linguaggio di programmazione ad alto livello (HLL)*, simile al linguaggio macchina :

- un traduttore complesso ed efficiente lo traduce in linguaggio macchina o in assembly;
- ha un alto livello di astrazione (esprime le operazioni da svolgere senza entrare nei dettagli, es. A+B calcola la somma di due valori);
- è più simile al linguaggio umano e quindi più facile da ricordare (es. "print X" potrebbe essere l'istruzione per visualizzare il valore di X);
- è (quasi) indipendente da piattaforma hardware e sistema operativo (PC, Mac, Windows, Linux, ecc.), è il traduttore che lo converte per il sistema in uso.

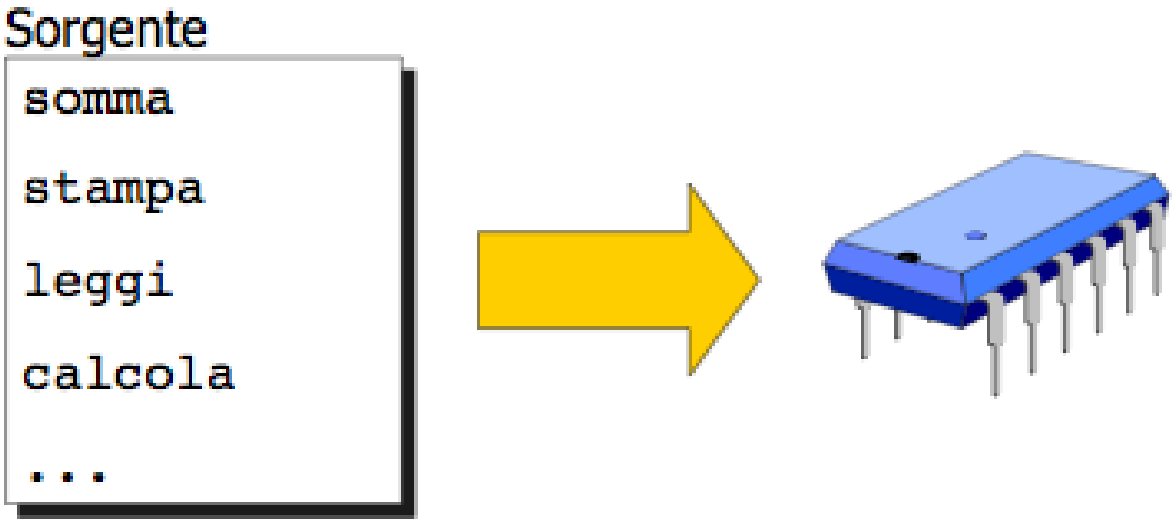
Programma Sorgente

I programmatore sviluppa un programma scrivendo in un linguaggio di programmazione (di alto o basso livello) le operazioni da far eseguire al calcolatore e le memorizza in un file detto:

- Programma.
- Programma sorgente.
- Codice sorgente.
- Sorgente.

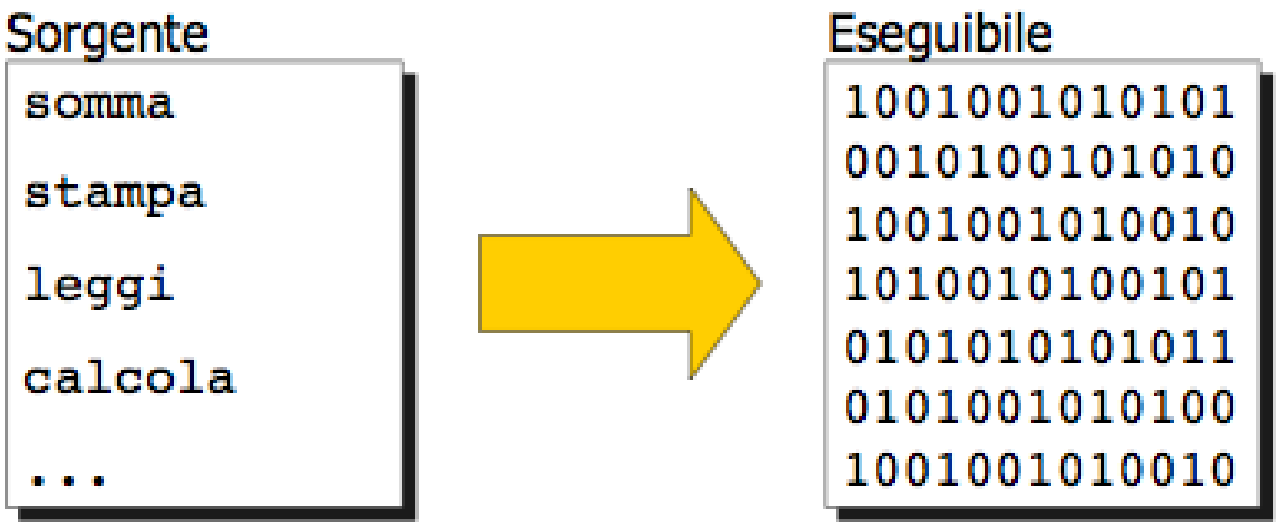
Traduttore di Tipo Interprete

Le istruzioni del codice sorgente vengono ad una ad una tradotte in linguaggio macchina e subito eseguite dalla CPU



Traduttore di Tipo Compilatore

Tutto il codice sorgente viene tradotto in linguaggio macchina e memorizzato in un file detto *programma*(o file o codice) eseguibile.



Differenze

Velocità di esecuzione.

- Ogni volta che l'interprete esegue un programma, deve attuare la traduzione delle istruzioni in linguaggio macchina: lento.
- Il programma compilato è già tradotto e ha quindi una velocità di esecuzione molto superiore.
- Il compilatore è in genere in grado di produrre una traduzione più efficiente dal punto di vista della velocità di esecuzione del codice prodotto (oppure della dimensione del codice).

Librerie

- In un HLL il programmatore non ha necessità di programmare le operazioni di base (ad es. leggere un numero dalla tastiera, calcolare la radice quadrata, visualizzare una parola, ecc.).
- Queste operazioni sono state programmate e compilate dal produttore del traduttore e sono a disposizione del programmatore sotto forma di funzioni.
- I codici eseguibili (quindi già tradotti in linguaggio macchina) che realizzano queste operazioni vengono raggruppati in file detti librerie (collezioni di funzioni di base).
- Esistono vari tipi di librerie: matematiche, grafiche ....

Creazione di un Eseguibile

Il processo di creazione di un eseguibile a partire dai sorgenti (*build*) è composto da 2 fasi:

- *Compilazione* - Il sorgente viene compilato, ma alcune parti (le *operazioni di base*) sono ancora mancanti; viene generato un file intermedio detto **file oggetto**.
- *Linking* - il file oggetto e le librerie vengono unite (collegate – link) così da aggiungere al file oggetto le parti mancanti e costituire un unico file eseguibile. La fase di link può creare un eseguibile collegando più file oggetto e più librerie.

Il Linguaggio C:Caratteristiche generali

- Un compilatore C è disponibile su tutti sistemi.
- Codice molto efficiente (veloce).
- Ha caratteristiche di alto livello: adatto per programmi anche complessi.
- Ha caratteristiche di basso livello (accesso ai bit): permette di sfruttare le peculiarità proprie di una macchina o architettura (efficienza).
- Ha disponibili tantissime librerie per aggiungere funzionalità.
- Tra i linguaggi più diffusi, è il più usato per sviluppare software di sistema

Il Linguaggio C:Fasi di Compilazione

- Il **preprocessore** elabora le direttive *#include*, *#define*,... modificando il sorgente
- Il **compilatore** traduce il codice C in linguaggio macchina con ottimizzazione (della velocità di esecuzione o della dimensione dell'eseguibile).
- Il **linker** assembla in un unico file eseguibile i file oggetto prodotti da diversi file sorgente e le librerie (I/O, matematiche, network, ecc.).

Il Linguaggio C: Errori e Warning

Il compilatore verifica la correttezza del codice C e produce due tipi di errori:

- **Error** - Errori sintattici che impediscono la generazione del codice eseguibile;
- **Warning** - Errori non sintattici che non impediscono la generazione del codice eseguibile; i *warning* segnalano un possibile (e altamente probabile) problema che il compilatore risolve in base a regole generiche (ma attenzione: la soluzione generica potrebbe non essere quella corretta).

