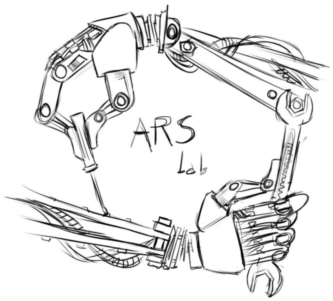
Controllo PID in posizione

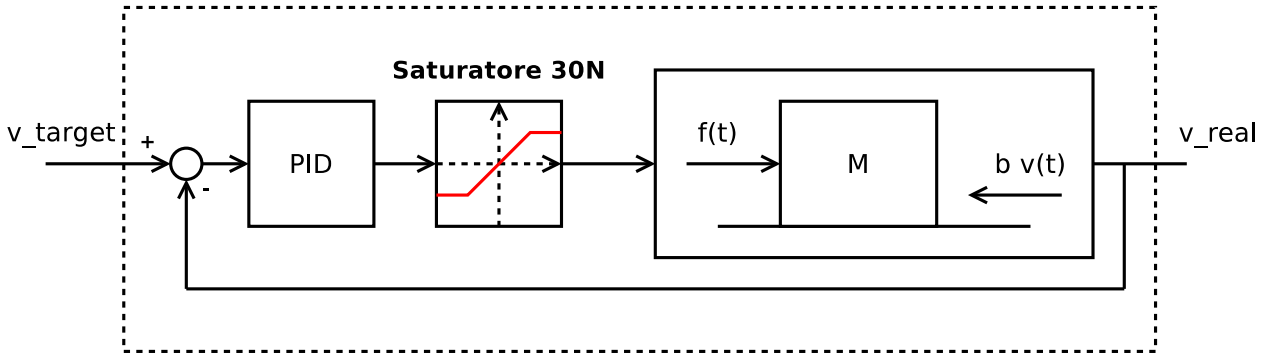
Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



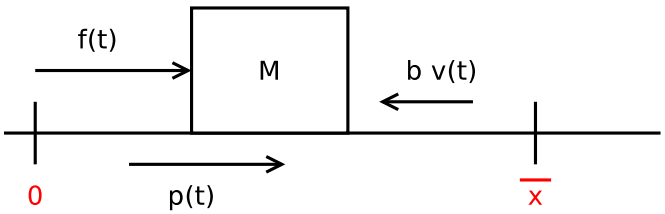
Programmazione Sistemi Robotici

Controllo in velocità di una massa



- Consideriamo il classico esempio di controllo in velocità di massa su piano con attrito
- Includiamo l'intero sistema in una "black box" (che chiamiamo $S + M$, *speed control+ mass*) che ha come input una **velocità target** e output la **velocità effettiva**
- Compito di tale sistema è assicurare che la velocità effettiva segua sempre la velocità target
- Ciò è assicurato dal PID presente dentro il blocco $S + M$

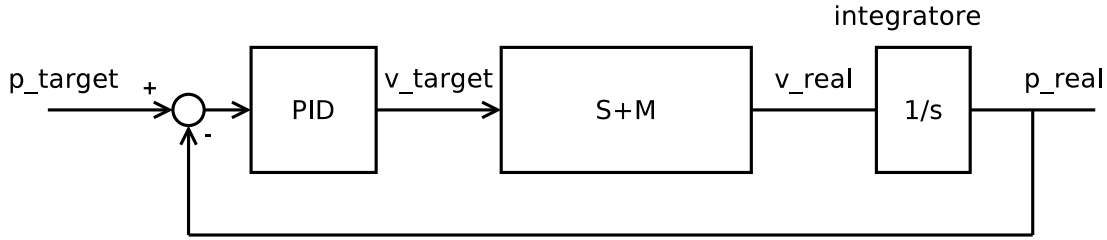
Controllo in posizione di una massa



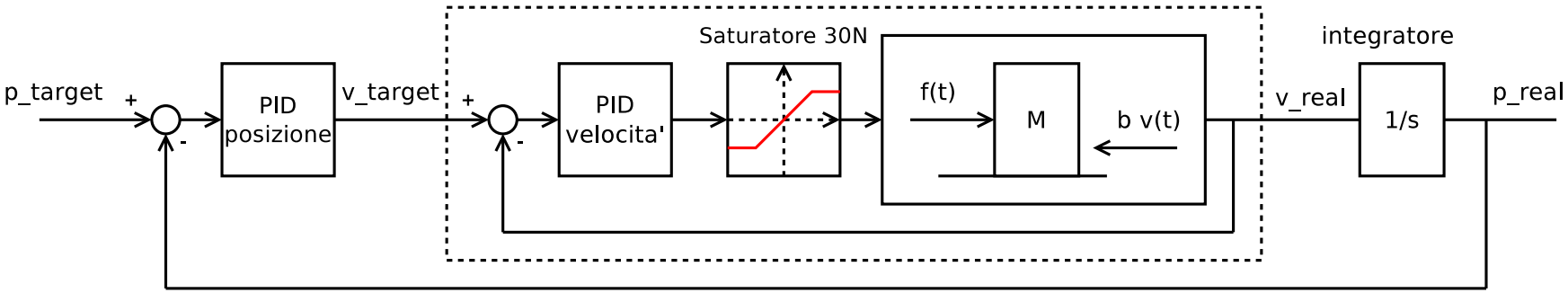
- Adesso supponiamo che desideriamo far sì che la nostra massa raggiunga una certa **posizione lineare** $\overline{x}$
- Avendo a disposizione il **controllo in velocità** possiamo:
  - Misurare la posizione corrente $p(t)$ (integrale della velocità)
  - Confrontarla con la nostra posizione target $\overline{x}$
  - Usare un **PID** che, sulla base dell'errore $e(t) = \overline{x} - p(t)$ produce la **velocità target** che la nostra massa deve avere

Controllo in posizione di una massa

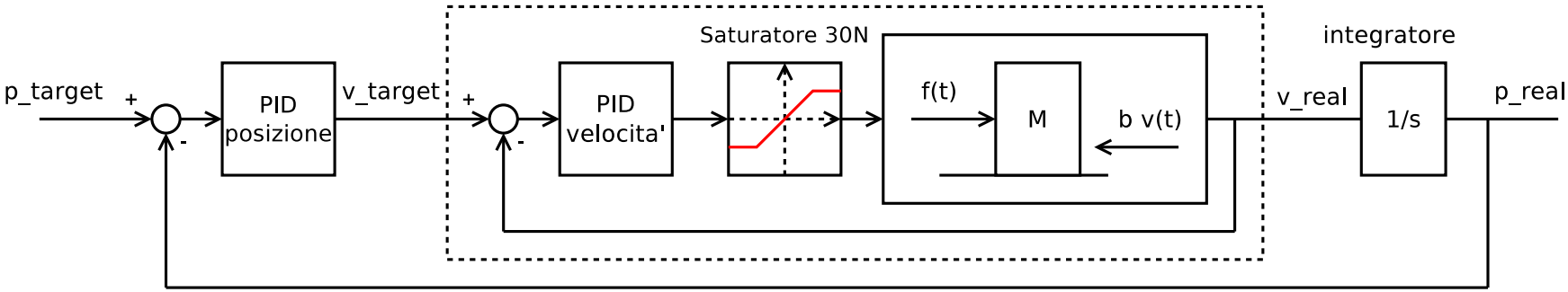
- Lo schema di controllo diventa il seguente:



- ed “esplodendo” il blocco $S + M$ abbiamo:

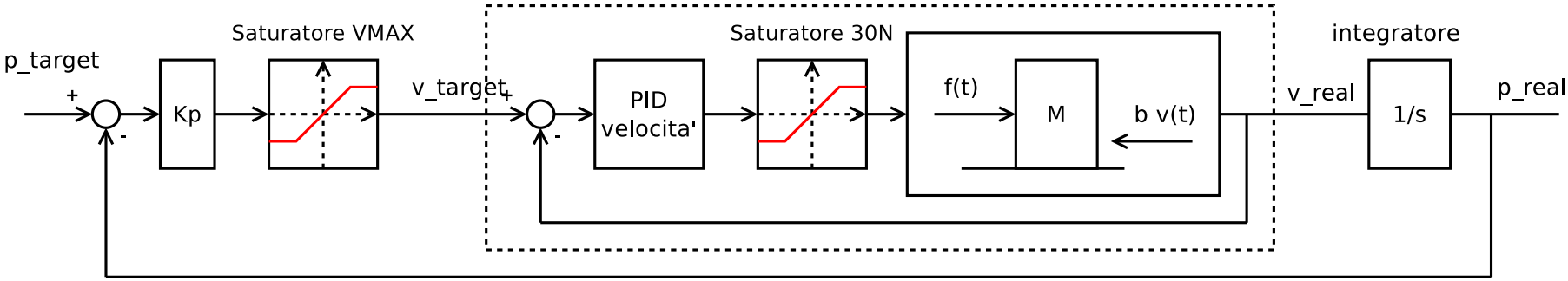


Controllore di posizione



- Il comportamento del controllore in posizione può essere immaginato secondo due opzioni:
  - un **PID** a tutti gli effetti;
  - uno specifico algoritmo che genera un **profilo di velocità**.

