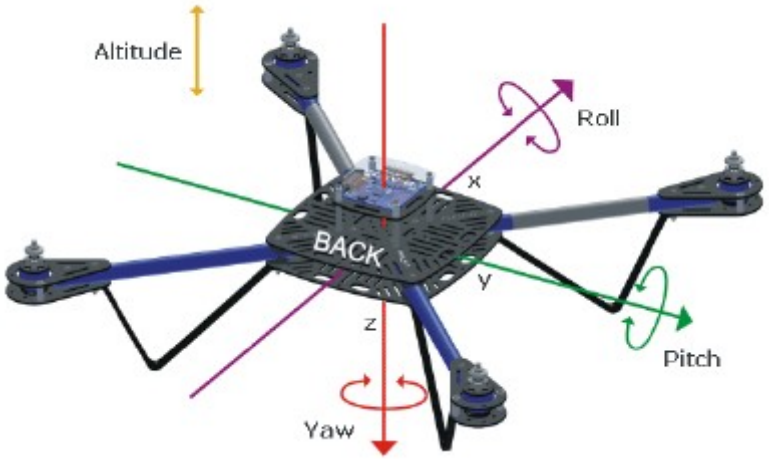
Controllo di un Multirotore

(a.k.a. “Drone”)

Scandura
Salvatore
1000012410



Un Multirotore, comunemente detto anche Drone, è un velivolo con capacità di VTOL (Vertical Take-off and Landing), completamente controllato tramite software.

L'unico aspetto critico è legato alla parte meccanica rappresentata esclusivamente dalle eliche orizzontali, in genere in numero pari e poste in modo da avere un frame simmetrico e bilanciato.

Caratteristica fondamentale è la determinazione dell'assetto del drone tramite un sistema di riferimento che sfrutta gli angoli di Eulero:

- roll, descrive l'inclinazione rispetto l'asse x;
- pitch, descrive l'inclinazione rispetto l'asse y;
- yaw, descrive l'inclinazione rispetto l'asse z;
- X, Y e Z, rappresentano invece le coordinate geografiche.

In questa presentazione viene trattato il modello semplificato bidimensionale, dove la posa del multirotore è individuata attraverso la terna {X, Z, pitch}.

Per modellare il moto del velivolo viene utilizzata la seconda legge di Newton rotazionale, da cui distinguiamo i seguenti aspetti dinamici:

- dinamica rotazionale;
- traslazione orizzontale;
- traslazione verticale.

Dinamica Rotazionale

Viene modellata attraverso la seguente legge: $F_2 L - F_1 L = I\ddot{\theta}$

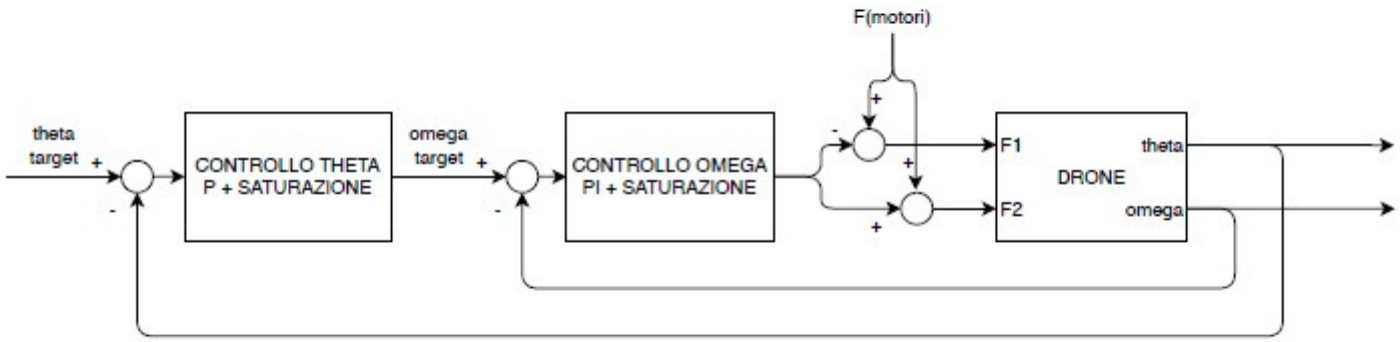
da cui si ottiene il seguente sistema: $\begin{cases} \dot{\theta} = \omega \\ \dot{\omega} = \frac{L}{I}(F_2 - F_1) \end{cases}$

