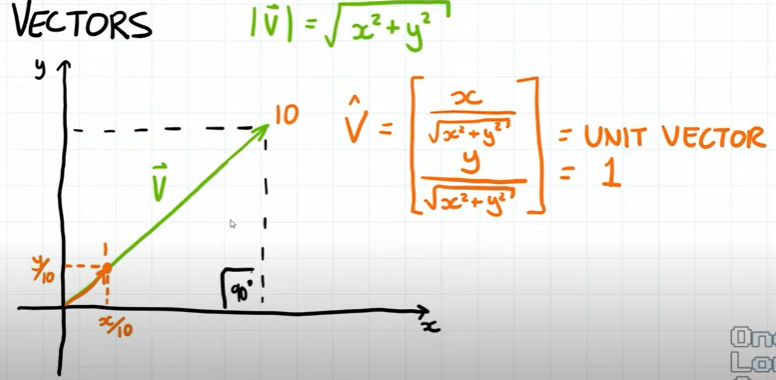
**ESSENTIAL MATHEMATICS**

**UNITARY VECTOR – NORMALIZING**

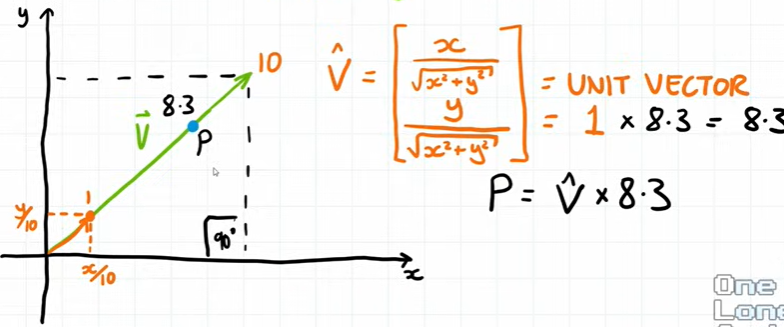
Ogni vettore può essere “normalizzato” – reso di lunghezza (o magnitudo) = 1 – calcolando la lunghezza  
di tale vettore attraverso il teorema di pitagora e dividendo le due componendi x ed y per tale lunghezza



Ricordiamo che le componenti fondamentali di un vettore sono **lunghezza** e **direzione**. Attraverso la   
normalizzazione possiamo “eliminare” la componente lunghezza per tenere conto solo della direzione.

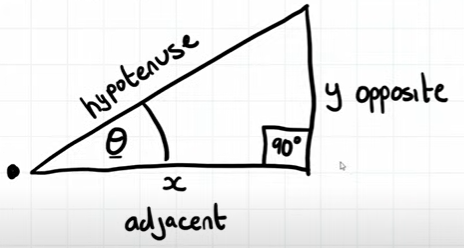
• Supponiamo di voler **trovare un punto lungo la direzione del mio vettore che dista un dato valore   
 dall’origine**:

Quello che dovrò fare sarà semplicemente moltiplicare il mio vettore unitario per tale distanza



**ANGLES**

**GRADI**



**Formule in angoli**

NOTA: In molti linguaggi di programmazione invece di troviamo

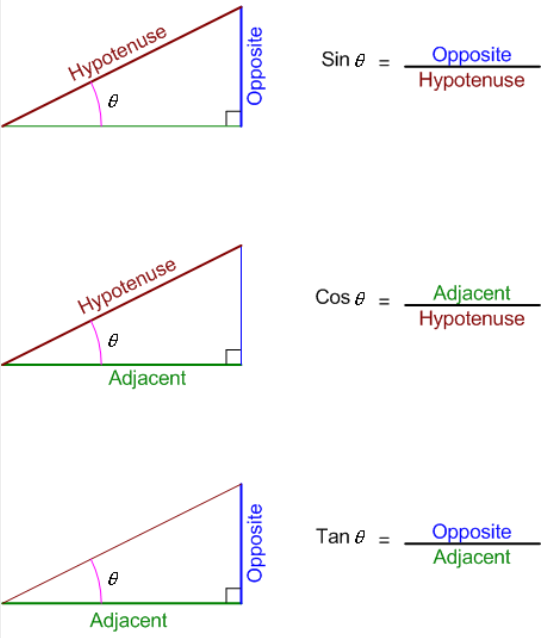
Bisogna però fare attenzione ad esempio che x sia diversa da zero, per questo  
spesso di fa uso della funzione **atan2** che esegue tutta una serie di controlli

**RADIANTS**  
• Solitamente i programmi non elaborano questo tipo di informazioni in termini di “angoli” ma di **radianti**.