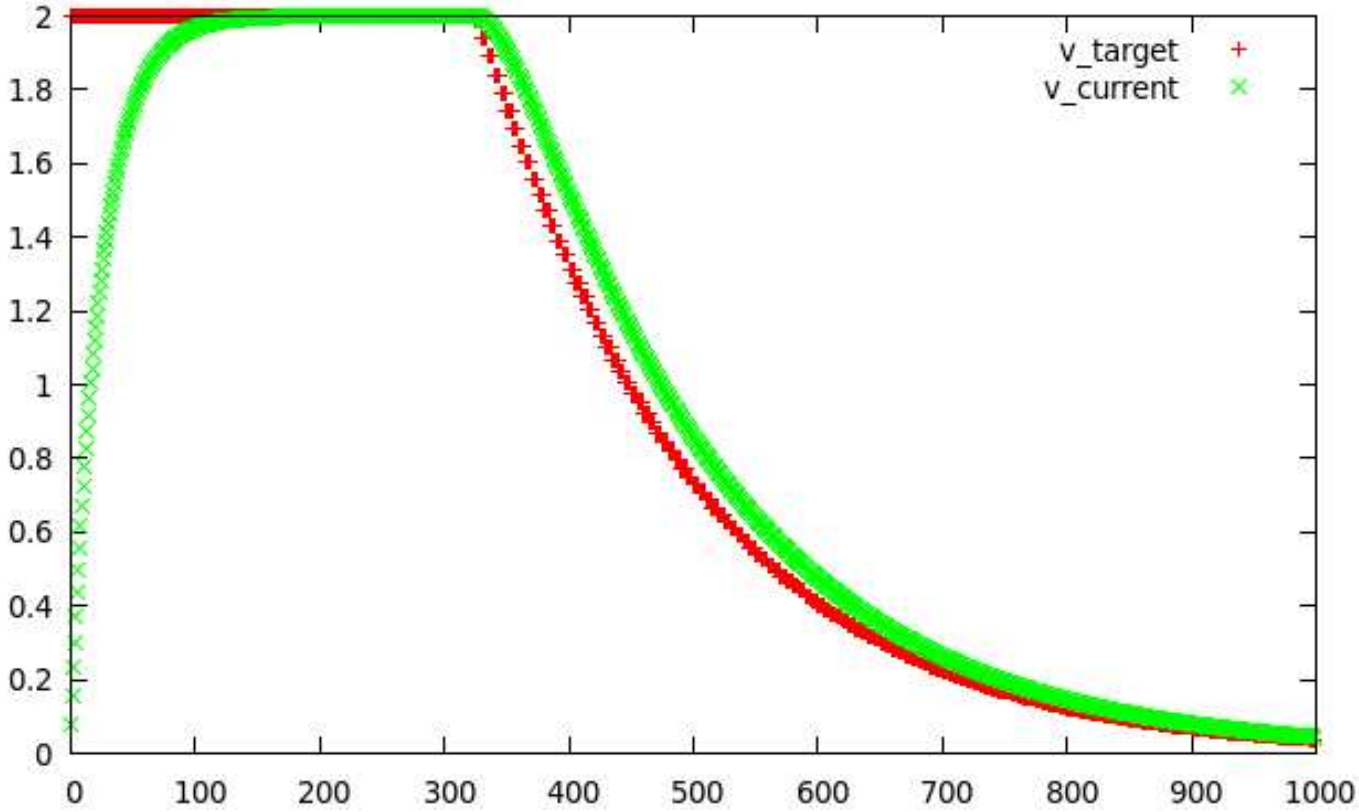
Controllo di posizione tramite PID



- Possiamo immaginare il seguente comportamento:
  - Se siamo sul target ($errore = 0$) la nostra velocità sarà **nulla**
  - Se siamo lontani dal target ($errore \neq 0$) la nostra velocità sarà **tanto più elevata quanto più lontani** siamo dal target, fino ad un **limite massimo**
- questa tipologia di controllo implica un semplice **controllore proporzionale con saturazione**
- La **saturazione** è data dalla velocità massima *VMAX*
- Il valore di K_p determina "quanto velocemente" vogliamo avvicinarci al target

Andamento della Velocità

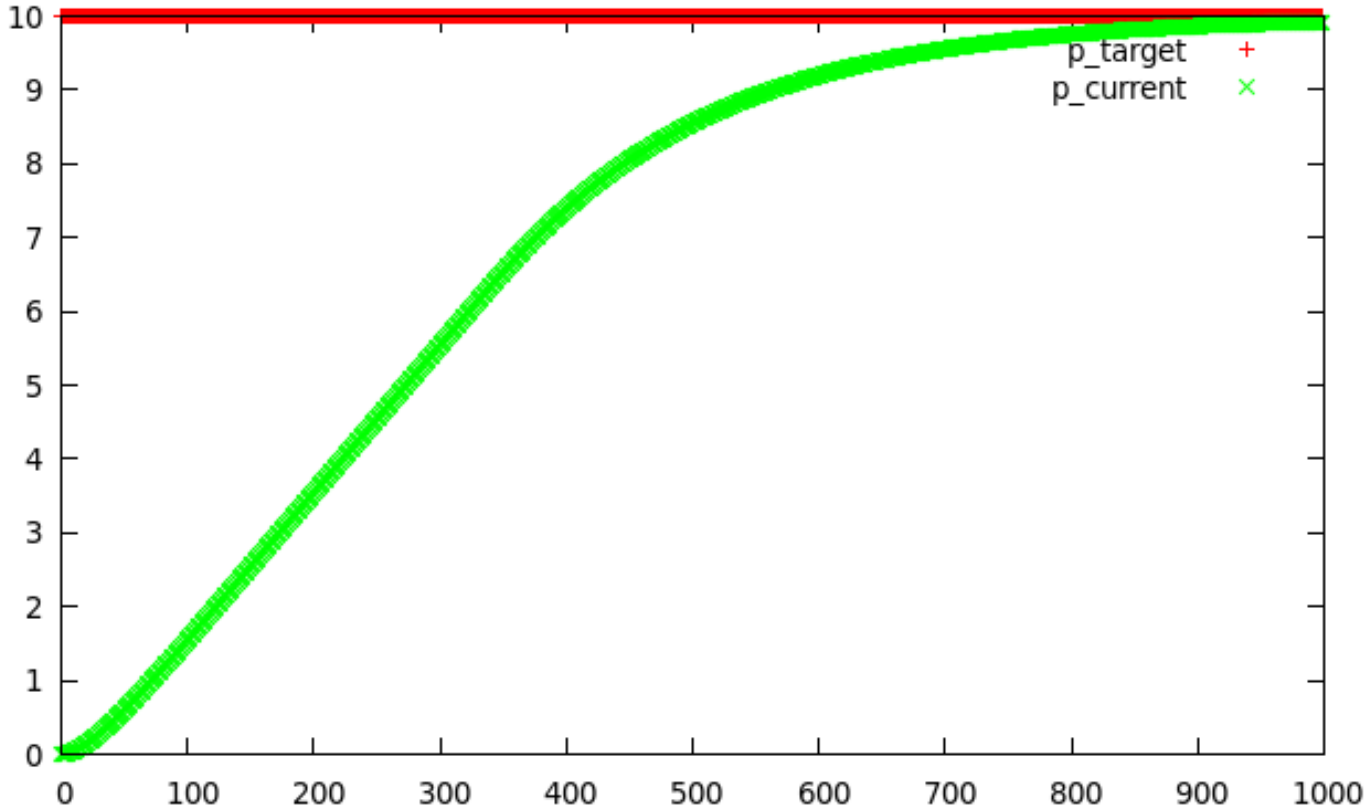
$$K_p = 0.5$$



- La velocità è massima durante la “marcia” e scende gradualmente man mano che ci si avvicina alla posizione desiderata