Procedendo con la discretizzazione di tale sistema, si ottiene infine la legge di aggiornamento dello stato per la dinamica rotazionale:

$$\begin{cases} \theta = \theta + \Delta T * \omega \\ \omega = \omega + \Delta T * L * (F_2 - F_1) / I \end{cases}$$

Dinamica Rotazionale

Schema a blocchi per il controllo della rotazione:



Dallo schema a blocchi precedente, si passa a modellare i vari sistemi, per poi giungere all'implementazione seguendo semplicemente il flusso dei dati in esame.

In questo caso è stata implementata la classe *Angle_Controller* che, oltre a configurare i parametri necessari ai vari sottosistemi, ovvero il controllore Proporzionale e quello Proporzionale-Integrale, riceve gli input dal metodo *evaluate*, tra cui *theta_target* che rappresenta l'ingresso principale di questa elaborazione.

L'output di questo blocco rappresenta invece la spinta che successivamente verrà fornita ai motori del multirotore, modulata poi con l'uscita del blocco per il controllo della traslazione verticale.

```
from controllore_p import *
from controllore_pi import *

class Angle_Controller:
    def __init__(self, p_kp, p_sat, pi_kp, pi_ki, pi_sat):
        self.p_controller = Controllore_P(p_kp, p_sat)
        self.pi_controller = Controllore_PI(pi_kp, pi_ki, pi_sat)
    def evaluate(self, delta_t, theta_target, theta_current, omega_current):
        theta_error = theta_target - theta_current
        omega_target = self.p_controller.evaluate(theta_error)
        omega_error = omega_target - omega_current
        out = self.pi_controller.evaluate(delta_t, omega_error)

        return out
```

Traslazione Orizzontale

Viene modellata attraverso la seguente legge: $(F_1 + F_2)sin(-\theta) - bv_x = M\dot{v}_x$

da cui si ottiene il seguente sistema:

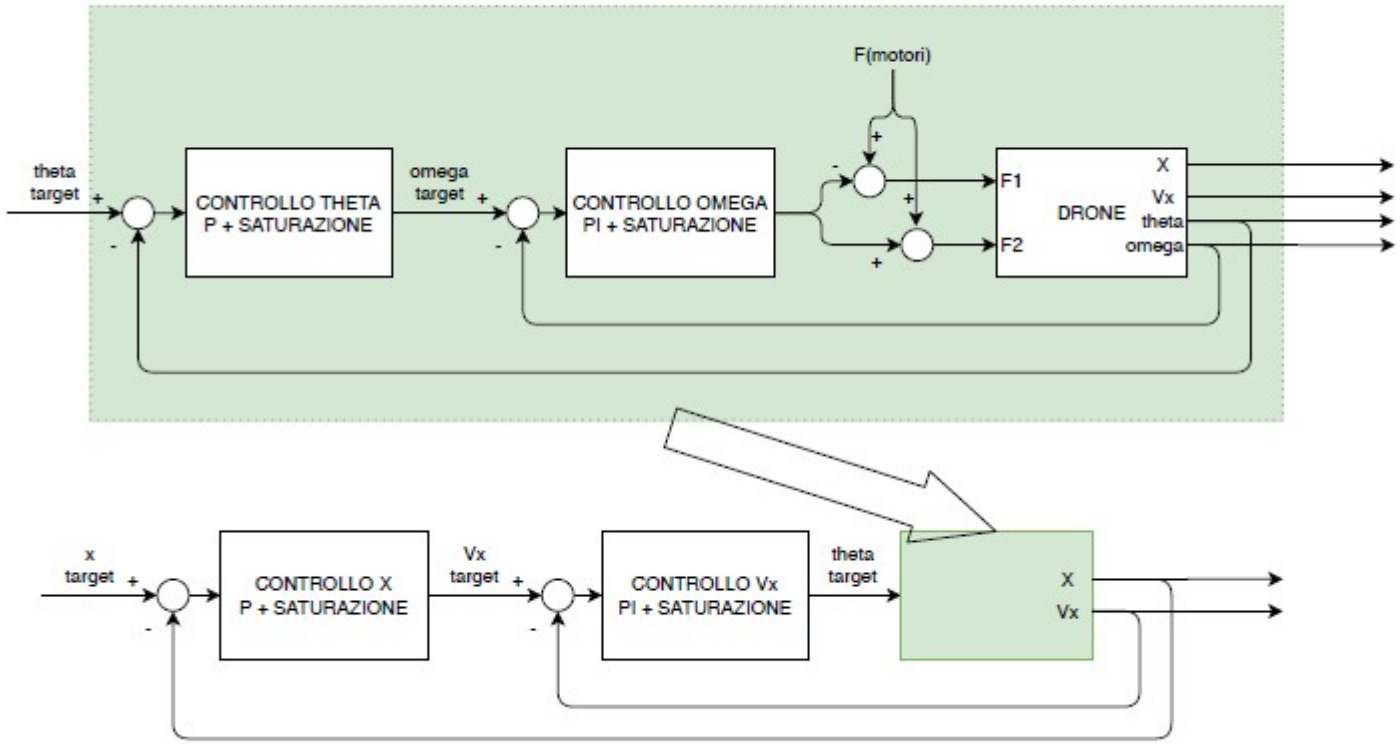
$$\begin{cases} \dot{x} = v_x \\ \dot{v}_x = -\frac{b}{M}v_x + \frac{F_1+F_2}{M}sin(-\theta) \end{cases}$$

Procedendo con la discretizzazione di tale sistema, si ottiene infine la legge di aggiornamento dello stato per la traslazione orizzontale:

$$\begin{cases} X = X + \Delta T * V_x \\ V_x = V_x * (1 - \Delta T * b / M) + \Delta T * \sin(-\theta) * (F_1 + F_2) / M \end{cases}$$

Traslazione Orizzontale

Schema a blocchi per il controllo della traslazione orizzontale:



In questo caso è stata implementata la classe *X_Controller* che riceve come input fondamentale del metodo *evaluate x_target*, che rappresenta la coordinata lungo l'asse x che il multirotore dovrà raggiungere.

L'output di questo blocco costituisce l'angolo *theta_target* che verrà fornito come input al blocco descritto dall'*Angle_Controller* (visto precedentemente).

```
from controllore_p import *
from controllore_pi import *

class X_Controller:
    def __init__(self, p_kp, p_sat, pi_kp, pi_ki, pi_sat):
        self.p_controller = Controllore_P(p_kp, p_sat)
        self.pi_controller = Controllore_PI(pi_kp, pi_ki, pi_sat)
    def evaluate(self, delta_t, x_target, x_current, vx_current):
        x_error = x_target - x_current
        vx_target = self.p_controller.evaluate(x_error)
        vx_error = vx_target - vx_current
        theta_target = self.pi_controller.evaluate(delta_t, vx_error)

        return theta_target
```

Traslazione Verticale

Viene modellata attraverso la seguente legge: $(F_1 + F_2)cos\theta - bv_z - Mg = M\dot{v}_z$

da cui si ottiene il seguente sistema:

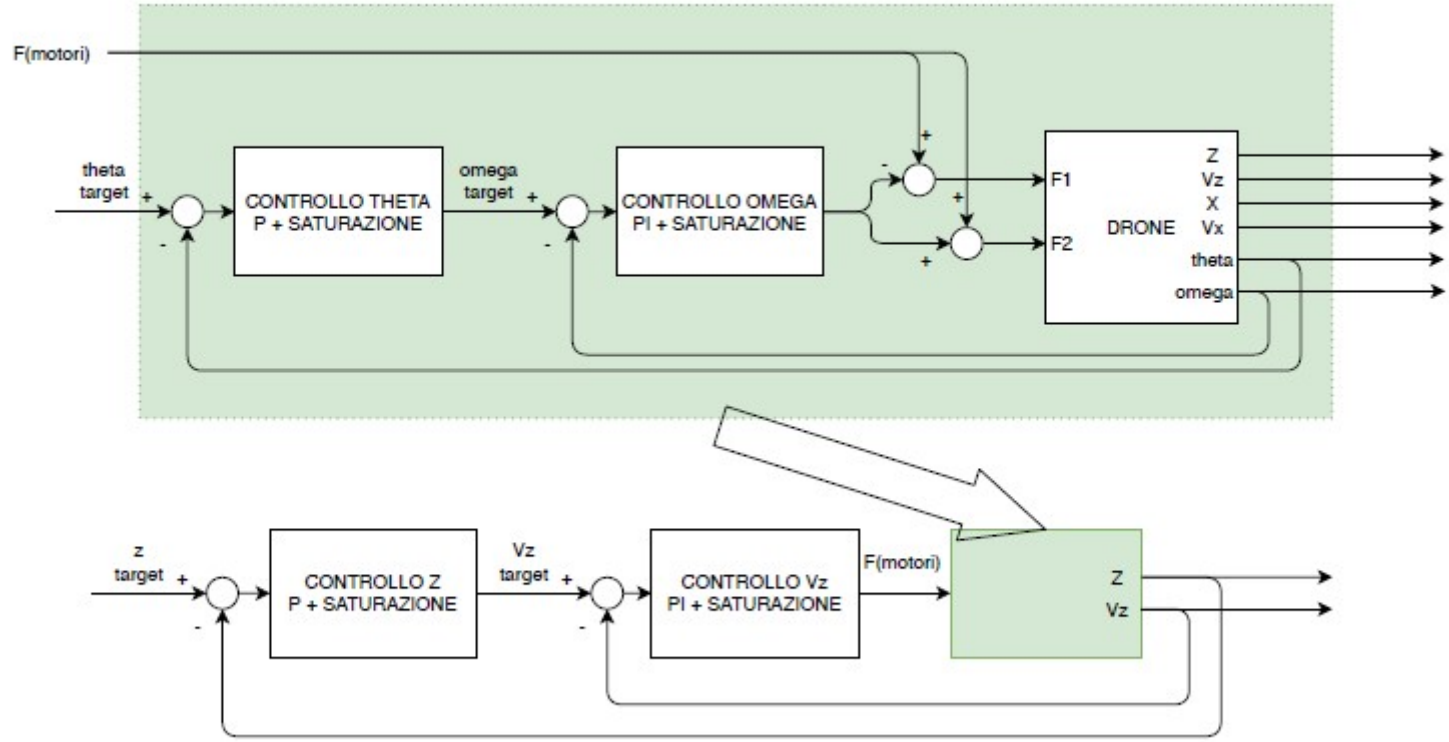
$$\begin{cases} \dot{z} = v_z \\ \dot{v}_z = -\frac{b}{M}v_z + \frac{F_1+F_2}{M}cos\theta - g \end{cases}$$

Procedendo con la discretizzazione di tale sistema, si ottiene infine la legge di aggiornamento dello stato per la traslazione verticale:

$$\begin{cases} Z = Z + \Delta T * V_z \\ V_z = V_z * (1 - \Delta T * b / M) + \Delta T * \cos(\theta) * (F_1 + F_2) / M - \Delta T * g \end{cases}$$

Traslazione Verticale

Schema a blocchi per il controllo della traslazione verticale:



Infine è stata implementata la classe *Z_Controller* che riceve come input del metodo *evaluate z_target*, che rappresenta la coordinata lungo l'asse z che il multirotore dovrà raggiungere.