Un codice pulito non deve produrre né errori né warning

Il Linguaggio C: La Sintassi 1

- I caratteri maiuscoli sono considerati diversi dai corrispondenti minuscoli (il linguaggio C è *case sensitive*).
- Le istruzioni sono una sequenza di caratteri terminate dal carattere **;**
- Quando l'istruzione è il solo carattere **;** essa è detta istruzione nulla e non produce alcuna azione (esempi più avanti nel corso).
- I commenti sono annotazioni sul codice fatte dal programmatore, iniziano con la coppia di caratteri /* e terminano con la coppia */. Essi vengono ignorati dal compilatore che li considera come un unico carattere spazio

Il Linguaggio C: La Sintassi 2

- Le istruzioni possono continuare su più righe.
- Si può andare a capo in ogni punto dove si può mettere uno spazio, esclusi quelli all'interno delle stringhe (sequenze di caratteri delimitate da doppie virgolette, es. ciao ciao).
- Una sequenza (anche mista) di uno o più caratteri quali spazi, tabulatori, ritorni a capo e commenti è considerata dal compilatore equivalente ad un unico spazio (tranne che all'interno delle stringhe)
- Un **blocco** di codice è un insieme di istruzioni racchiuso tra parentesi graffe e costituito, nell'ordine, da due parti:
  1. una sezione opzionale con la definizione di tutte le variabili ad uso esclusivo di quel blocco;
  2. una sezione con le istruzioni eseguibili;
- Per i **blocchi** le parentesi graffe sono opzionali e normalmente omesse se il blocco di codice è costituito da una sola istruzione (salvo il blocco che racchiude il corpo di una funzione, in particolare il main).

Il Linguaggio C: La Sintassi 3

Le istruzioni di un blocco (non le eventuali parentesi graffe) si scrivono tutte indentate di un numero fisso di spazi (es. 3)

```
{
   a = 12;
   b = 23;
   c = a + b;
}
```

L'indentazione viene ignorata dal compilatore ma aiuta il programmatore a comprendere meglio il flusso del programma per cui va fatta *mentre si programma* e non dopo.

Un primo Programma: La somma di due numeri

Vogliamo calcolare la somma di due numeri.

- **Algoritmo** - Definire due numeri reali a, b e calcolare $c = a + b$.
- Con un editor di testi scriviamo il programma:

```
#include <stdio.h>
int main()
{
     float a,b,c;//Definizioni Variabili
     a=3.2f;//Assegnazione diretta di un valore ad una variabile
     b=5.6f;//Assegnazione diretta di un valore ad una variabile
     c=a+b;//Assegnazione con un calcolo di un valore ad una variabile
     printf("%f + %f; = %f\n",a,b,c);//Stampa del risultato
     return 0;
}
```

- Utilizzare un compilatore (es. DevCpp) per tradurlo in linguaggio macchina e farlo eseguire.
- Il risultato è il seguente:

```
3.200000 + 5.600000 = 8.800000
```

Le Variabili

Le variabili sono dei contenitori nei quali vengono memorizzati, in modo ***temporaneo*** o ***definitivo*** , i valori che servono alla corretta esecuzione del programma. Vi è una **corrispondenza biunivoca fra celle di memoria e variabile.**
Al nome di una variabile il computer associa, in modo del tutto trasparente al programmatore, una cella della propria RAM. **I principali tipi di variabili sono i seguenti:**

Tipi di dichiarazione	Rappresentazione	N. di byte
`char`	Carattere	1 (8 bit)
`int`	Numero intero	2 (16 bit)
`short`	Numero intero "corto"	2 (16 bit)
`long`	Numero intero "lungo"	4 (32 bit)
`float`	Numero reale	4 (32 bit)
`double`	Numero reale "lungo"	8 (64 bit)

I nomi delle variabili possono essere caratteri alfanumerici. Il primo deve essere sempre alfabetico e sono esclusi caratteri particolari come, per esempio, lo spazio o la &. Da osservare che nel programma di esempio con l'assegnazione $a = \mathbf{3.2}f$ abbiamo usato i numeri decimali in *singola precisione* con un evidente risparmio di memoria a scapito però della precisione

Programma N.2

Vogliamo decidere se un numero è pari o dispari.

- **Algoritmo** - Dato un numero intero *a*, calcolare il resto *r* della divisione di questo numero per 2.
- **se** il resto *r* è zero il numero è ***pari*** , **altrimenti** ***dispari***
- Con un editor di testi scriviamo il programma:

```
#include <stdio.h>
int main()
{
     int a,r;
     a=7;
     r= a%2 ;
     /* L'operatore % esegue la divisione tra interi
     e ne calcola il resto che viene memorizzato nella variabile r
     */
     if(r==0){
          printf("Il numero %i è pari \n",a);
     }else{
          printf("Il numero %i è dispari \n",a);
     }
     return 0;
}
```

- Il risultato per $a = 7$ è:

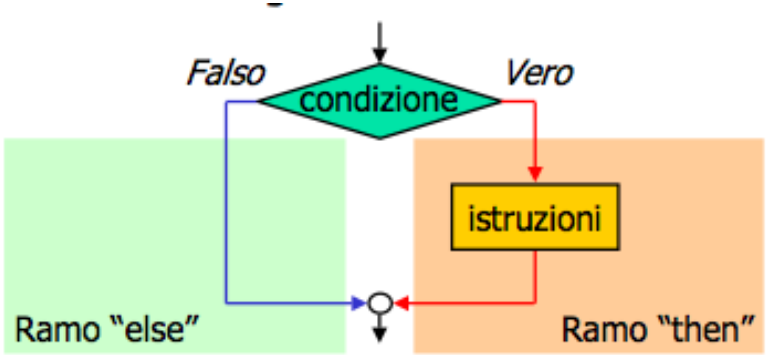
```
Il numero 7 è dispari
```

- Il risultato $a = 12$ è:

```
Il numero 12 è pari
```

Il Programma Sceglie: I salti condizionati

Analizziamo, partendo dal precedente programma, come si possono trattare le diverse alternative. Nel programma precedente abbiamo realizzato il seguente schema



Occorre osservare che in questo caso non sarebbero necessarie le parentesi { e } in quanto i blocchi sono costituiti da un'unica istruzione, ma per chiarezza del codice a volte è opprtuno utilizzarle sempre. Non vi sono limitazione sui blocchi e possono essere formati da qualunque tipo di istruzione anche più scelte *nidificate* all'interno.

Operatori di confronto e Logici

Nelle selezioni si usano gli operatori di confronto e logici mostrati di seguito.

Simbolo	Significato	Utilizzo
==	uguale a	`a == b`
!=	diverso da	`a != b`
<	minore	`a < b`
>	maggiore	`a > b`
<=	minore o uguale	`a <= b`
>=	maggiore o uguale	`a >= b`

Simbolo	Significato	Utilizzo
&&	AND logico	`a && b`
||	OR logico	`a || b`

Da osservere il simbolo == usato per decide l'uguaglianza tra due variabili o espressioni. Questa scelta è resa necessaria dal fatto che nella quasi totalità dei linguaggi di programmazione il simbolo = ha il significato di *assegna il valore* . . .

Massimo Comun Divisore: L'Algoritmo di Euclide

Dati due interi a e b il loro Massimo Comun Divisore (MCD(a,b)) è il più grande intero che li divide entrambi. Ogni intero è divisibile per 1, quinndi 1 è un loro divisore comune ma non è detto che sia il più grande. Se MCD(a,b)=1, i due numeri si dicono *primi tra loro*. Si condideri la divisione intera tra a e b cioè $a = b \cdot q + r$, dove q è il quoziente e r il reso della divisione. Euclide dimostrò che MCD(a,b) = MCD(b,r). Questa dimostrazione porta al seguente algoritmo di calcolo del MCD per due interi assegnati a e b.

1. **Si calcoli il resto r della divisione di a per b.**
2. **Se $r = 0$, b è il MCD cercato.**
3. **Altrimenti a diventa b ($a = b$), b diventa r ($b = r$) e si torna al punto 1.**

Alcune considerazioni.

- **L'esecuzione di questo algoritmo richiede la *ripetizione* di un blocco di istruzione un numero di volte *non predefinito*.**
- **Non si è tenuto conto del fatto che possa essere $a < b$. E' necessario?**

Algoritmo di Euclide: Il Programma

Il programma:

```
#include <stdio.h>
int main()
{
    int a,b,r;
    a=553;
    b=154;
    do{
        printf("MCD(%i,%i)=",a,b);// Senza \n non va a capo
        r=a%b;
        if(r!=0){
            a=b;
            b=r;
        }
    }while(r!=0);
    printf("%i\n",b);//Stampa risultato e va a capo
        return 0;
}
```

Il risultato:

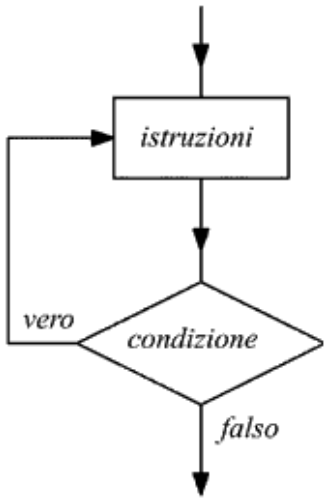
```
MCD(553,154)=MCD(154,91)=MCD(91,63)=MCD(63,28)=MCD(28,7)=7
```

Il Ciclo do. . . while

Per ciclo si intende la *ripetizione* **di un blocco di istruzione. Un tipo di ciclo è il ciclo**

```
do{
     ......
     }
}while(test=vero);
```

che ha la seguente rappresentazione garfica. Osservare che in questo caso il blocco delle istruzioni viene eseguito *almeno una volta*.