Andamento della Posizione

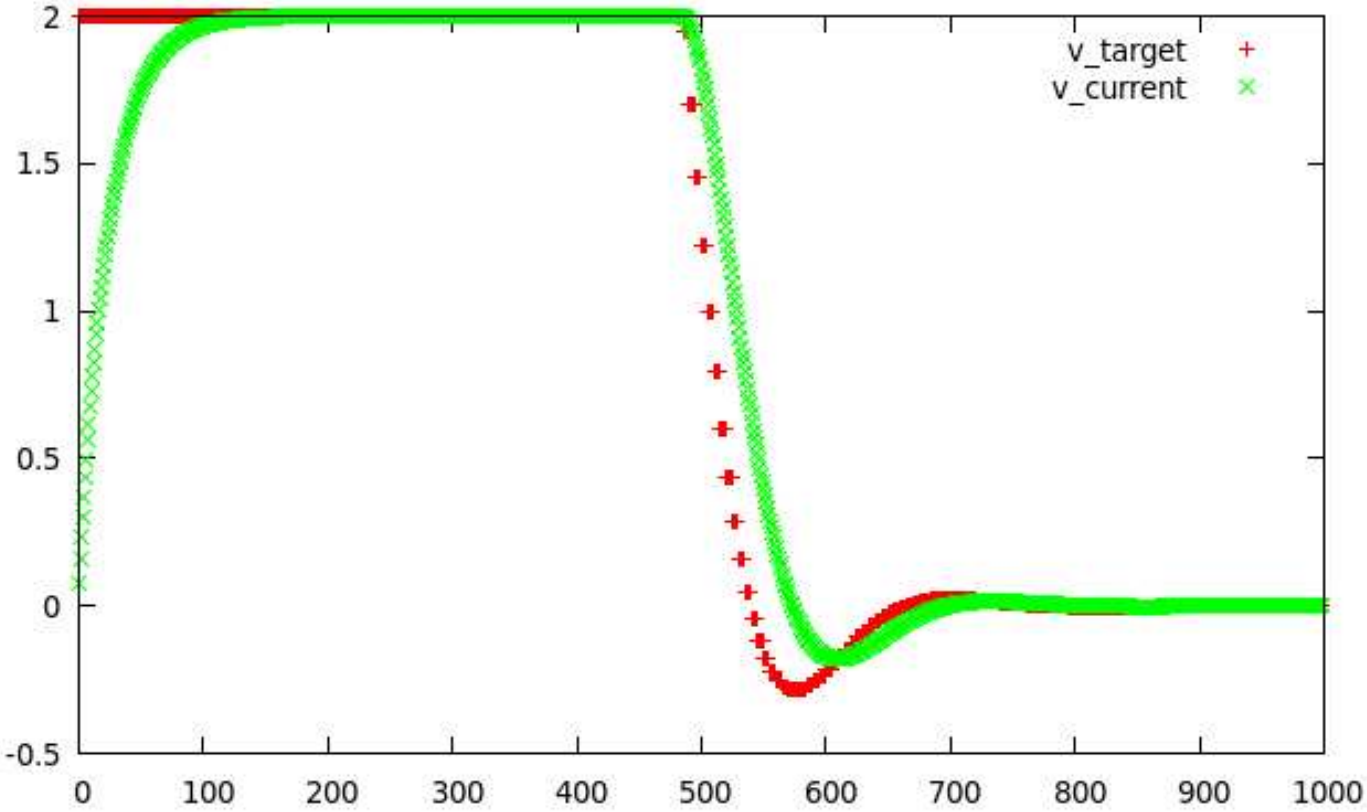
$$K_p = 0.5$$



- La massa si avvicina gradualmente alla posizione target (10 m) e poi si ferma lì

Andamento della Velocità

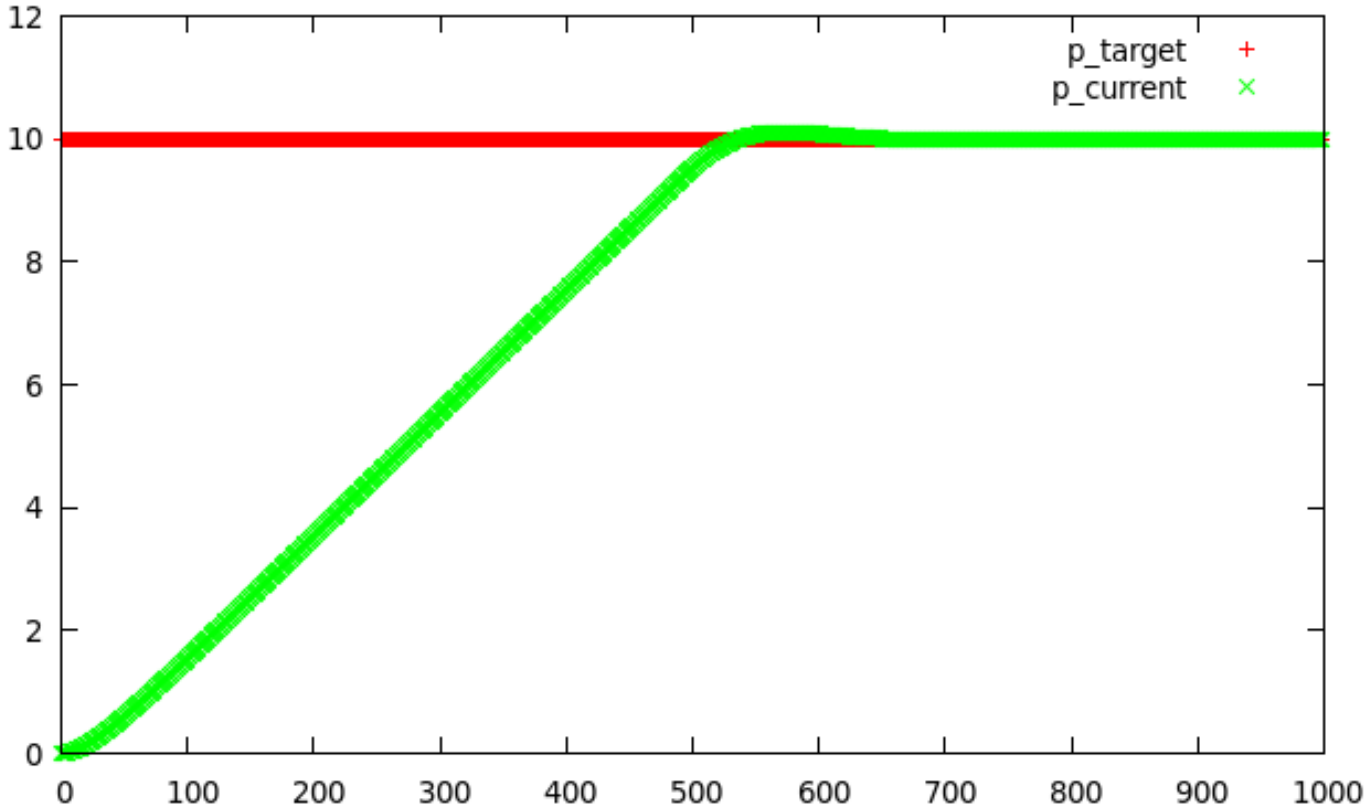
$$K_p = 2.5$$



- Se il valore di K_p è troppo elevato, l'effetto che si ottiene è un **superamento** del punto target, per poi "tornare indietro"