L'output del blocco rappresenta la spinta con cui verranno modulati i due ingressi del velivolo, ovvero *f1* e *f2*, considerando anche l'uscita del blocco costituito dall'*Angle_Controller*.

```
from controllore_p import *
from controllore_pi import *

class Z_Controller:
    def __init__(self, p_kp, p_sat, pi_kp, pi_ki, pi_sat):
        self.p_controller = Controllore_P(p_kp, p_sat)
        self.pi_controller = Controllore_PI(pi_kp, pi_ki, pi_sat)
    def evaluate(self, delta_t, z_target, z_current, vz_current):
        z_error = z_target - z_current
        vz_target = self.p_controller.evaluate(z_error)
        vz_error = vz_target - vz_current
        spinta_motori = self.pi_controller.evaluate(delta_t, vz_error)

        return spinta_motori
```

Modello Semplificato Multirotore

Realizzato a partire dalla seguente classe *Multirotore*, di cui distinguiamo i due componenti (metodi) principali: *init* e *evaluate*.

Con il metodo *init* vengono inizializzati i vari attributi della classe nel momento in cui viene instanziato l'oggetto, tra cui le variabili di stato che descrivono la posa del velivolo {X, Z e θ}.

```
def __init__(self, massa, lunghezza, attrito):
    self.massa = massa
    self.lunghezza = lunghezza
    self.attrito = attrito
    self.inerzia = (1/12.0)*self.massa*4.0*(self.lunghezza**2)
    self.grv = 9.81

    self.theta = 0.0
    self.omega = 0.0

    self.x = 0.0
    self.vx = 0.0

    self.z = 0.0
    self.vz = 0.0
```

Con il metodo *evaluate* invece viene implementato il modello cinematico del multirotore, a partire dalle leggi di aggiornamento dello stato che sono state ricavate precedentemente dai processi di discretizzazione dei vari aspetti dinamici:

- dinamica rotazionale;
- traslazione orizzontale;
- traslazione verticale.

```
def evaluate(self, delta_t, f1, f2):
    theta_tmp = self.theta + delta_t*self.omega
    omega_tmp = self.omega + delta_t*self.lunghezza/self.inerzia*(f2 - f1)

    x_tmp = self.x + delta_t*self.vx
    vx_tmp = self.vx*(1 - delta_t*self.attrito/self.massa) +
    delta_t*(math.sin(-self.theta))*(f1 + f2)

    z_tmp = self.z + delta_t*self.vz
    vz_tmp = self.vz*(1 - delta_t*self.attrito/self.massa) - delta_t*self.grv +
    delta_t/self.massa*(math.cos(self.theta))*(f1 + f2)

    self.theta = theta_tmp
    self.omega = omega_tmp

    self.x = x_tmp
    self.vx = vx_tmp

    self.z = z_tmp
    self.vz = vz_tmp
```

Classe Autopilot

Avendo modellato tutti i componenti necessari, viene poi implementata la classe *Autopilot* che, oltre ad instanziare i vari componenti, si occupa di far rispettare il corretto flusso di dati ai fini dell'elaborazione e del raggiungimento dell'obiettivo finale, che consisterà nel far seguire un determinato path al multirotore.

Si noti che il path sarà composto da una serie di punti, descritti tramite coordinate (X, Z), che vengono forniti ante esecuzione del software di volo.

Anche in questo caso verranno analizzati i due metodi fondamentali della suddetta classe: *init* e *evaluate*.