La potenza di un numero: a^n

Come è noto dalla definizione di potenza ad esponente intero $a^0 = 1$ e $a^n = a \cdot a \cdot a \cdots n$ *volte*. L'algoritmo è semplice.

- **Assegnare la base a, l'esponente n e la potenza p con il valore iniziale 1.**
- ***Mentre* risulta $n > 0$, eseguire le seguenti operazioni;**
  1. **Assegnare alla potenza p, l'*attuale* valore di p moltiplicato per a ($p = p * a$).**
  2. **Assegnare all'esponente n l'*attuale* valore di n diminuito di 1 ($n = n - 1$).**

Osservare che nel caso in cui $n = 0$ il ciclo non viene mai eseguito e il risultato è 1, come da definizione.

Il Ciclo while 1

```
#include <stdio.h>
int main()
{
    int a,n,p;
    a=2;n=6;p=1;
    printf("(%i)^%i=",a,n);
    while(n>0){
        p=p*a;
        n=n-1;
    }
    printf("%i\n",p);
    return 0;
}
```

Il ciclo *while* può anche non essere eseguito se il test è *falso* fin dall'inizio.

```
    while(test=vero){
       ........
    }
```

Il Ciclo while 2

Di seguito viene riportato lo schema che illustra come il test preceda il blocco di istruzioni. Se il test e *falso* fin dall'inizio le istruzioni non vengono mai eseguite. Particolare attenzione va posta al contenuto del test. E' necessario assicurarsi che il ciclo prima o poi si interrompa.

Nota sulle istruzioni:

```
......
n=6;p=1;
.......
    p=p*a;
    n=n-1;
.......
```

Ricordando che il segno = significa *assegna*, **il primo assegnamento può essere interpretato così:**
memorizza nella cella di memoria p l'attuale contenuto di tale cella moltiplicato per il contenuto della cella a.

E'importante che alle variabili interessate venga asseganto un valore iniziale.

La somma dei primi n numeri

Si vuol calcolare la somma dei primi *n* numeri. Algoritmo:

- *Inizializzare* **a 0 la variabile** *somma* (*somma* = **0**).
- **Aggiungere a** *somma* **il numero 1.**
- **Ripetere il passo precedente per tutti i numeri fino ad** *n*.

Il programma è il seguente

```
 #include <stdio.h>
int main()
{
    int somma,i,n;
    n=20;somma=0;
    for(i=1;i<=n;i++){
        somma=somma+i;
    }
    printf("La somma dei primi %i numeri è %i\n",n,somma);
    return 0;
}
```

Il risultato

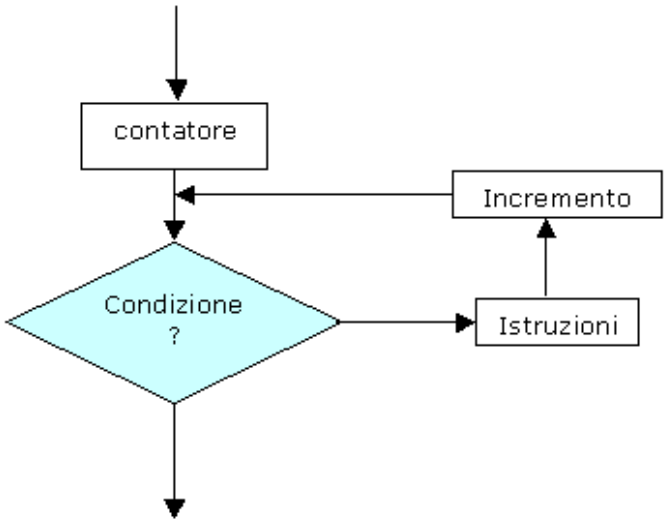
```
La somma dei primi 20 numeri è 210
```

Il Ciclo for

Il ciclo *for* ha la seguente struttura

```
  ......
     for(<int>=<start>;<int>=<end>;<incremento>){
          ........
     }
  ........
}
```

Ad una variabile *intera* vengono progressivamente assegnati i valori a partire da un valore iniziale (*start*) fino ad un valore finale (*end*) secondo un certo *incremento*. Questo tipo di ciclo viene utilizzato quando un blocco di istruzioni deve essere ripetuto un numero ben definito di volte. Di seguito viene rappresentato l'aspetto logico del ciclo for.



Sintesi sui Cicli

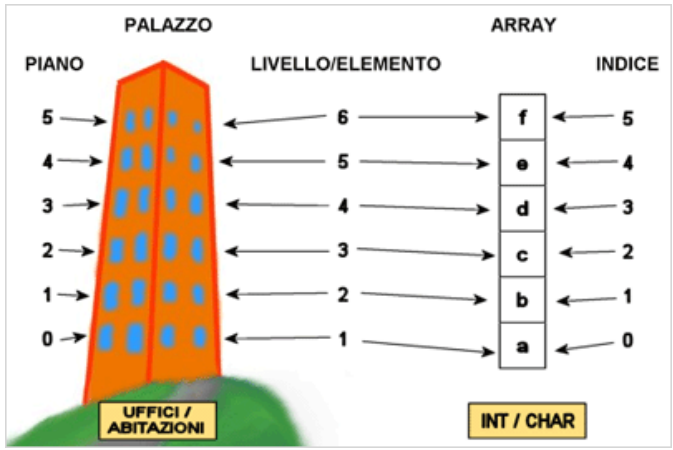
Riassumendo: Con il termine *Ciclo* si intende una situazione nella quale uno o più blocchi di istruzioni debbono essere ripetutepiù di una volta. I cicli sono di tre tipi:

- **Il ciclo *for*. Si utilizza quando è predefinito il numero delle volte che debbono essere ripetute le istruzioni appartenenti a blocchi definiti.E' necessario definire una variabile intera come *contatore*.**
- **Il ciclo *do* $\cdots$ *while*(*test*). Si utilizza quando non è stabilito a priori quante volte debba essere ripetuto un blocco di istruzioni, ma si vuole che tale blocco venga eseguito *almeno una volta*.**
- **Il ciclo *while*. Si utilizza quando non è stabilito a priori quante volte debba essere ripetuto un blocco di istruzioni e non è necessario che tale blocco venga eseguito *almeno una volta*.**

Occorre sottolineare che si può, complicando la sequenza di istruzioni, utilizzare comunque un solo tipo di ciclo.

Le variabili dimensionate: i Vettori (array)

Si consideri il caso di dover descrivere la posizione spaziale di un certo oggetto. Evidentemente avremo a che fare con le sue *coordinate spaziali* **riferite ad un particolare sistema di riferimento e quindi abbiamo bisogno+ di tre numeri. Per scrivere un programma potremmo usare tre variabili distinte che però porterebbero a delle complicazioni non indifferenti. La soluzione è quella di usare un** *vettore* (*array*) **di** *dimensione* **3. Generalizzando un** *vettore* (*array*) **può essere definito come una** *collezione organizzata di oggetti*. **Analizziamo la definizione e capiremo molte cose, innanzitutto il concetto di** *collezione* **implica che tali oggetti siano dello stesso tipo, così, prendendo spunto dal mondo reale, potremmo definire un array di mele, che, quindi non può contenere nessun** *oggetto pera*; **un array in C è una collezione di variabili dello stesso tipo.** *Organizzata* **invdece implica che sia possibile identificare univocamente tutti gli oggeti dell'array in modo sistematico; questo in C viene fatto tramite l'uso di indici numerici che, in un array di dimensione N, vanno da 0 ad N-1. I** *vettori* **possono essere di vari tipi, noi per adesso tratteremo solo quelli numerici.**



Distanza tra due punti 1

Come è noto la distanza d tra due punti $A(a_x, a_y, a_z)$ e $B(b_x, b_y, b_z)$ è data dalla formula

$$d = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$

Osserviamo il programma che la calcola.

```
#include <stdio.h>
#include <math.h>
int main()
{
    float a[3],b[3];
    float s,d;
    int i;
    a[0]=0.6f;a[1]=-0.5f;a[2]=0.98f;
    b[0]=-0.45f;b[1]=0.6f;b[2]=-0.28f;
    s=0.0f;
    for(i=0;i<3;i++)s=s+(a[i]-b[i])*(a[i]-b[i]);
    d=sqrtf(s);
    printf("Distanza = %f\n",d);
    return 0;
}
```

```
Distanza = 1.974867
```

Distanza tra due punti 2

Analizziamo il sorgente.