**Conversione**

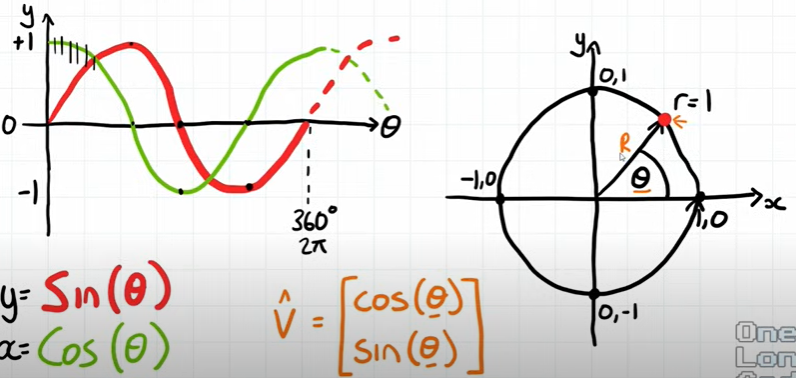
**Formule**

Un modo per memorizzare le relazione che si creano fra ipotenusa ed assi è l’acronimo **SOH CAH TOA**



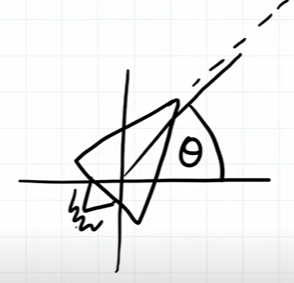
**SIN & COS**

Possiamo descrivere un vettore unitario in termini di seno e coseno



Questo risulta utile nella programmazione di un gioco: Supponiamo di avere un’astronave e di conoscere   
di cui conosciamo solo l’angolo rispetto all’asse cartesiano

Attraverso le coordinate polari posiamo trasformare questo angolo in un “vettore direzione” lungo il quale  
viaggerà l’astronave. Sarà inoltre semplice ruotare l’astronave in quanto basterà aumentare o diminuire   
l’angolo

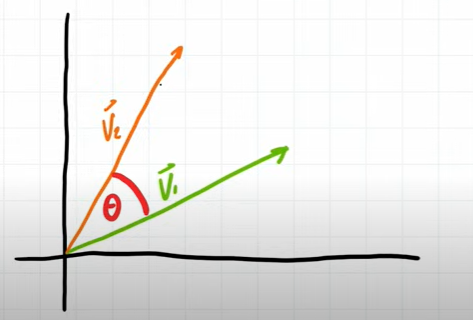


Passare da angoli a vettori può essere utili a seconda dei casi: Ruotare un oggetto verrà più facile in termini   
di angoli mentre muoverlo in una certa direzione richiederà l’uso di vettori

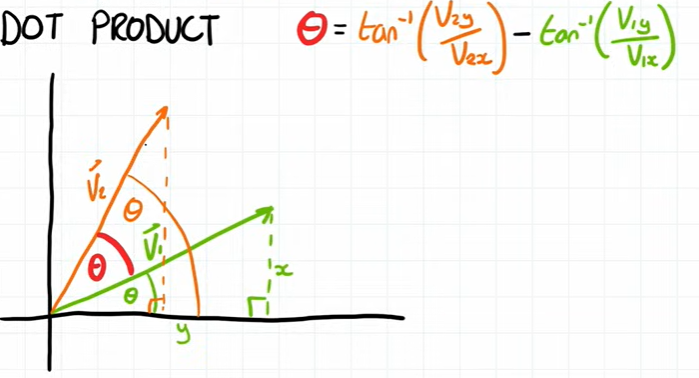
NOTA: Sin, Cos, tan, sqrt risultano **pesanti in termini computazionali**. Mentre i calcoli richiesti dai vettori  
 risultano più leggeri (risultano quindi preferibili)

**DOT PRODUCT**Dato il discorso sul costo computazionale, risulta utile il “dot product” che ci permette di lavorare sugli   
angoli in vermini di vettori.

Supponiamo di avere due vettori sul piano e di voler calcolare l’angolo che formano fra loro

****  
Cominciamo rappresentando questo vettori con vettori unitari in quanto il Dot Product sarà dato  
dalla somma del prodotto delle singole componenti dei vettori.