Init

Con il metodo *init* vengono instanziati tutti i componenti e inizializzati i parametri che verranno utilizzati durante l'elaborazione.

X_target e *Z_target* rappresentano proprio quelle variabili che costituiranno il target corrente che il velivolo si porrà come obiettivo da raggiungere.

Path invece è l'array che conterrà tutti i punti che formeranno il percorso da seguire.

```
def   init  (self):
    self.drone = Multirotore(1.0, 0.25, 7.0*(10.0**(-5)))
    self.angle_controller = Angle_Controller(4.0, 1.57, 2.0, 0.2, 15.0)
    self.x_controller = X_Controller(0.4, 2.0, 0.5, 0.2, 0.52)
    self.z_controller = Z_Controller(4.0, 2.0, 20.0, 40.0, 15.0)

    self.x_target = 0.0
    self.z_target = 0.0

    self.threshold = 0.2
    self.flag = 0

    self.tmp = 0

    self.path = []
```

Evaluate

Costituisce il corpo del software di volo dove vengono concentrati i vari aspetti critici.

```
def evaluate(self, delta_t):
    distance = math.sqrt(((self.x_target - self.drone.x)**2) + (self.z_target -
    self.drone.z)**2)
    if(self.flag == 0 and distance <= self.threshold):
        self.showPath()
        print("Punto " + str(self.path[0]) + " raggiunto")
        print("------------------------------")
        self.path.pop(0)
        if(len(self.path) == 0):
            print("Il Path e' stato completato con successo  :)")
            self.flag = 1
            return
        self.showPath()
        (x, z) = self.path[0]
        self.x_target = x
        self.z_target = z
        print("Nuovo Target: " + str((x,z)))
        print("------------------------------")
        print("------------------------------")
    theta_target = (-1)*self.x_controller.evaluate(delta_t, self.x_target, self.drone.x,
    self.drone.vx)
    out_1 = self.angle_controller.evaluate(delta_t, theta_target, self.drone.theta,
    self.drone.omega)

    out_2 = self.z_controller.evaluate(delta_t, self.z_target, self.drone.z,
    self.drone.vz)

    f1 = out_2 - out_1
    f2 = out_2 + out_1

    self.drone.evaluate(delta_t, f1, f2)
```

Evaluate

Nella prima parte del metodo viene effettuato un controllo sul raggiungimento del target tramite la distanza Euclidea, considerando ovviamente il target da raggiungere e la posizione corrente del velivolo.

Se la distanza è inferiore ad una certa soglia, il target può considerarsi raggiunto.

Il punto in questione viene dunque eliminato dall'array *path* (tramite *pop*), ovvero dalla posizione *head*, che invece verrà assunta dal successivo punto del percorso precedentemente impostato.

Se il punto in esame risultasse essere l'ultimo del path ciò implicherebbe che la dimensione dell'array sarebbe nulla, ovvero che il percorso predeterminato sarebbe stato completato con successo.

In caso contrario, vengono estratte dalla *head* dell'array *path* (posizione 0) le coordinate del successivo punto da raggiungere che serviranno per aggiornare le variabili *x_target* e *z_target* su cui si baserà l'elaborazione sottostante.

Evaluate

La parte centrale dell'elaborazione è finalizzata ad estrarre i valori *f1* e *f2* che poi verranno utilizzati come ingressi del multirotore, al fine di consentire la modifica della posa del velivolo stesso e regolarne la movimentazione seguendo le leggi ottenute dall'analisi del modello fisico iniziale.