- **Si includono le librerie matematiche per calcolare la radice quadrata.**

```
#include <math.h>
```

- **Si definiscono due vettori con tre *componenti* per contenere le coordinate dei due punti.**

```
float a[3],b[3];
```

- **Si assegnano i valori a ciascuno dei due *array* facendo riferimento all'*indice* dei singoli elementi.**

```
a[0]=0.6f;a[1]=-0.5f;a[2]=0.98f;
b[0]=-0.45f;b[1]=0.6f;b[2]=-0.28f;
```

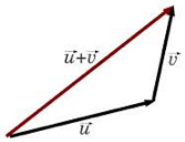
- **Si calcola la distanza.**

```
s=0.0f;
for(i=0;i<3;i++)s=s+(a[i]-b[i])*(a[i]-b[i]);
d=sqrtf(s);
```

Osservare lo 0 iniziale attibuito alla variabile che conterrà la somma e l'uso della singola precisione con *sqrtf*.

Interludio Matematico : Operazioni sui vettori 1

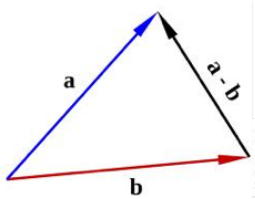
- **Somma di vettori.**



$$\begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} + \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} v_1 + w_1 \\ \vdots \\ v_n + w_n \end{pmatrix}$$

```
for(i=0;i<n;i++)z[i]=v[i]+w[i];
```

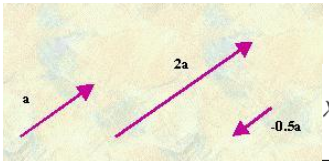
- **Differenza di vettori.**



$$\begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} - \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} v_1 - w_1 \\ \vdots \\ v_n - w_n \end{pmatrix}$$

```
for(i=0;i<n;i++)z[i]=v[i]-w[i];
```

- **Prodotto di un vettore per un numero.**

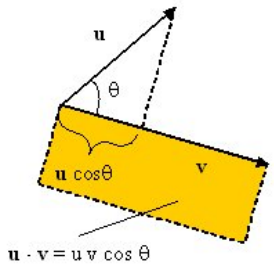


$$\lambda \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} \lambda v_1 \\ \vdots \\ \lambda v_n \end{pmatrix}$$

```
for(i=0;i<n;i++)w[i]=k*v[i];
```

Interludio Matematico : Operazioni sui vettori 2

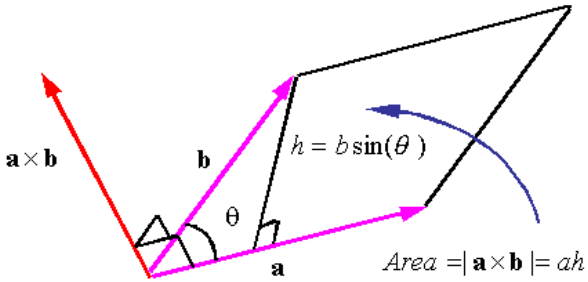
- **Prodotto scalare (*Inner*) tra vettori.**



$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

```
s=0.0f;
for(i=0;i<n;i++)s=s+x[i]*y[i];
```

- **Prodotto vettoriale (*Cross*) tra vettori.**



$$Area = |\mathbf{a} \times \mathbf{b}| = ah = ab\sin(\theta)$$

$$\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \times \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - b_y a_z \\ a_z b_x - b_z a_x \\ a_x b_y - b_x a_y \end{pmatrix}$$

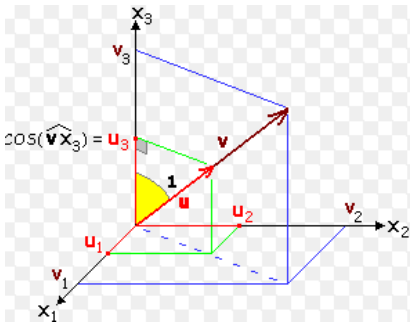
```
c[0]=a[1]*b[2]-b[1]*a[2];c[1]=a[2]*b[0]-b[2]*a[0];
c[2]=a[0]*b[1]-b[0]*a[1];
```

Interludio Matematico : Operazioni sui vettori 3

- **Modulo di un vettore** $\vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ \cdots \\ v_n \end{pmatrix} \rightarrow |\vec{v}| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2} = \sqrt{\sum_{i=1}^{n} v[i]^2}$

```
s=0.0f;
for(i=0;i<n;i++)s=s+v[i]*v[i];
modV=sqrtf(s);
```

- **Versore** $\vec{u}$ **di un vettore** $\vec{v}$; $\vec{u} = \frac{\vec{v}}{|\vec{v}|}$




```
s=0.0f;
for(i=0;i<n;i++)s=s+v[i]*v[i];
modV=sqrtf(s);
if(modV!=0){
   for(i=0;i<n;i++)u[i]=v[i]/modV;
  }
```

Le funzioni: Introduzione 1

Si supponga di avere 4 vettori, $\vec{u}, \vec{v}, \vec{w}, \vec{z}$. Dobbiamo calcolare $\vec{a} = \vec{u} + \vec{v}$ e $\vec{b} = \vec{w} + \vec{z}$. Il programma associato potrebbe essere il seguente:

```
#include <stdio.h>
int main()
{
    float u[3],v[3],w[3],z[3],a[3],b[3];
    int i;
    u[0]=0.1f;u[1]=0.2f;u[2]=0.3f;
    v[0]=0.4f;v[1]=0.5f;v[2]=0.6f;
    w[0]=0.7f;w[1]=0.8f;w[2]=0.9f;
    z[0]=1.0f;z[1]=1.1f;z[2]=1.2f;
    for(i=0;i<3;i++)a[i]=u[i]+v[i];
    for(i=0;i<3;i++)b[i]=w[i]+z[i];
    return 0;
}
```

Si può osservare che la procedura di calcolo del vettore somma è la stessa e cambiano solo gli addendi e il vettore risultato. Se questa operazione dovesse essere ripetuta 100 volte, dovremmo scrivere 100 istruzioni simili e, nel caso di algorimi complessi, renderebbe il programma difficile da gestire.

Le funzioni: Introduzione 2

Si consideri il seguente programma:

```
#include <stdio.h>
int main()
{
    float u[3];
    u[0]=0.1f;u[1]=0.2f;u[2]=0.3f;
    printf("Indirizzo del vettore = %i\n",u);
    printf("Indirizzo Seconda cella = %i\n",&u[1]);
    printf("Contenuto Seconda cella = %f\n",u[1]);
    return 0;
}
```

ed il suo risultato.

```
Indirizzo del vettore = 1606416536
Indirizzo Seconda cella = 1606416540
Contenuto Seconda cella = 0.200000
```

- **Il** *nome* **della variabile vettore indica** *l'indirizzo* **della cella di memria dalla quale inizia il vettore.**
- **la & indica** *l'indirizzo* **della cella di memoria ne quale è memorizzata una variabile qualsiasi.**

Le funzioni 1

Si consideri la modifica al programma per il calcolo della somma di due vettori.

```
#include <stdio.h>
void mathVecAdd(float *res,float *add1,float *add2, int dim);
int main(){
 float u[3],v[3],w[3],z[3],a[3],b[3];
 u[0]=0.1f;u[1]=0.2f;u[2]=0.3f;v[0]=0.4f;v[1]=0.5f;v[2]=0.6f;
 w[0]=0.7f;w[1]=0.8f;w[2]=0.9f;z[0]=1.0f;z[1]=1.1f;z[2]=1.2f;
 mathVecAdd(a,u,v,3);mathVecAdd(b,w,z,3);
 return 0;
}
void mathVecAdd(float *res,float *add1,float *add2, int dim){
    int i;
    for(i=0;i<dim;i++)res[i]=add1[i]+add2[i];
}
```

- **La funzione in ingresso si aspetta tre** *puntatori* **a variabili float e un** *valore* **intero. Queste variabili vengono passate quindi per** *riferimento* **e possono essere** *modificate* **dalla funzione.**
- **La variabile intera viene passata per** *valore*, **la funzione lavora quindi su una sua** *copia* **e non la può modificare.**

Le funzioni 2

Per la sintassi delle funzioni valgono i seguenti casi:

- **Restituiscono un valore di qualuque tipo (intero, float, char, ...). La sintassi è la seguente**

```
<tipo> <nomeFunzione>(<eventuali parametri>){
.....
return <tipo>;
}
```

- **Non restituiscono alcun valore. La sintassi è la seguente**

```
void <nomeFunzione>(<eventuali parametri>){
.....
return;//non necessaria
}
```

- **Se una fuzzione non ha parametri in ingresso la sua sintassi, nel caso che non restituisca alun valore è**

```
void <nomeFunzione>(){
.....
return;//non necessaria
}
```