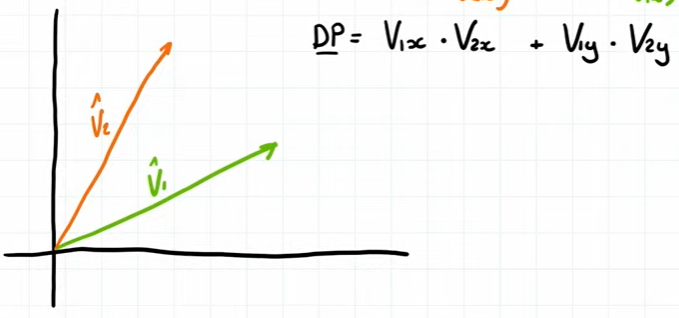
Vediamo prima il metodo classico per calcolarlo:



Come vediamo ci sono diversi calcoli da fare ed il costo computazionale sarebbe elevato.

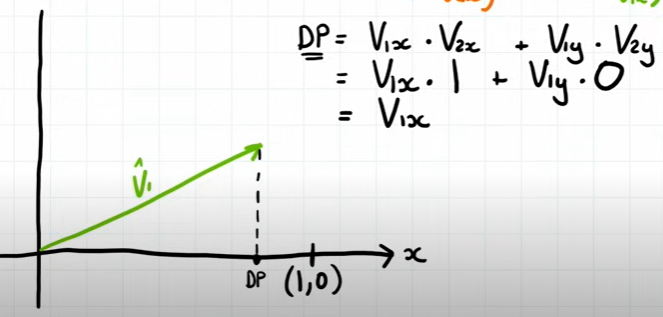
• C’è un altro modo per risolvere questo problema. Ricordiamo che un vettore è composto da una direzione ed una lunghezza. In questa situazione la lunghezza risulta però irrilevante perché al variare di quest’ultima non cambia l’angolo fra i due vettori.

Possiamo quindi considerare i vettori unitari ed il dot product sarà dato dalla **somma del prodotto delle singole componenti dei vettori**



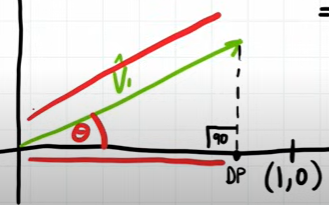
Questo sarà un risultato scalare (un numero) e non un vettore. Vediamo perché.

• Consideriamo un singolo vettore e che anche il nostro asse x sia un vettore unitario. In questo caso il   
 prodotto scalare sarà:



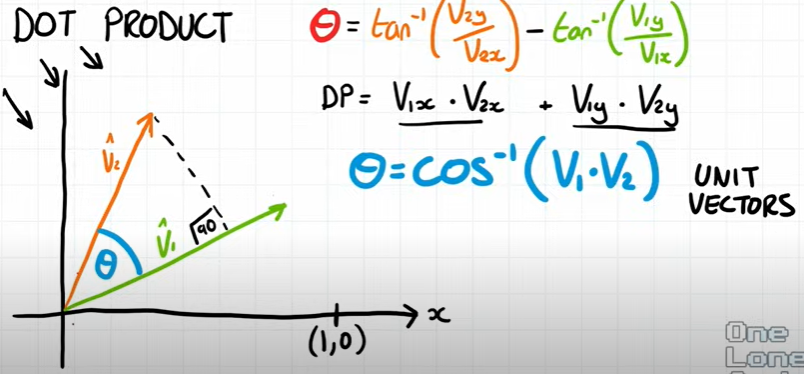
Il dot product rappresenterà quindi **la proiezione di un vettore su un altro** (un punto quindi).

• Ora possiamo notare che attraverso questa operazione abbiamo creato un triangolo rettangolo di   
 cui conosciamo l’adjacent e l’hypotenuse e possiamo quindi calcolare l’angolo



Abbiamo quindi calcolato l’angolo con l’asse delle x per uno dei nostri vettori.

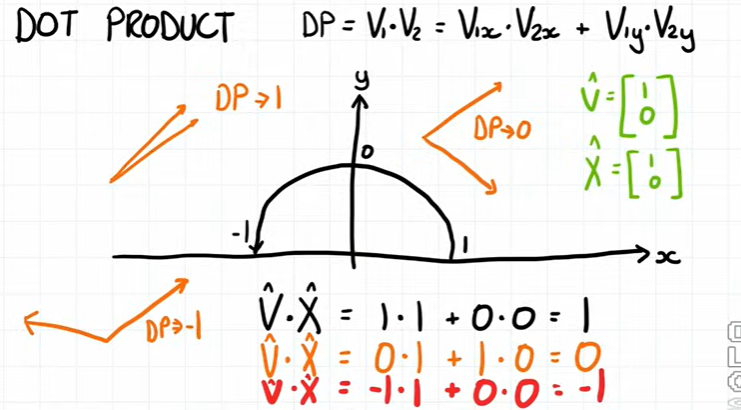
• Tornando ora ai nostri due vettori iniziali possiamo usare lo stesso meccanismo ma, invece di proiettare   
 sull’asse delle x andremo a proiettare V2 su V1 (in modo da formare un angolo retto) per poi trovare  
 l’angolo fra questi due.