Andamento della Posizione

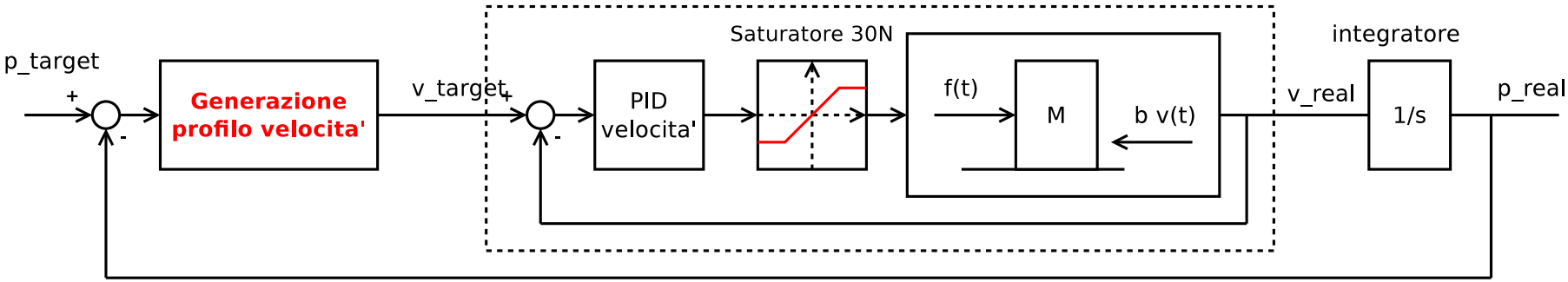
$$K_p = 2.5$$



- Nell'andamento della posizione compare dunque una **sovraelongazione**

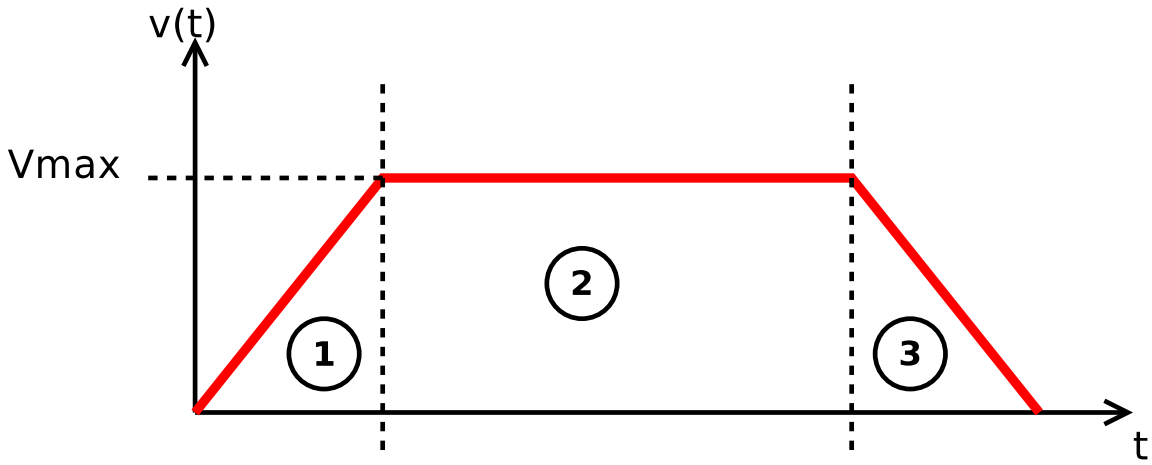
Implementazione del controllo in posizione tramite profilo di velocità

Controllore di posizione



- La seconda tecnica di controllo prevede l'uso di un **algoritmo software** che genera il **profilo di velocità** da seguire per poter raggiungere la posizione target.

Profilo di Velocità



- I tipici profili di velocità dei sistemi meccanici prevedono tre fasi:
  1. Una prima fase di **accelerazione**, parametro $a_1 = cost$
  2. Una fase di *crociera* a **velocità costante**, $a_2 = 0$, $v = V_{max}$
  3. Una fase di **decelerazione**, con parametro $a_3 = cost$
- L'obiettivo è far sì che, al termine della fase di decelerazione, la massa da controllare si trovi effettivamente alla **posizione target**
- Il controllore è totalmente determinato dai **parametri** a_1, V_{max} e a_3, più il tempo di campionamento T_{camp}.

Profilo di Velocità e Moto uniformemente accelerato

Per determinare l'algoritmo del controllore, useremo le formule del **moto uniformemente accelerato** che qui ricordiamo:

$$\begin{aligned} a &= cost \\ v(t) &= v(0) + at \\ x(t) &= x(0) + v(0)t + \frac{1}{2}at^2 \end{aligned}$$

Prima definizione della classe

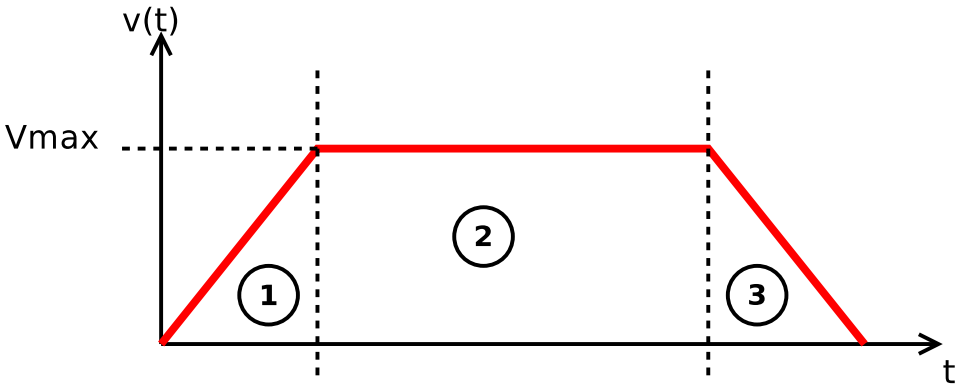
Implementeremo il controllore non più usando la classe `Pid`, ma una una classe ad-hoc che chiameremo **`PositionController`**:

```
class PositionController {
 public:
    PositionController(float accel, float v_max, float decel, float dt);
    float evaluate(float target, float current_pos, float current_speed);
 protected:
    float m_accel; // accelerazione
    float m_vmax;  // vel max
    float m_decel; // decelerazione
    float m_dt;    // tempo di campionamento

    float m_next_speed; // velocity

    float m_accel_step;
    float m_decel_distance;
};
```

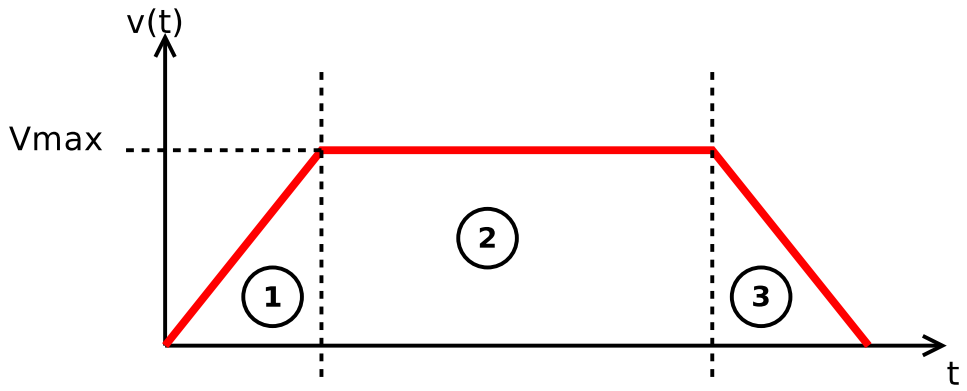
Implementazione delle fasi



- Le fasi 1 e 2 sono abbastanza semplici: si tratta di incrementare la velocità, ad ogni step di simulazione, del valore $a_1 T_{camp}$, fino alla saturazione V_{max}:

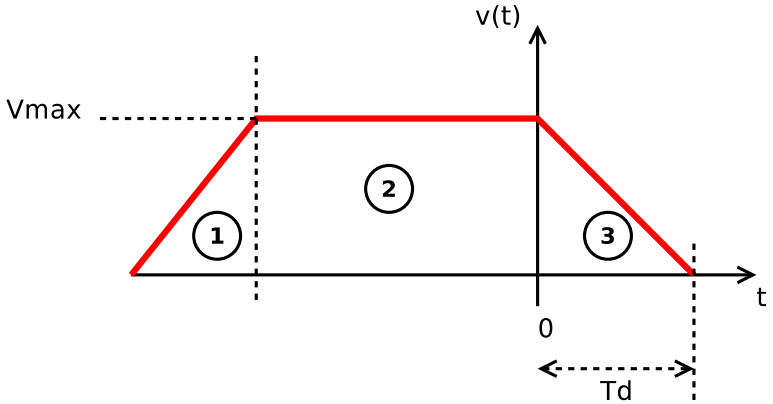
```
float PositionController::evaluate(float target, float current_pos, float current_speed)
{
    ....
        // Fase 1, fase 2
        if (m_next_speed < m_vmax) {
            m_next_speed += m_accel * m_dt; // incremento
            if (m_next_speed > m_vmax) m_next_speed = m_vmax;
        }
     ....
     return m_next_speed;
}
```

Implementazione delle fasi



- Per implemntare la fase 3 è necessario identificare il punto in cui la fase 3 stessa **deve cominiciare**
- Poichè possiamo conoscere la **distanza** (errore) tra la posizione corrente $x(t)$ e la posizione target $d = X_{target} - x(t)$, possiamo usare questa informazione per calcolare **a quale distanza deve iniziare la decelerazione**
- Determiniamo quindi la **distanza di decelerazione**

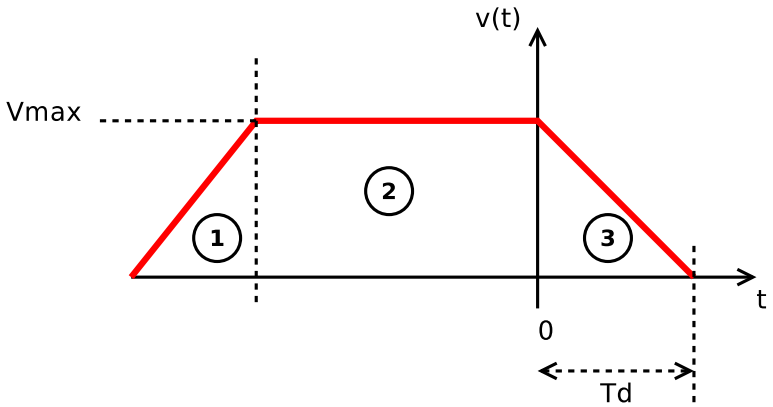
Implementazione della fase 3



- Operiamo una **traslazione temporale** e consideriamo il tempo 0 l'inizio della fase 3
- Usando la formula del moto uniformemente accelerato, calcoliamo il **tempo necessario per arrivare a velocità zero** (tempo di decelerazione) T_D:

$$\begin{aligned} v(T_D) &= v(0) + a_3 T_D \\ 0 &= V_{max} + a_3 T_D \\ T_D &= -\frac{V_{max}}{a_3} \end{aligned}$$

Implementazione della fase 3



$$
\begin{aligned}
v(T_D) &= v(0) + a_3 T_D \\
0 &= V_{max} + a_3 T_D \\
T_D &= -\frac{V_{max}}{a_3}
\end{aligned}
$$

- Il segno “meno” è normale perchè, essendo a_3 **negativo** (è una decelerazione), accade che T_D risulti alla fine **positivo**.

Implementazione della fase 3

- Determiniamo a questo punto la **distanza di decelerazione** $X_D = x(T_D)$:

$$
\begin{aligned}
x(T_D) &= x(0) + v(0)T_D + \frac{1}{2}a_3 T_D^2 \\
x(T_D) &= 0 + V_{max}T_D + \frac{1}{2}a_3 T_D^2 \\
x(T_D) &= V_{max}(-\frac{V_{max}}{a_3}) + \frac{1}{2}a_3(-\frac{V_{max}}{a_3})^2 \\
x(T_D) &= -\frac{V_{max}^2}{a_3} + \frac{1}{2}a_3\frac{V_{max}^2}{a_3^2} \\
x(T_D) &= -\frac{V_{max}^2}{a_3} + \frac{1}{2}\frac{V_{max}^2}{a_3} \\
X_D &= -\frac{V_{max}^2}{2a_3}
\end{aligned}
$$

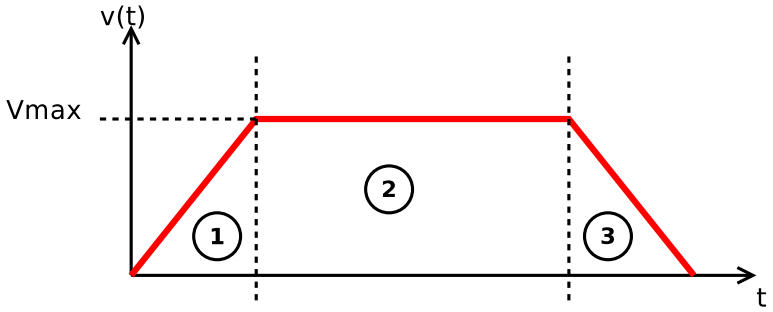
Implementazione delle fasi

- Se la distanza dal target è inferiore a X_D allora siamo nella fase di decelerazione. Il codice diventa:

```
PositionController::PositionController(float accel, float v_max, float decel, float dt)
    : m_accel(accel), m_vmax(v_max), m_decel(decel), m_dt(dt), m_next_speed(0)
{
    m_accel_step = m_accel * m_dt;
    m_decel_distance = (m_vmax * v_max) / (2 * m_decel); // si considera decel positiva
}

float PositionController::evaluate(float target, float current_pos, float current_speed)
{
    float distance = target - current_pos;
    if (distance < m_decel_distance) {
        // fase 3
        ....
    }
    else {
        // Fase 1, fase 2
        if (m_next_speed < m_vmax) {
            m_next_speed += m_accel * m_dt; // incremento
            if (m_next_speed > m_vmax) m_next_speed = m_vmax;
        }
    }
    return m_next_speed;
}
```

Implementazione della fase 3



- Durante la fase 3, il controllore deve fornire la velocità alla quale la massa deve andare
- Poichè conosciamo la distanza tra posizione target e posizione corrente usiamo questa informazione per determinare la velocità zero
- Dobbiamo quindi trovare una funzione del tipo

$$v(t) = f(e(t)), e(t) = X_{target} - x(t)$$

- A tale scopo usiamo ancora una volta le equazioni della cinematica applicate alla **sola fase di decelerazione**

Implementazione della fase 3

- Consideriamo il tempo 0 l'inizio della fase della decelerazione, quindi $v(0) = V_{max}$ e $x(0) = 0$, abbiamo:

$$\begin{aligned} v(t) &= V_{max} + a_3 t \\ x(t) &= V_{max} t + \frac{1}{2} a_3 t^2 \end{aligned}$$

- Poniamo, per brevità, $x = x(t)$ e $v = v(t)$ e calcoliamo il tempo dalla prima equazione:

$$t = \frac{v - V_{max}}{a_3}$$

- e sostituiamolo nella seconda:

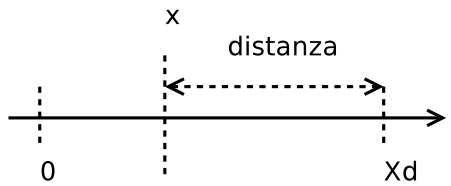
Implementazione della fase 3

$$
\begin{aligned}
x &= V_{max}\frac{v - V_{max}}{a_3} + \frac{1}{2}a_3\left(\frac{v - V_{max}}{a_3}\right)^2 \\
x &= \frac{V_{max}v - V_{max}^2}{a_3} + \frac{v^2 - 2vV_{max} + V_{max}^2}{2a_3} \\
2a_3x &= 2V_{max}v - 2V_{max}^2 + v^2 - 2V_{max}v + V_{max}^2 \\
2a_3x &= -2V_{max}^2 + v^2 + V_{max}^2 \\
2a_3x &= v^2 - V_{max}^2 \\
v &= \sqrt{V_{max}^2 + 2a_3x}
\end{aligned}
$$

- x è la distanza percorsa a partire **dall'inizio della fase di decelerazione**
- Tuttavia noi possediamo la **distanza dal target**, quindi ...