Funzioni per il calcolo vettoriale 1

- **Somma di due vettori**.

```
void mathVecAdd(float *res,float *add1,float *add2,int dim){
      int i;
      for(i=0;i<dim;i++)res[i]=add1[i]+add2[i];
}
```

- **Differenza di due vettori**.

```
void mathVecSubtract(float *res,float *min,float *sub,int dim){
     int i;
     for(i=0;i<dim;i++)res[i]=min[i]-sub[i];
}
```

- **Prodotto scalare**.

```
float mathVecInner(float *v1,float *v2, int dim){
     int i;
     float s;
     s=0;
     for(i=0;i<dim;i++)s=s+v1[i]*v2[i];
     return s;
}
```

Funzioni per il calcolo vettoriale 2

- **Prodotto vettoriale (è definito solo per vettori tridimensionali).**

```
void mathVecCross(float *res,float *v1,float *v2){
      res[0]=v1[1]*v2[2]-v2[1]*v1[2];
      res[1]=v1[2]*v2[0]-v2[2]*v1[0];
      res[2]=v1[0]*v2[1]-v2[0]*v1[1];
}
```

- **Calcolo del modulo di un vettore.**

```
float mathVecMagnitude(float *v, int dim){
     int i;
     float s;
     s=0;
     for(i=0;i<dim;i++)s=s+v[i]*v[i];
     s=sqrtf(s);
     return s;
}
```

Funzioni per il calcolo vettoriale 3

- **Prodotto di un vettore per un numero.**

```
void mathVecMult(float *res,float *v,float fact, int dim){
      int i;
      for(i=0;i<dim;i++)res[i]=v[i]*fact;
}
```

- **Calcolo del versore associato ad un vettore.**

```
int mathVecNormalize(float *v, int dim){
     float d;
     d=mathVecMagnitude(v,dim);
     if(d==0)return -1;//fallimento
     mathVecMult(v,v,d,dim);
     return 1;//OK
}
```

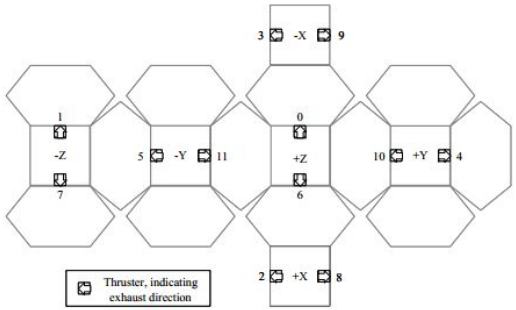
Esercitazione

Utilizzando le funzioni precedenti scrivere i programmi relativi ai seguenti problemi nello spazio tridimensionale:

1. **Calcolare la distanza di un dato punto P dalla retta passante per due punti assegnati A,B utilizzando il vettoriale.**
2. **Calcolare la distanza di un dato punto P dalla retta passante per due punti assegnati A,B utilizzando il prodotto scalare.**
3. **Utilizzando il prodotto scalare determinare se tre punti A, B, C sono allineati o meno.**
4. **Dati tre punti A, B, C non allineati, determinare il vettore perpendicolare al piano che li contiene.**
5. **Dati tre punti A, B, C non allineati determinare, in gradi e radianti, l'angolo tra i vettori $\vec{AC}$ e $\vec{AB}$**
6. **Dati due punti A, B determinare le coordinate di un terzo punto C appartennete alla retta passante per A e B e tale che, dette d_{AB} la distanza di A da B e d_{CB} la distanza di C da B, risulti $d_{CB} = k \cdot d_{AB}$ con $k < 1$.**

SPHERES: la Struttura meccanica

La struttura delle SPHERES è quella mostrata nelle immagini e la massa è di 4.3 Kg.

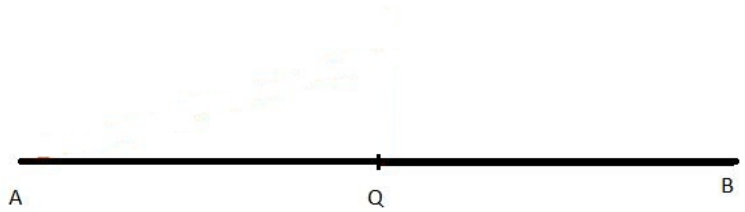


Il movimento avviene per mezzo di 12 propulsori (*thruster*) ciascuno dei quali può essere azionato per 0.2 *sec* e fornisce un forza di *0.12 N*. Il movimento viene attuato attivando 2 propulsori contemporaneamente per una forza impulsiva di 0.24 N per un massimo di 0.2 *sec*. Se vogliamo applicare una forza di f *N* per 1 *sec* i due propulsori debbono essere accesi per un tempo Δt tale che l'impulso prodotto sia uguale a quello richiesto. Quindi $\mathbf{0.24} \cdot \Delta t = f \cdot \mathbf{1}\ sec \Rightarrow \Delta t = \frac{f}{\mathbf{0.24}}$. **Ne deriva che** $f_{max} \cdot \mathbf{1}\ sec = \mathbf{0.24} \cdot \mathbf{0.2} = \mathbf{0.048}\ N \cdot sec$ **e**

$a_{max} = \frac{\mathbf{0.048}}{\mathbf{4.3}} = \mathbf{0.011163}\ \frac{m}{sec^{\mathbf{2}}}$

Il Movimento Lineare

Volendo andare da un punto A ad un punto B dobbiamo prima accelerare il satellite e poi fermarlo in modo di arrivare al punto prescelto.



Vedremo tre modi per eseguire questo compito e ciascuno di essi ha caratteristiche diverse. Per ciascun metodo controlleremo il tempo impiegato e il carburante risparmiato. Il primo metodo che affronteremo lascia al sistema l'intera gestione del processo, utilizzando funzioni di libreria.

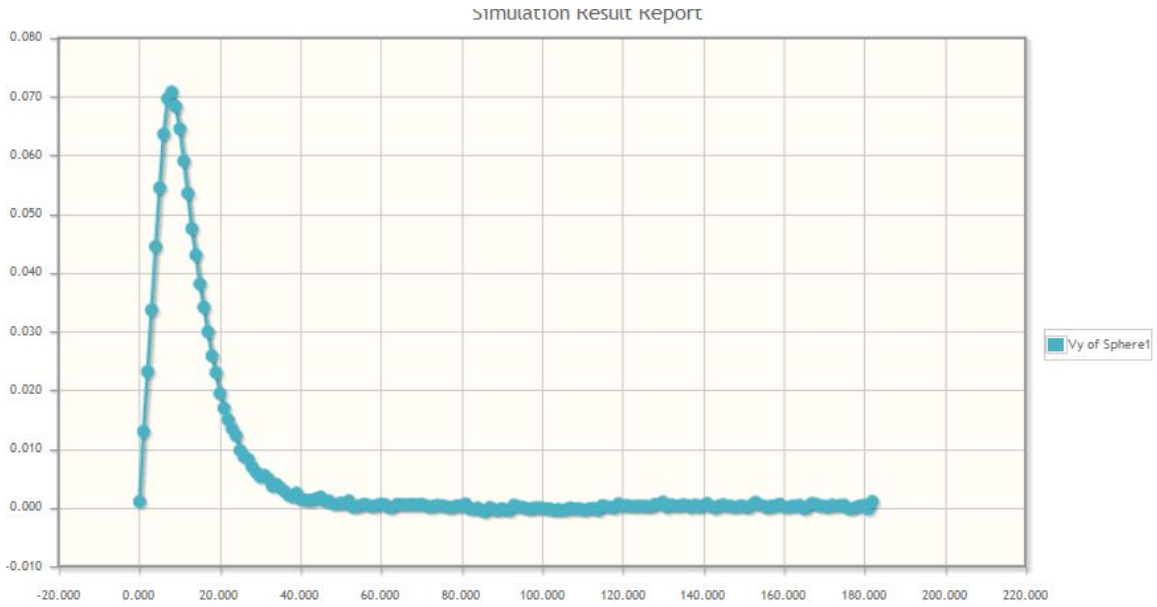
```
float vPositionTarget[3];
void init(){
        vPositionTarget[0]=0.40;vPositionTarget[1]=0.40;
        vPositionTarget[2]=0;
}
void loop(){
        api.setPositionTarget(vPositionTarget);
}
```

Struttura del Programma

Il programma è suddiviso in tre parti fisse e obbligatorie unite ad eventuali parti facoltative.