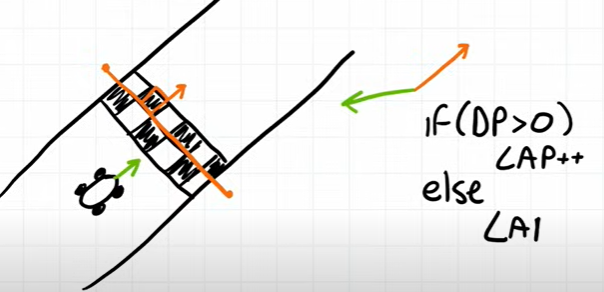
NOTA: **Ricordiamo che la formula scritta in questo modo vale solo per vettori unitari**

• Consideriamo l’immagine sotto. Possiamo notare che

Vettori paralleli 🡪 PD = 1  
Vettori perpendicolari 🡪 PD = 0  
Vettori opposti 🡪 PD = -1



Possiamo quindi aspettarci che vettori quasi paralleli daranno un PD vicino ad 1, vettori quasi   
perpendicolari un PD vicino a 0 e vettori quasi opposti un PD vicino a -1

Come può essere utili questa approssimazione ?  


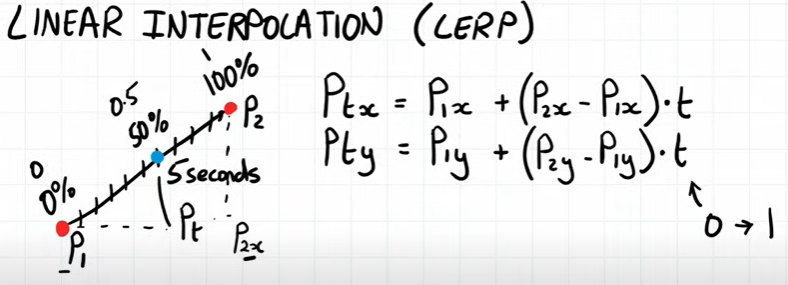
Consideriamo un gioco di corse in cui, al passaggio della linea di arrivo, un contatore di giri venga aumentato. La mia auto avrà un vettore direzione e l’arrivo può essere rappresentato da un secondo vettore al quale andrò ad associare un secondo vettore direzione perpendicolare ad esso.  
  
A questo punto posso dire che:   
Se nel momento dell’attraversamento dell’arrivo il PD > 0 (quindi i due vettori sono più o meno paralleli)   
aumenta il counter dei giri.  
Se invece i due vettori sono opposti (quindi sto attraversando la pista al contrario) diminuiscilo

Cosa comporta tutto questo?  
Che possiamo esegui tutta una serie di azioni senza ricorrere a funzioni come l’inverso del coseno (che   
richiederebbero un alto costo computazionale), semplicemente ricorrendo ad un’approssimazione attraverso il prodotto scalare

**LINEAR INTERPOLATION**  
• Supponiamo di avere due punti nello spazio P1 e P2 e di voler viaggiare da uno all’altro in esattamente  
 5 secondi.  
 Questo si traduce in un render della posizione lungo questa linea in posizioni diverse durante questi  
 5 secondi.

Ci può aiutare pensare al completamente del percorso in termini di percentuali. La partenza sarà allo 0%  
e l’arrivo al 100%.