Esecuzione tramite analisi dei grafici nei termini dei vari target

File: *final_test_system_3.py*
Comando: *python3 final_test_system_3.py*

```
from autopilot_up import *
import pylab
import math

delta_t = 1.0*(10.0**(-3))
t = 0

pilota = Autopilot()

#--------------------------
# PATH 1
pilota.addPoint(0.0, 1.0)
pilota.addPoint(2.0, 1.0)
pilota.addPoint(2.0, 3.0)
pilota.addPoint(0.0, 3.0)
pilota.addPoint(-2.0, 3.0)
pilota.addPoint(-2.0, 1.0)

#--------------------------
# PATH 2
#pilota.addPoint(-2.0, 1.0)
#pilota.addPoint(2.0, 1.0)
#pilota.addPoint(0.0, 0.8)

#--------------------------
# PATH 3
#pilota.addPoint(2.0, 2.0)
#pilota.addPoint(-2.0, 0.0)
#pilota.addPoint(-2.0, 3.0)

vett_x = []
vett_z = []

while t <= 20.0:
    pilota.evaluate(delta_t)

    vett_x.append(pilota.drone.x)
    vett_z.append(pilota.drone.z)

    t = t + delta_t

pylab.figure(1)
pylab.plot(vett_x, vett_z, 'b-', label = "target")
pylab.legend()

pylab.show()
```

Esecuzione del simulatore grafico relativo alla movimentazione del drone

File: *final_sim_grafico_3.py*
Comando: *python final_sim_grafico_3.py*

```
import sys
import math

from PyQt4 import QtGui, QtCore

from autopilot_up import *

class MainWindow(QtGui.QWidget):
    def __init__(self):
        super(MainWindow, self).__init__()
        self.initUI()
    def initUI(self):
        self.setGeometry(0, 0, 800, 600)
        self.setWindowTitle('Simulatore Quadrirotore 2D')
        self.show()

        self.drone = QtGui.QPixmap("drone.png")

        self.delta_t = 1.0*(10.0**(-3))

        self._timer_painter = QtCore.QTimer(self)
        self._timer_painter.start(self.delta_t*1000)
        self._timer_painter.timeout.connect(self.go)

        self.pilota = Autopilot()

        self.pilota.addPoint(0.0, 1.0)
        self.pilota.addPoint(2.0, 1.0)
        self.pilota.addPoint(2.0, 3.0)
        self.pilota.addPoint(0.0, 3.0)
        self.pilota.addPoint(-2.0, 3.0)
        self.pilota.addPoint(-2.0, 1.0)

    def go(self):
        self.pilota.evaluate(self.delta_t)

        self.update()
```

```
    def paintEvent(self, event):
        qp = QtGui.QPainter()
        qp.begin(self)
        qp.setPen(QtGui.QColor(255, 255, 255))
        qp.setBrush(QtGui.QColor(255, 255, 255))
        qp.drawRect(event.rect())

        qp.setPen(QtGui.QColor(0, 0, 0))
        qp.drawText(650, 20, "X = %6.3f m" % (self.pilota.drone.x))
        qp.drawText(650, 40, "Vx = %6.3f m/s" % (self.pilota.drone.vx))
        qp.drawText(650, 60, "Z = %6.3f m" % (self.pilota.drone.z))
        qp.drawText(650, 80, "Vz = %6.3f m/s" % (self.pilota.drone.vz))
        qp.drawText(650, 100, "Theta = %6.3f deg" % (math.degrees(self.pilota.drone.theta)))
        qp.drawText(650, 120, "Omega = %6.3f rad/s" % (self.pilota.drone.omega))

        x pos = 336 + (self.pilota.drone.x*100)
        y pos = 500 - (self.pilota.drone.z*100)

        t = QtGui.QTransform()
        s = self.drone.size()
        t.translate(x pos + s.height()/2, y pos + s.width()/2)
        t.rotate(-math.degrees(self.pilota.drone.theta))
        t.translate(-(x_pos + s.height()/2), -(y_pos + s.width()/2))

        qp.setTransform(t)
        qp.drawPixmap(x_pos, y_pos, self.drone)

        qp.end()

def main():
    app = QtGui.QApplication(sys.argv)
    ex = MainWindow()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()
```

Fine