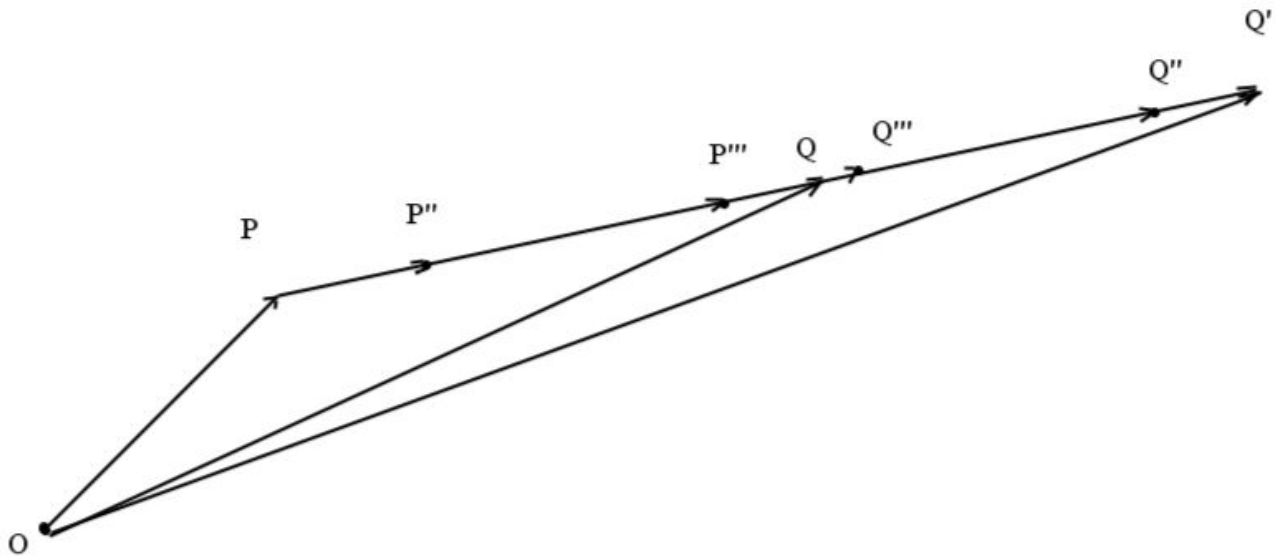
- La sezione di dichiarazione delle variabili nella quale viene definito un vettore *(array* formato da tre celle (*di dimensione 3)* che conterrà le coordinate della nostra *posizione target*, cioè le coordinate del punto di arrivo.
- La sezione di inizializzazione *(init)* nella quale definiamo valori di queste coordinate (unità di misura in metri e sistema di riferimento una zona virtuale della stazione spaziale)
- La sezione del *(loop)* nella quale compare l'unica istruzione che fa muovere il satellite fino alla posizione desiderata. L'istruzione eseguita fa riferimento ad una *funzione di sistema* cioè appartiene a delle librerie (*api*) predefinite.
- La parte facoltativa è costituita da una *pagina* nella quale mettere le funzioni dell'utente. Anche se facoltiva è raccomandata per la chiarezza del programma.

Analisi Cinematica Movimento Lineare



- **Il tempo impiegato è circa 40** *sec*, **la velocità massima è circa 0.07** $\frac{m}{s}$.
- **Il carburante residuo è circa il 65 % di quello iniziale.**
- **Il moto per la prima metà del percorso è uniformemente accelerato, nella seconda metà inizialmente è uniformemente decelerato e poi la velocità tende a zero in modo quasi *esponenziale*.**
- **Se proviamo a definire un *target* più lontano la velocità massima aumenta**

Spostamento veloce : Algoritmo



Sia O l'origine del sistema di assi cartesiani, P il punto di partenza e Q quello di arrivo. Il calcolo vettoriale ci fornisce le seguenti relazioni : $P\vec{Q} = O\vec{Q} - O\vec{P}$, $O\vec{Q}' = O\vec{P} + P\vec{Q}'$, $P\vec{Q}' = k \cdot P\vec{Q}$, con k numero reale > 1. Quando siamo nel punto P e vogliamo andare nel punto Q, diciamo al sistema di puntare a Q' punto più lontano e quindi il sistema fa muovere il satellite più velocemente. Avendo sempre Q come *target* finale, quando siamo in P'' puntiamo a Q'', da P''' puntiamo a Q'''. Più grande è k maggiore sarà la velocità ma non possiamo *esagerare*, come vedremo.

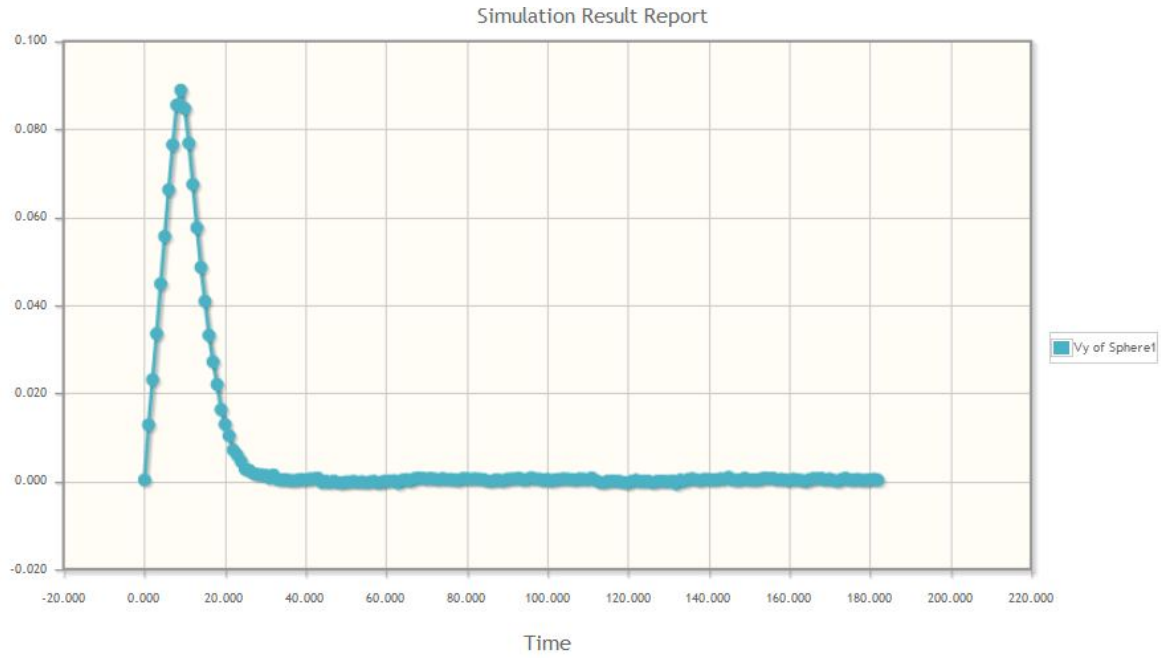
Spostamento veloce : Il Programma

```
float vPositionTarget[3];
state_vector myState;//Vettore di 12 celle
void init(){
  vPositionTarget[0]=0.40;vPositionTarget[1]=0.40;
  vPositionTarget[2]=0;
}

void loop(){
api.getMySphState(myState);//Acquisisce lo stato del sistema
setPosFaster(1.5);//Chiama la funzione
}
```

```
float setPosFaster (float k){
   float v[3],r;
   mathVecSubtract(v,vPositionTarget,myState,3);//Calcolo vettore PQ
   r=mathVecMagnitude(v,3);// Calcolo la distanza PQ
   mathVecMult (v,v,k,3);// Moltiplicazione il vettore PQ per k
   mathVecAdd (v,v,myState,3);// Calcolo vettore OQ'
   api.setPositionTarget(v);// Tende al nuovo Target
return r; // Restituzione la distanza dall'obbiettivo
}
void mathVecMult (float *v, float *a,float k,int dim){
  int i;
  for (i=0;i<dim;i++)v[i]=a[i]*k;
}
```

Spostamento Veloce: Analisi Cinematica

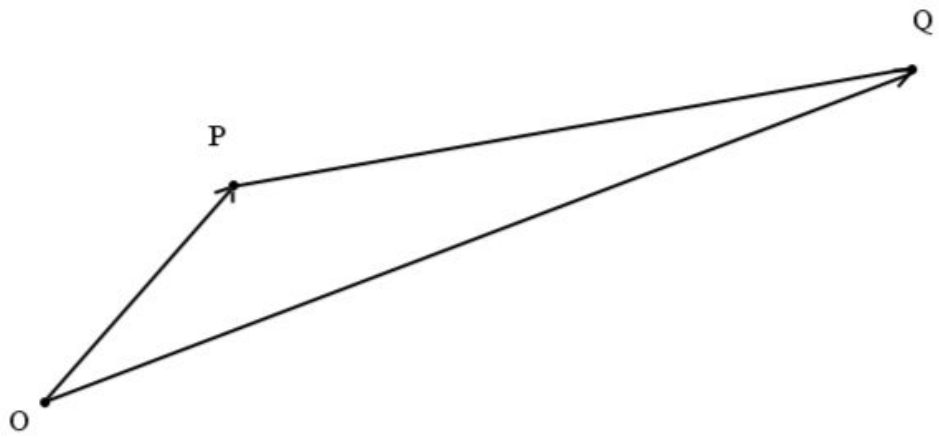


Si possono osservare i segenti fatti:

- Il tempo impiegato è circa 25 *sec* (contro 40 *sec*), il carburante rimasto 63% (contro 65%).
- Il moto è quasi *simmetricamente* accelerato e poi decelerato.