Consideriamo il punto mediano del percorso e chiamiamolo Pt. Questo avrà una componente x ed y



Noi conosciamo le coordinate x ed y del punto di partenza e arrivo quindi, ad esempio per la coordinata x,  
la posizione di Ptx sarà data dalla coordinata x di partenza + (la coordinata x di arrivo – la coordinata x di partenza) [quindi la lunghezza della proiezione della retta sull’asse delle x] moltiplicato per t  
dove t rappresenta la mia percentuale (in termini computazionali 50% sarà 0,5)

(Cioè prendo la metà della lunghezza P1x e P2x e la sposto di P1x sull’asse)

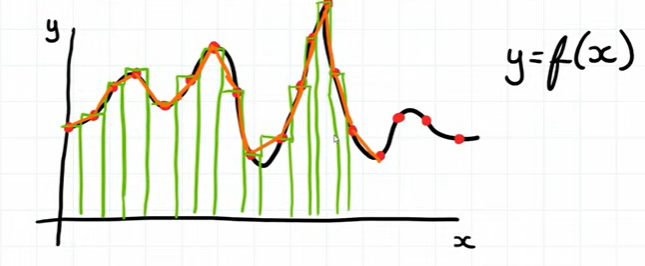
Stessa cosa per l’asse y

In generale:  
**Dati due punti nello spazio possiamo trovare un terzo punto fra questi due** attraverso la formula

Ora supponiamo che trascorrano 0.1 s per frame e che dall’inizio del movimento io accumuli   
questo tempo T = T + 0,1

Voglio che l’intero percorso duri 5 secondi quindi esprimerò T / 5 come la percentuale t

• L’interpolazione lineare ci permette di implementare funzioni che sarebbe difficile descrivere matematicamente

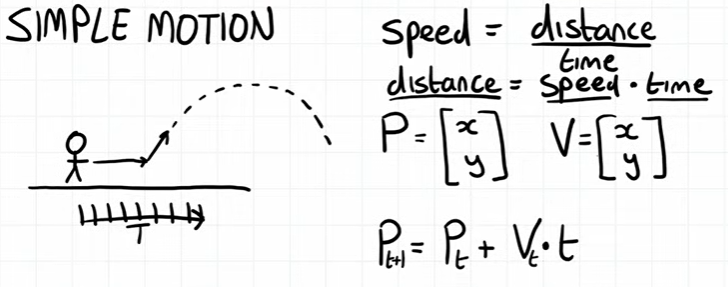


Fornendo valori conosciuti sulla curva (punti rossi) possiamo interpolarli (linee arancioni) fornendo   
un’approssimazione della curva

**Simple motion**  
Come abbiamo detto l’interpolazione lineare può essere usata per far seguire un determinato percorso al nostro personaggio, ma cosa succede se **non conosciamo in anticipo il percorso**?

Supponiamo ad esempio di far saltare il nostro personaggio durante un percorso lineare.  
La curva che seguirà non è predefinita (e soprattutto sarà un percorso non lineare)

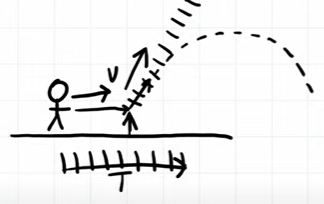
Partiamo dal calcolare il punto P nello spazio in cui si troverà il giocatore



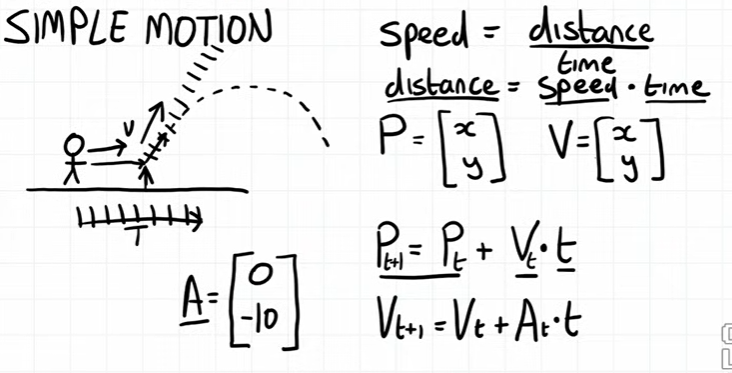
P avrà due componenti x ed y (può essere considerato una sorta di vettore), V rappresenta il vettore   
velocità e t (delta t) sarà l’intervallo temporale fra un frame e l’altro

Avrò quindi che la posizione del giocatore all’intervallo temporale successivo t+1 sarà la somma  
delle componenti di P all’istante t e della velocità V al tempo t moltiplicata per l’intervallo di tempo

Il problema però è che, avendo il personaggio un vettore velocità, questo continuerà a muoversi in quella direzione. Premendo il tasto salto questo vettore velocità cambierà direzione (e volendo anche magnitudo)  
ma il moto risulterebbe lineare



Dobbiamo quindi implementare l’accelerazione gravitazionale che andrà a modificare la velocità del personaggio



Con A che simula un accelerazione negativa di -10 sull’asse delle y