Implementazione della fase 3

$$v = \sqrt{V_{max}^2 + 2a_3 x}$$



- x è pari alla **distanza di decelerazione** X_D meno la **distanza dal target**:

$$x = X_D - (target\_pos - current\_pos)$$

- pertanto ...

$$v = \sqrt{V_{max}^2 + 2a_3(X_D - (target\_pos - current\_pos))}$$

- se a_3 la si considera in valore assoluto abbiamo:

$$v = \sqrt{V_{max}^2 - 2a_3(X_D - (target\_pos - current\_pos))}$$

Implementazione delle fasi

- Il codice che include anche la fase 3 diventa:

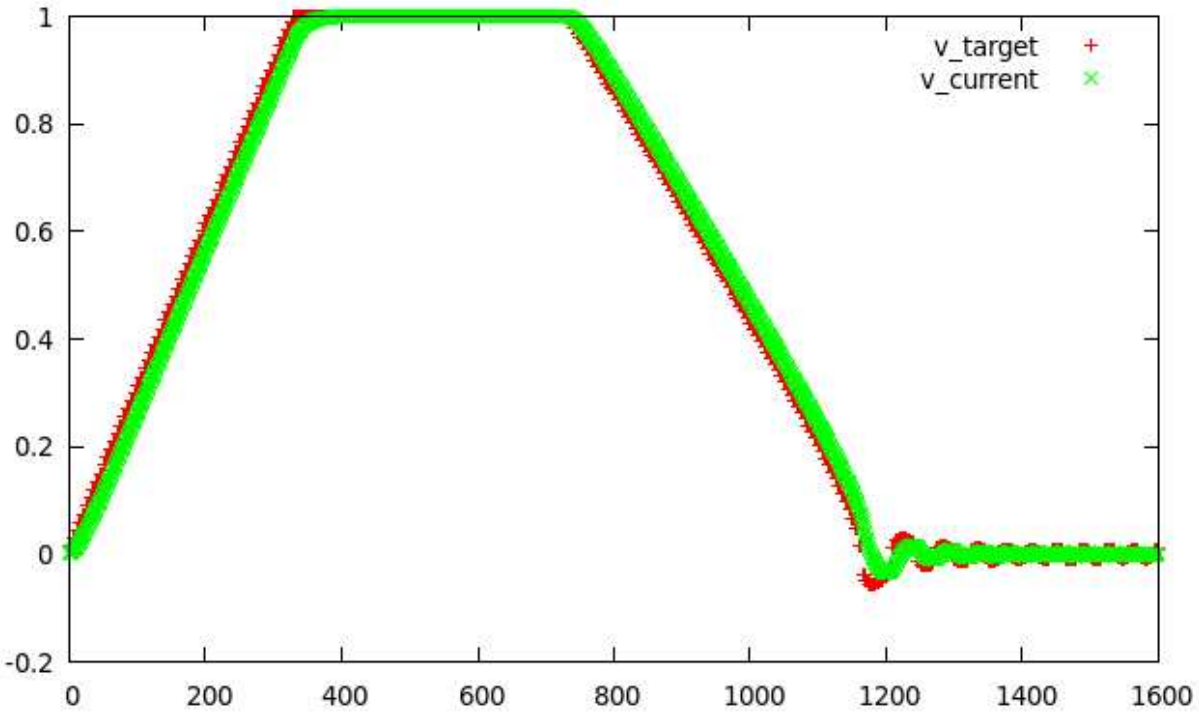
```
PositionController::PositionController(float accel, float v_max, float decel, float dt)
    : m_accel(accel), m_vmax(v_max), m_decel(decel), m_dt(dt), m_next_speed(0)
{
    m_accel_step = m_accel * m_dt;
    m_decel_distance = (m_vmax * v_max) / (2 * m_decel); // si considera decel positiva
}

float PositionController::evaluate(float target, float current_pos, float current_speed)
{
    float distance = target - current_pos;
    if (distance < m_decel_distance) {
        // fase 3
        float expected_speed = sqrt(m_vmax * m_vmax -
                                    2 * m_decel * (m_decel_distance - distance));
        m_next_speed = expected_speed;
    }
    else {
        // Fase 1, fase 2
        if (m_next_speed < m_vmax) {
            m_next_speed += m_accel * m_dt; // incremento
            if (m_next_speed > m_vmax) m_next_speed = m_vmax;
        }
    }
    return m_next_speed;
}
```

Andamento della velocità

L'andamento della velocità, con questo tipo di controllo, ed una distanza di $8m$, è

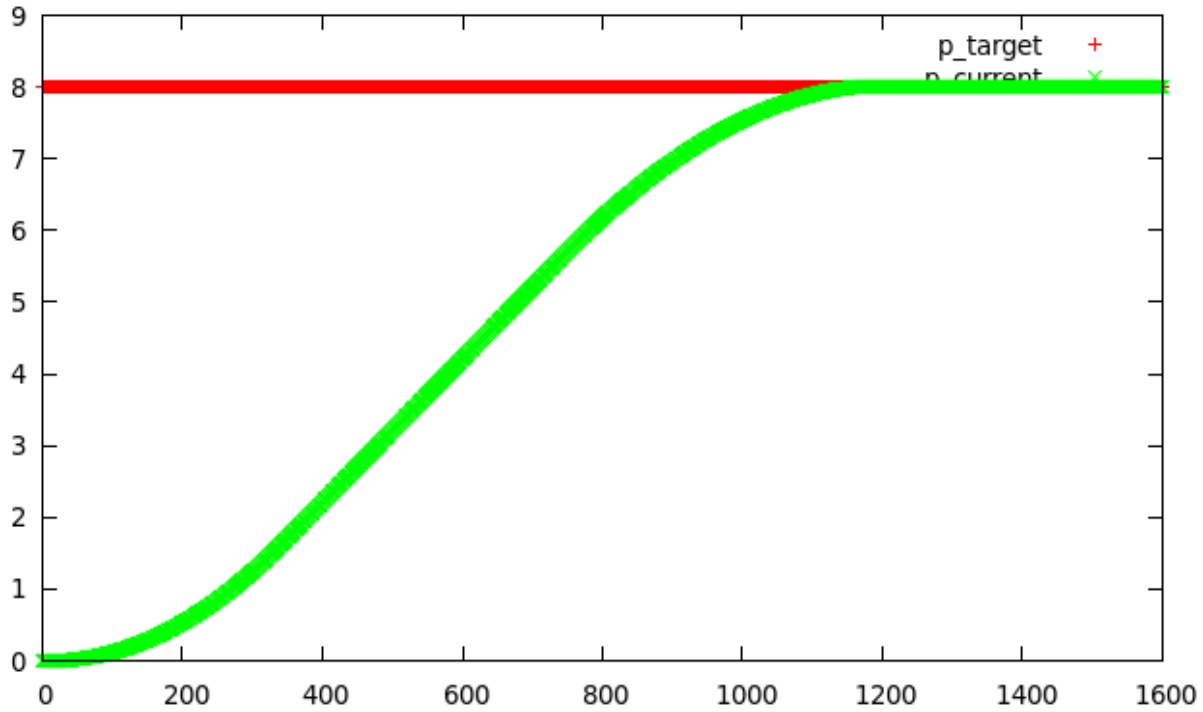
$$a_1 = 0.3m/s^2, V_{max} = 2m/s, a_3 = 0.2m/s^2$$



Andamento della distanza

L'andamento della distanza, con questo tipo di controllo, ed una distanza da percorrere di $8m$, è