Chiaramente questo metodo e preferibile all'altro. Il valore di k non può essere troppo alto altrimenti si ha un notevole peggioramento delle prestazione. Provare cosa accade per k=3.

Spostamento Uniforme: Algoritmo



Utilizzando le funzioni di sistema si può far muovere la sfera di moto uniforme tranne che nella fase di partenza e di frenata dove, ovviamente il moto è accelerato. Il vantaggio è che il consumo di carburante avviene solo nella fase iniziale e in quella di arresto. Queste fasi dipendono dalla velocità che si vuol raggiungere. La velocità deve avere direzione e verso di $\vec{PQ}$, e modulo assegnato. Pertanto dobbiamo determinare il *versore* della direzione $\vec{PQ}$ e quindi moltiplicarlo per la velocit‡ desiderata. La funzione di sistema

```
api.setVelocityTarget(v);
```

che richiede in ingresso un vettore di *dimensione* 3 rappresentante la velocità, si preoccupa di raggiungere e mantenere la velocità richiesta.

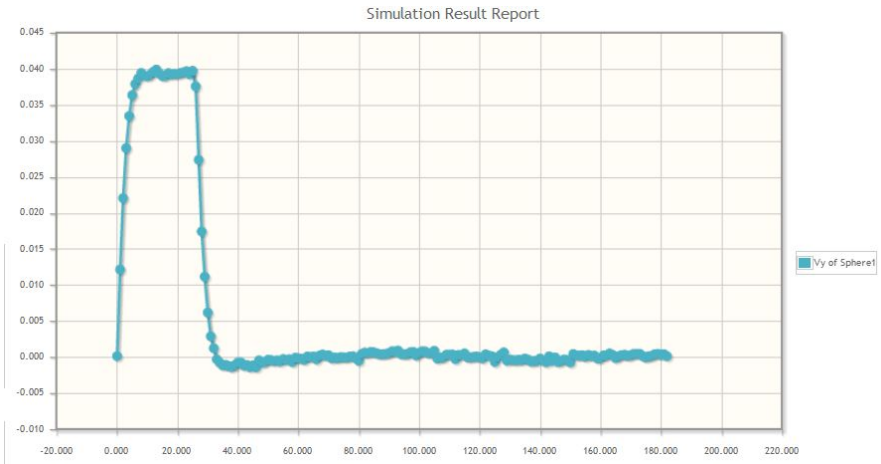
Moto Uniforme : Il Programma

```
float vPositionTarget[3];
state_vector myState;//Vettore di 12 celle
void init(){
  vPositionTarget[0]=0.40;vPositionTarget[1]=0.40;
  vPositionTarget[2]=0;
}

void loop(){
api.getMySphState(myState);setUniformSpeed (0.04);
}
```

```
float setUniformSpeed (float speed){
 float v[3],r;
 mathVecSubtract(v,vPositionTarget,myState,3);//Calcola vettore PQ
 r=mathVecMagnitude(v,3);//Calcolo distanza dal target
 mathVecNormalize(v,3);//Calcolo versore di PQ
 mathVecMult(v,v,speed,3);// Calcolo della velocit‡
 if(r<0.0887)
  api.setPositionTarget(vPositionTarget);//Siamo vicini? fremiamo
 else
  api.setVelocityTarget(v);//Altrimenti manteniamo la velocit‡
 return r;// restituisce la distanza dal target
}
void mathVecMult (float *v, float *a,float k,int dim){
  int i;
  for (i=0;i<dim;i++)v[i]=a[i]*k;
}
```

Spostamento Uniforme: Analisi Cinematica

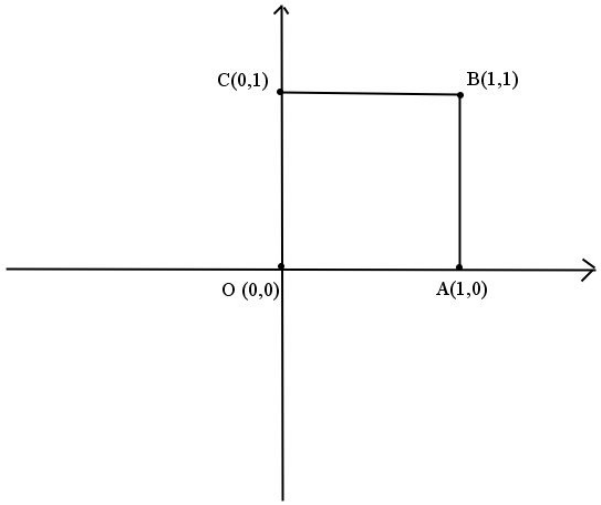


Si possono osservare i segenti fatti:

- Il tempo impiegato è circa 29 *sec* (contro 40 e 20 *sec*), il carburante rimasto 73% (contro 65 e 63%).
- Il moto è quasi costante eccetto che nella fase di partenza e di arresto.

Questo mezzo è meno efficiente se si vuole essere veloci, ma lo è di più se vogliamo risparmiare. Normalmente al risparmio di carburante viene dato un valore doppio rispetto al tempo. Ogni unità di percentuale di carburante risparmiata vale il doppio di ogni secondo di anticipo. La scelta fra i vari metodi dipende dalle situazione e la scelta *vincente* in assoluto non esiste.

Movimento su un Quadrato : Algoritmo

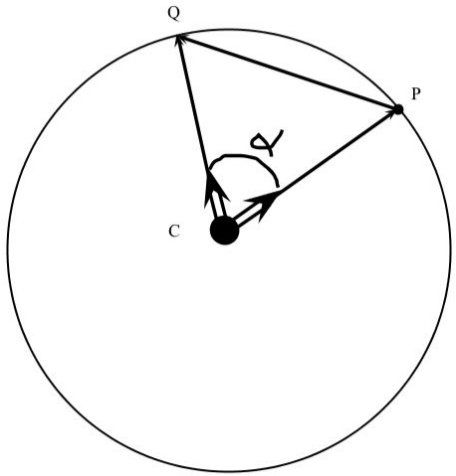


Vogliamo far descrivere al satellite un quadrato nel piano (x,y) partendo dall'origine O e seguendo i vertici A, B, C come in figura e ritornando alla fine in O. Decidiamo che il satellite è arrivato in un vertice quando la sua distanza da questo è minore di 5 cm, (0.05 m) e la sua velocità è minore di 0.01 $\frac{m}{s}$. Supponiamo il satellite come una *macchina a stati* a seconda dello stato nel quale si trova compie una certa azione e può cambiare di stato al verificarsi di certe condizioni. Nel nostro caso l'azione consiste nell'andare in avanti verso un vertice e cambiare vertice e stato una volta che ne abbiamo raggiunto uno. Completato il quadrato ci si ferma.

Moto Uniforme : Il Programma

```
float vPositionTarget[3];
state_vector myState;
float actSpeed;// Velocità attuale;
float actDist;//Distanza attuale dal terget
int sphState;//Stato del satellite
void init(){
  vPositionTarget[0]=0;vPositionTarget[1]=1;vPositionTarget[2]=0;
  sphState=100;//Il satellite percoore il primo lato
}
void loop(){
api.getMySphState(myState);
actSpeed=mathVecMagnitude(&myState[3],3);
actDist=setUniformSpeed (0.04);
switch(sphState){
  case 100:
       if(actDist<0.05 && actSpeed<0.01){
          vPositionTarget[0]=1;vPositionTarget[1]=1;sphState=200;
       }
       break;
  case 200:
       if(actDist<0.05 && actSpeed<0.01){
          vPositionTarget[0]=1;vPositionTarget[1]=0;sphState=300;
       }
       break;
  case 300:
       if(actDist<0.05 && actSpeed<0.01){
          vPositionTarget[0]=0;vPositionTarget[1]=0;sphState=400;
       }
       break;
  case 400: break;
  }
}
```

Movimento Circolare : Algoritmo 1



- **Scopo : vogliamo far muovere il satellite di moto circolare uniforme su una circonferenza di raggio R, posta sul piano (x, y), centro l'origine e con velocità angolare ω $\frac{rad}{sec}$.**
- **La differenza angolare tra una posizione e la successiva è data dalla relazione $\alpha = \omega \cdot \delta t$, dove δt è l'intervallo di tempo che intercorre tra una posizone e l'altra. Poichè nel caso delle SPHERES $\delta t = 1$ sec, avremo $\alpha = \omega$ rad.**
- **Poichè il centro coincide con l'origine degli assi, la posizione attuale del satellite è data dal vettore $\vec{OP}$ del quale viene calcolato il** *versore* $\vec{v} = \frac{\vec{OP}}{|\vec{OP}|}$.
- **Applicando al versore $\vec{v}$ l'equazione della rotazione di un angolo α e moltiplicadola per R otteniamo il vettore $\vec{w} = \vec{CQ}$ che rappresenta la posizione dopo un secondo alla quale tende il satellite.**

$$\vec{w} = \vec{CQ} \quad = \begin{cases} w[\mathbf{0}] = R * (v[\mathbf{0}] * \mathbf{cos}(\alpha) - v[\mathbf{1}] * \mathbf{sin}(\alpha)) \\ w[\mathbf{1}] = R * (v[\mathbf{0}] * \mathbf{sin}(\alpha) + v[\mathbf{1}] * \mathbf{cos}(\alpha)) \end{cases}$$

- **Il** *versore* $\vec{v}$ **della differenza $\vec{CQ} - \vec{CP}$, rappresenta la direzione del moto che deve avere il satellite. Quindi $\omega * R * \vec{v}$, rappresenta la velocità tangenziale il satellite.**

Moto Circolare : Il Programma

```
float Radius,Omega,alpha,cosalpha,sinalpha;
state_vector myState;

void init(){
  Radius=0.50;Omega=0.14;
  alpha=0.14;cosalpha=cosf(alpha);sinalpha=sinf(alpha);
}

void loop(){
  api.getMySphState(myState);
  goOnCircle(Radius);
}
```

```
void goOnCircle(float radius){
  float v[3],w[3];
  mathVecMult(v,myState,1,3);
  mathVecNormalize(v,3);
  w[0]=radius*(v[0]*cosalpha-v[1]*sinalpha);
  w[1]=radius*(v[0]*sinalpha+v[1]*cosalpha);
  w[2]=0;
  mathVecSubtract(v,w,myState,3);
  mathVecNormalize(v,3);
  mathVecMult(v,v,Omega*radius,3);
  api.setVelocityTarget(v);
}
void mathVecMult(float *v, float *a,float k,int dim){
  int i;
  for(i=0;i<dim;i++)v[i]=a[i]*k;
}
```

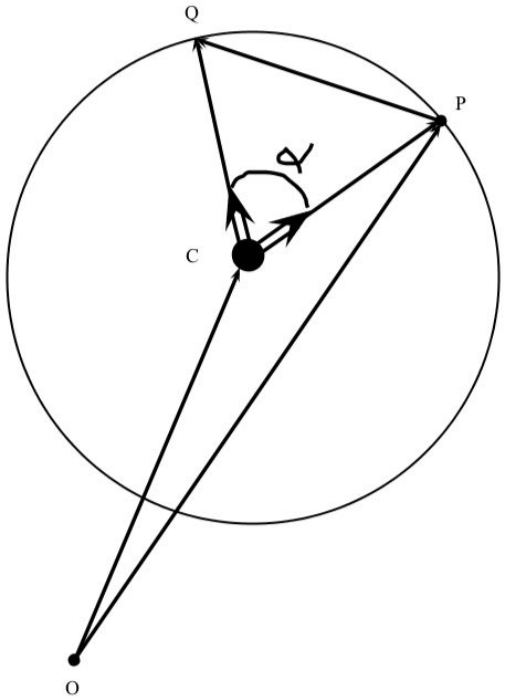
Movimento Circolare : Note e Indicazioni

- **La rotazione è stata applicata al** *versore* **di** $\vec{CP}$ **e non direttamente al vettore** $\vec{CP}$ **in quanto non sappiamo se, per errori di approssimazione, il punto P si trovi esattamente sulla circonferenza. Operando sui** *versori* **siamo sicuri che il punto Q si troverà sulla cirsonferenza.**
- **La rotazione cessa quando scade iltempo della simulazione. Modificare la funzione** *goOnCirlce* **in modo che in ingresso riceva un angolo oltre il quale cessa di girare e compie un altro movimento, per esempio si posiziona nel centro della circonferenza che in questo caso coincide con l'origine.**

Seguono indicazioni per realizzare la modifica.

- **Dove memorizzare l'angolo totale descritto? Serve una variabile globale** *angoloTotale* **di valore iniziale 0 e che viene incrementata (di quanto ?) all'interno della funzione** *goOnCircle*.
- **Quando fermarsi? Serve una variabile globale che memorizza il valore raggiunto il quale occore dirigersi verso l'origine.**
- **E quando si è raggiunto il valore limite dell'angolo? Non si assegna più la velocità ma ci si dirige verso l'origine con** *api.SetPosition*
- **A parte le variabili globali che vanno** *definite* **e** *inizializzate* **nel** *main*, **tutte le altre modifiche debbono essere effettuate in** *goOnCircle*

Movimento Circolare : Algoritmo 2



- **Scopo : vogliamo far muovere il satellite di moto circolare uniforme su una circonferenza di raggio R, posta sul piano (x, y), centro il punto C, distinto dall'origine, e con velocità angolare ω $\frac{rad}{sec}$.**
- **Le uniche modifiche da eseguire riguardano la chiamata della funzione *goOnCricle* che adesso richiede le coordinate del centro e in pratica basta calcolare $\vec{CP} = \vec{OP} - \vec{OC}$ tenedo conto della traslazione dal centro degli assi coordinati a quello della circonferenza..**

Moto Circolare 2: Il Programma

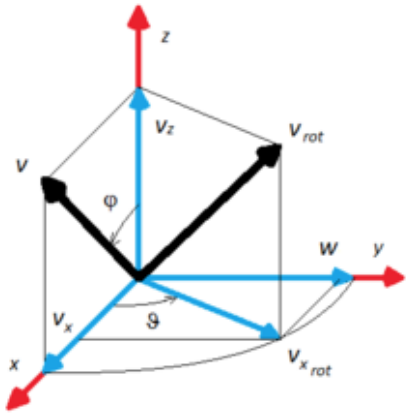
```
float Radius,Omega,alpha,cosalpha,sinalpha;
state_vector myState;
float Centro[3];

void init(){
  Radius=0.50;Omega=0.14;
  alpha=0.14;cosalpha=cosf(alpha);sinalpha=sinf(alpha);
  Centro[0]=0;Centro[1]=0.25;Centro[2]=0;
}
void loop(){
  api.getMySphState(myState);
  goOnCircle(Radius,Centro);
}
```

```
void goOnCircle(float radius,float c[3]){
  float v[3],w[3],sup[3];
    mathVecMult(v,myState,1,3);mathVecMult(sup,myState,1,3);
  mathVecSubtract(v,v,c,3);//Traslazione nel centro
  mathVecNormalize(v,3);
  w[0]=c[0]+radius*(v[0]*cosalpha-v[1]*sinalpha);
  w[1]=c[1]+radius*(v[0]*sinalpha+v[1]*cosalpha);w[2]=0;
  mathVecSubtract(v,w,sup,3);
  mathVecNormalize(v,3);
  mathVecMult(v,v,Omega*radius,3);
  api.setVelocityTarget(v);
}
void mathVecMult(float *v, float *a,float k,int dim){
  int i;
  for(i=0;i<dim;i++)v[i]=a[i]*k;
}
```

Circonferenza nello spazio

Nel caso che la rotazione debba avvenire nello spazio attorno ad asse qualunque $\vec{k}$ è opportuno utilizzare la formula di *Rodrigues* che è implemetata nella funzione riportata di seguito.



$$\begin{aligned}\mathbf{v}_{\text{rot}} &= \mathbf{v}_{x\ \text{rot}} + \mathbf{v}_{z\ \text{rot}} \\ &= \mathbf{v}_{x\ \text{rot}} + \mathbf{v}_{z} \\ &= (\mathbf{v} - (\mathbf{k}\cdot\mathbf{v})\mathbf{k})\cos\theta + (\mathbf{k}\times\mathbf{v})\sin\theta + (\mathbf{k}\cdot\mathbf{v})\mathbf{k} \\ &= \mathbf{v}\cos\theta + (\mathbf{k}\times\mathbf{v})\sin\theta + \mathbf{k}(\mathbf{k}\cdot\mathbf{v})(1-\cos\theta),\end{aligned}$$

```
void executeRotation(float *vr, float *k, float *v, float costh, float sinth){
    float rr,ww[3];
  rr=mathVecInner(k,v,3);
  mathVecMult(ww,k,rr*(1-costh),3);
  mathVecMult(vr,ww,1,3);
  mathVecCross(ww,k,v);
  mathVecMult(ww,ww,sinth,3);
  mathVecAdd(vr,vr,ww,3);
  mathVecMult(ww,v,costh,3);
  mathVecAdd(vr,vr,ww,3);
}
```