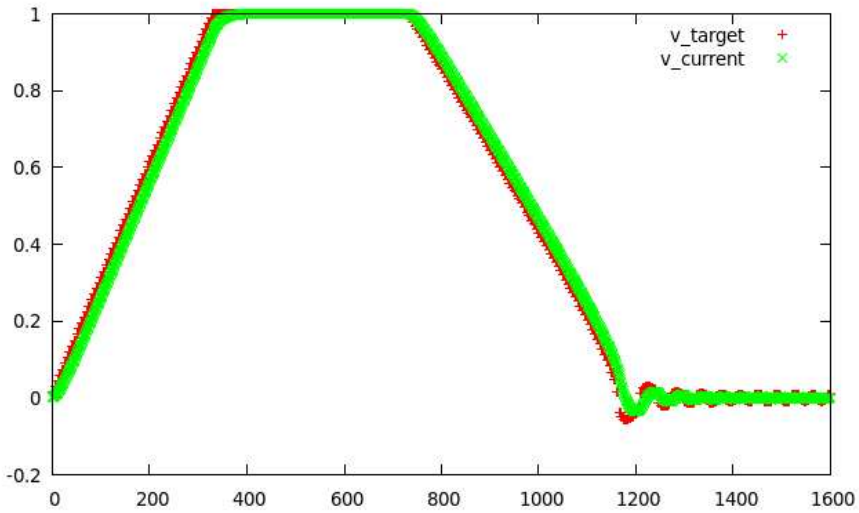
$$a_1 = 0.3m/s^2, V_{max} = 2m/s, a_3 = 0.2m/s^2$$



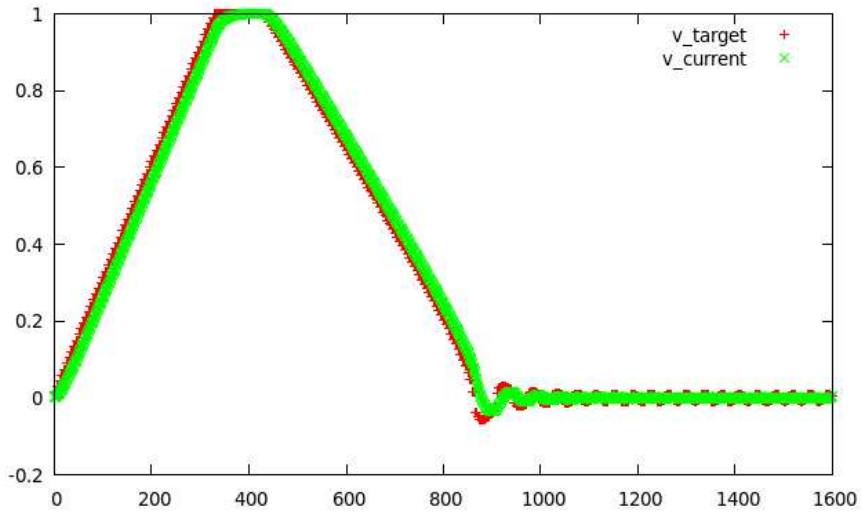
Fasi e Durata

- La durata delle fasi di accelerazione e decelerazione è funzione diretta delle costanti di accelerazione e decelerazione
- Una volta impostata a_1 e a_3, tali durate sono **sempre uguali**
- Al variare della distanza da percorrere, varierà pertanto la durata della fase a velocità costante

target $= 8m$



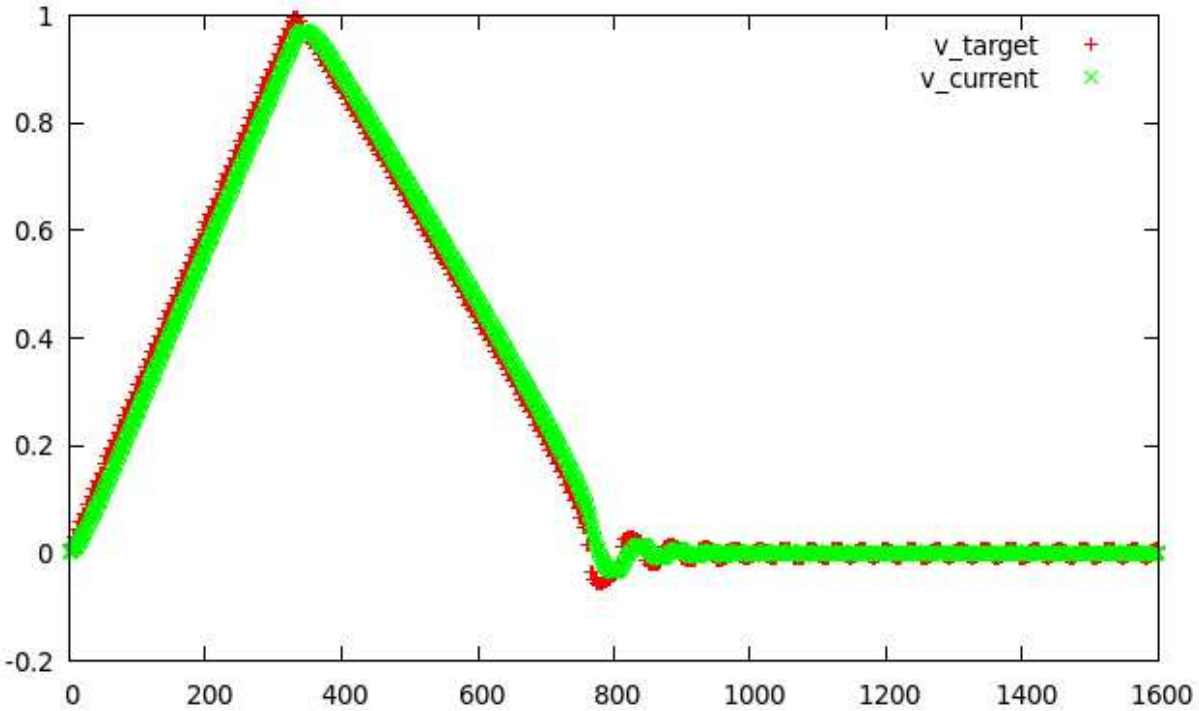
target $= 5m$



Fasi e Durata

- La fase a velocità costante potrebbe addirittura sparire nel caso la distanza sia particolarmente “corta”
- In tal caso il trapezio degenera in un **triangolo**

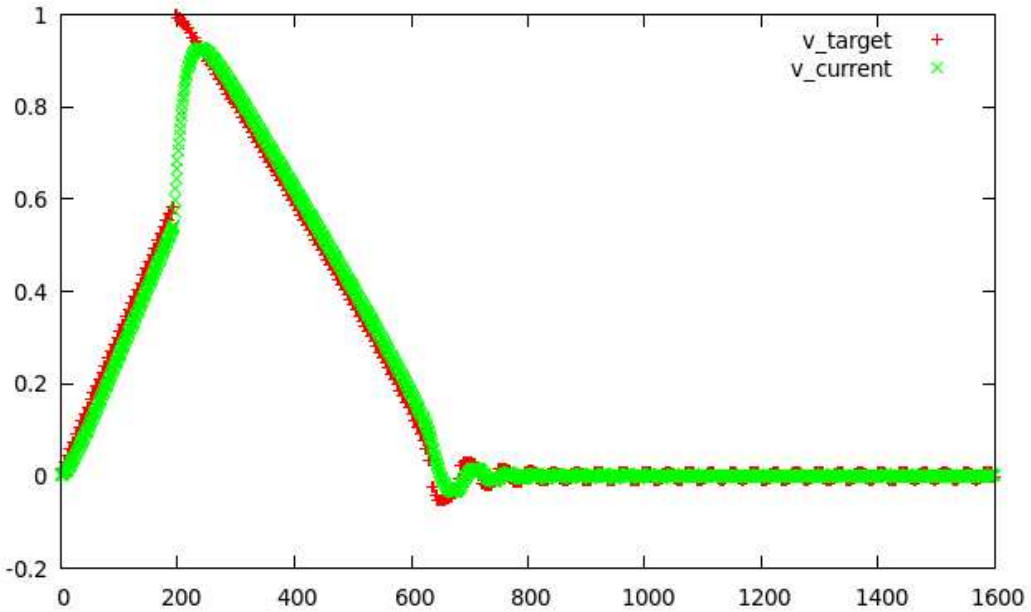
$$target = 4m$$



Degenerazione del trapezio e discontinuità

- Tuttavia se distanza è ulteriormente “corta”, le fasi 1 e 3 si *accavallano*
- In tal caso, l’algortimo descritto presenta un fastidioso punto di discontinuità e il comportamento diventa il seguente:

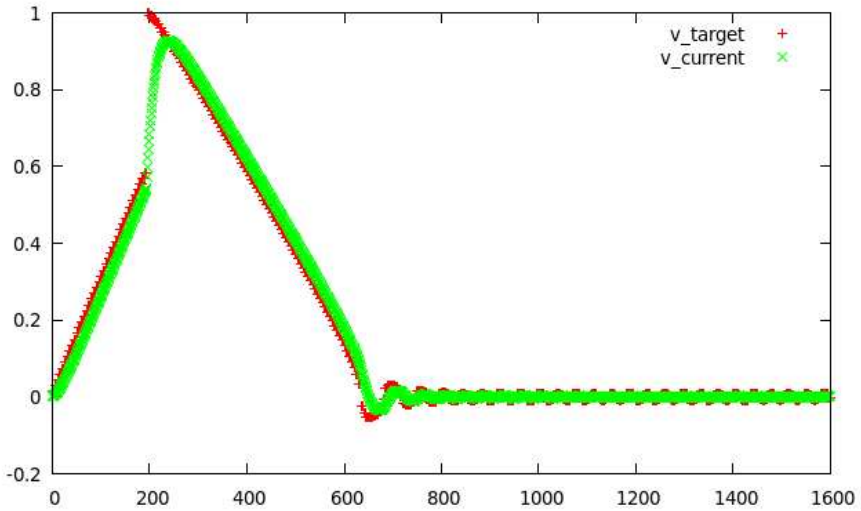
$$target = 3m$$



- Il problema è dato dal fatto che siamo già entrati nella **distanza di decelerazione** senza aver **finito la fase di accelerazione**
- La soluzione consiste nel **prolungare la fase di accelerazione** fin quando il segmento crescente non **incontra** il tratto di decelerazione

Soluzione del problema della discontinuità

$target = 3m$



- La fase 3 deve essere caratterizzata da una **invariabilità**: **la velocità da raggiungere deve essere minore della velocità corrente**, altrimenti non ha senso parlare di *decelerazione*
- Nel caso in figura accade proprio il contrario: quando inizia la fase di decelerazione, la velocità calcolata **è maggiore** della velocità corrente
- Possiamo pertanto sfruttare questa proprietà per "far continuare" l'accelerazione fino all'incontro con il segmento di decelerazione

Soluzione del problema della discontinuità

- Aggiungiamo questo caso particolare...

```
float PositionController::evaluate(float target, float current_pos, float current_speed)
{
    float distance = target - current_pos;
    if (distance < m_decel_distance) {
        // fase 3
        float expected_speed = sqrt(m_vmax * m_vmax -
                                    2 * m_decel * (m_decel_distance - distance));
        if (expected_speed > current_speed) {
            // la vel da raggiungere risulta maggiore della vel corrente
            // siamo ancora nella fase di accelerazione, pertanto
            // continuiamo ad accelerare
            m_next_speed += m_accel * m_dt; // incremento
            // tuttavia verifichiamo di non aver raggiunto la vel calcolata
            if (m_next_speed > expected_speed) m_next_speed = expected_speed;
            // o di superare la vel massima
            if (m_next_speed > m_vmax) m_next_speed = m_vmax;
        }
        else // qui siamo regolarmente nella fase di decelerazione
            m_next_speed = expected_speed;
    }
    else {
        // Fase 1, fase 2
        if (m_next_speed < m_vmax) {
            m_next_speed += m_accel * m_dt; // incremento
            if (m_next_speed > m_vmax) m_next_speed = m_vmax;
        }
    }
    return m_next_speed;
}
```

Implementazione finale

- A questo punto il codice va generalizzato tenendo in considerazione distanze **positive** e distanze **negative**
- Il segno della distanza implica un segno analogo nella velocità risultante
- Pertanto, mettiamo da parte il segno, lavoriamo con i valori assoluti, e poi consideriamo il segno al momento di restituire la velocità effettiva

```
float PositionController::evaluate(float target, float current_pos, float current_speed)
{
    float distance = target - current_pos;
    float s;

    if (distance < 0) {
        s = -1;
        distance = -distance;
    }
    else
        s = 1;
    ....
}
```

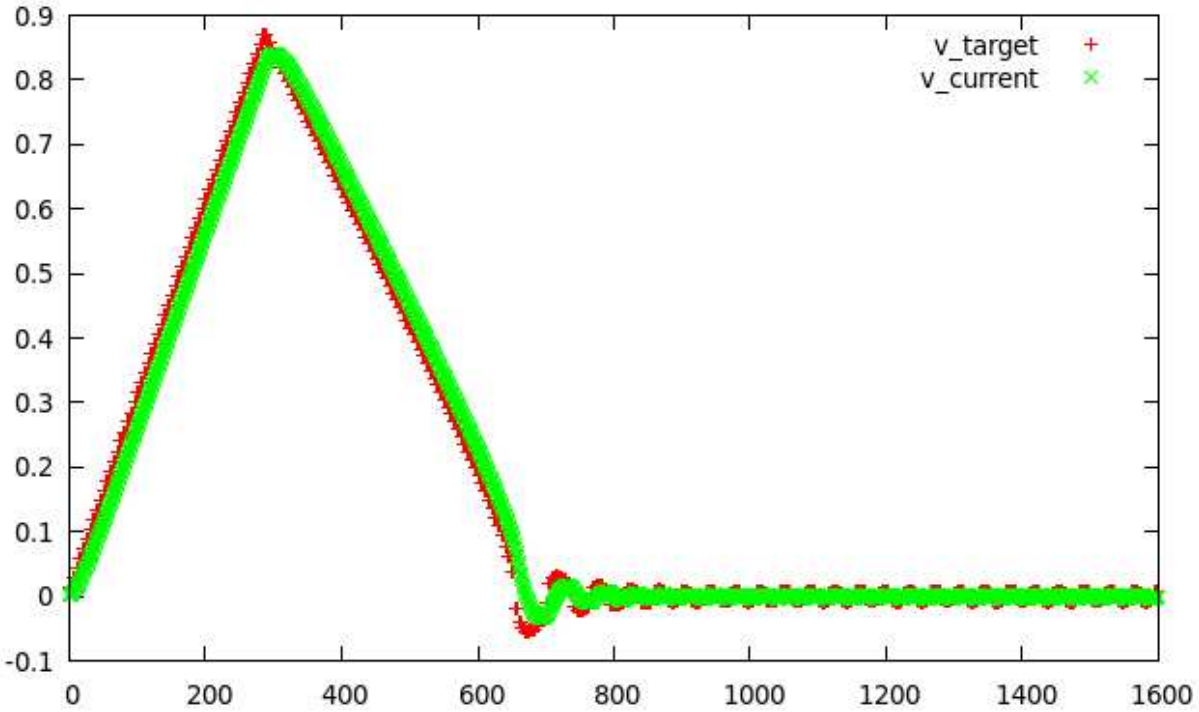
Implementazione finale

```
float PositionController::evaluate(float target, float current_pos, float current_speed)
{
    ....
    if (distance < m_decel_distance) {
        // fase 3
        float expected_speed = sqrt(m_vmax * m_vmax -
                                    2 * m_decel * (m_decel_distance - distance));
        if (expected_speed > fabs(current_speed)) {
            // la vel da raggiungere risulta MAGGIORE della vel corrente
            // questo vuol dire che siamo ancora nella fase di accelerazione
            // pertanto acceleriamo
            m_next_speed += m_accel_step;
            if (m_next_speed > expected_speed) m_next_speed = expected_speed;
            if (m_next_speed > m_vmax) m_next_speed = m_vmax;
        }
        else {
            m_next_speed = expected_speed;
        }
    }
    else {
        // Fase 1, fase 2
        if (m_next_speed < m_vmax) {
            m_next_speed += m_accel_step;
            if (m_next_speed > m_vmax) m_next_speed = m_vmax;
        }
    }
    return s*m_next_speed;
}
```

Test con distanza corta

- A questo punto, anche sulle distanze “corte”, l’algoritmo funziona correttamente:

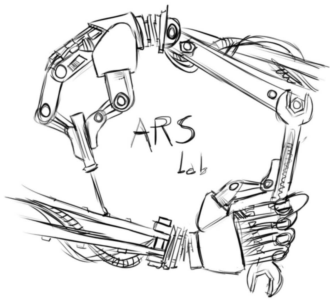
$$target = 3m$$



Controllo PID in posizione

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



Programmazione Sistemi Robotici