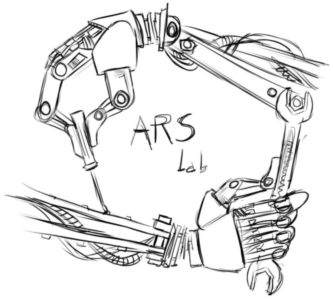
Progetto e realizzazione di un framework per il controllo di un sistema robotico

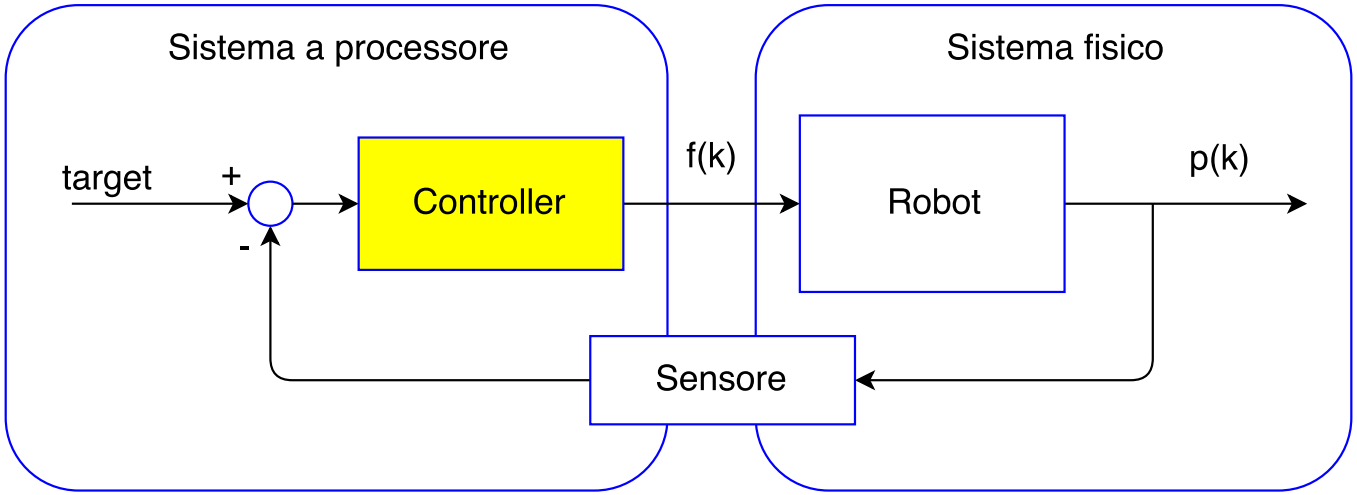
Corrado Santoro

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



Programmazione Sistemi Robotici

Il Problema del “Controllo”



`doControl(`*target*`)`

while *True* **do**
    On each ΔT;
    $current \leftarrow read\_sensors()$;
    $error \leftarrow target - current$;
    $f \leftarrow controller(error)$;
    $drive\_actuators(f)$;
end

Il Problema del "Controllo"

`doControl(`*target*`)`

while *True* **do**
    On each ΔT;
    $current \leftarrow read\_sensors()$;
    $error \leftarrow target - current$;
    $f \leftarrow controller(error)$;
    $drive\_actuators(f)$;
end

- Il codice riportato deve essere eseguito **sempre!**
- Tuttavia il **target** (set-point) **cambia** durante l'operatività del robot
- Thread/processo specifico per il **loop di controllo**
- Cambio "dall'esterno" della variabile **target** (uso di opportuni costrutti per il controllo della concorrenza)

L'importanza della periodicità

`doControl(`*target*`)`

while *True* **do**
    On each ΔT;
    ...;
end

- La **periodicità** è importantissima!!
- Deve essere **rispettata** in modo **stringente**, altrimenti il sistema potrebbe non essere adeguatamente "controllabile"
- Tuttavia, questo nei sistemi multiprogrammati "tradizionali" **non è affatto garantito**

L'importanza della periodicità

`doControl(`*target*`)`

while *True* **do**
    On each ΔT;
    ...;
end

Tecniche di garanzia del vincolo temporale

- Programmazione "bare metal" e uso di timer hardware
- Task ad interrupt, agganciati a timer hardware
- Kernel "real-time" con garanzia delle priorità
- Kernel "real-time" con supporto nativo per i task periodici

L'importanza della periodicità

`doControl(`*target*`)`

while *True* **do**
    On each ΔT;
    ...;
end

Riusciremo a soddisfare il vincolo del tempo?

- No! Il framework python che realizzermo **non ha** supporto "real-time"
- Tuttavia "nella maggior parte dei casi" riesce a soddisfare il vincolo
- E' dunque un framework **didattico**

Periodic Task

Periodic Task

- Classe base per implementare i **task periodici**
- Usa la classe **`Timer`** del package “threading” di python
- E’ implementato come un thread “a tempo” che, allo scadere di un timer, chiama un metodo di callback

```
import threading
import time

class PeriodicTask:

    def __init__(self, period):
        self.__period = period
        self.__lasttime = None

    def start(self):
        threading.Timer(self.__period, self.__callback).start()

    def __callback(self):
        ....
```

Periodic Task

- La classe gestisce la periodicità riavviando il timer
- Calcola il “periodo effettivo” sull’attributo (pubblico) **`cycletime`** (da usarsi per i calcoli basati sul tempo)
- Avvia il metodo **`run`** la cui implementazione specificaè lasciata alla classe ereditata

```
import threading
import time

class PeriodicTask:
    ....
    def __callback(self):
        current = time.time()
        self.start()
        if self.__lasttime is None:
            self.cycletime = 0
        else:
            self.cycletime = current - self.__lasttime
        self.__lasttime = current
        self.run()

    def run(self):
        pass
```

Cinematica del Sistema

- Ogni sistema fisico ha una sua “cinematica” di cui il software deve tenere conto
- Le caratteristiche cinematiche implicano il calcolo delle grandezze cinematiche del sistema: **posizione**, **velocità**, **accelerazione**
- Implementiamo una classe `Kinematics` che, interfacciata al MotorDriver, e invocata periodicamente, effettua il campionamento dell’encoder e calcola le grandezze cinematiche

Kinematics

```
#
# attributes:
#    speed = current speed in RPM
#    position = current position in ticks
#
class Kinematics:

    def __init__(self, motor_driver):
        self.__motor_driver = motor_driver
        self.position = None

    def run(self, delta_t):
        # get current encoder
        p = self.__motor_driver.encoder()
        if self.position is None:
            # if first time speed is 0
            self.speed = 0
        else:
            # compute speed
            s = (p - self.position) / delta_t
            # convert in RPM (84000 ticks per revolution * 60 revolutions in a minute)
            self.speed  = (s / 84000.0) * 60.0

        # update position
        self.position = p
```

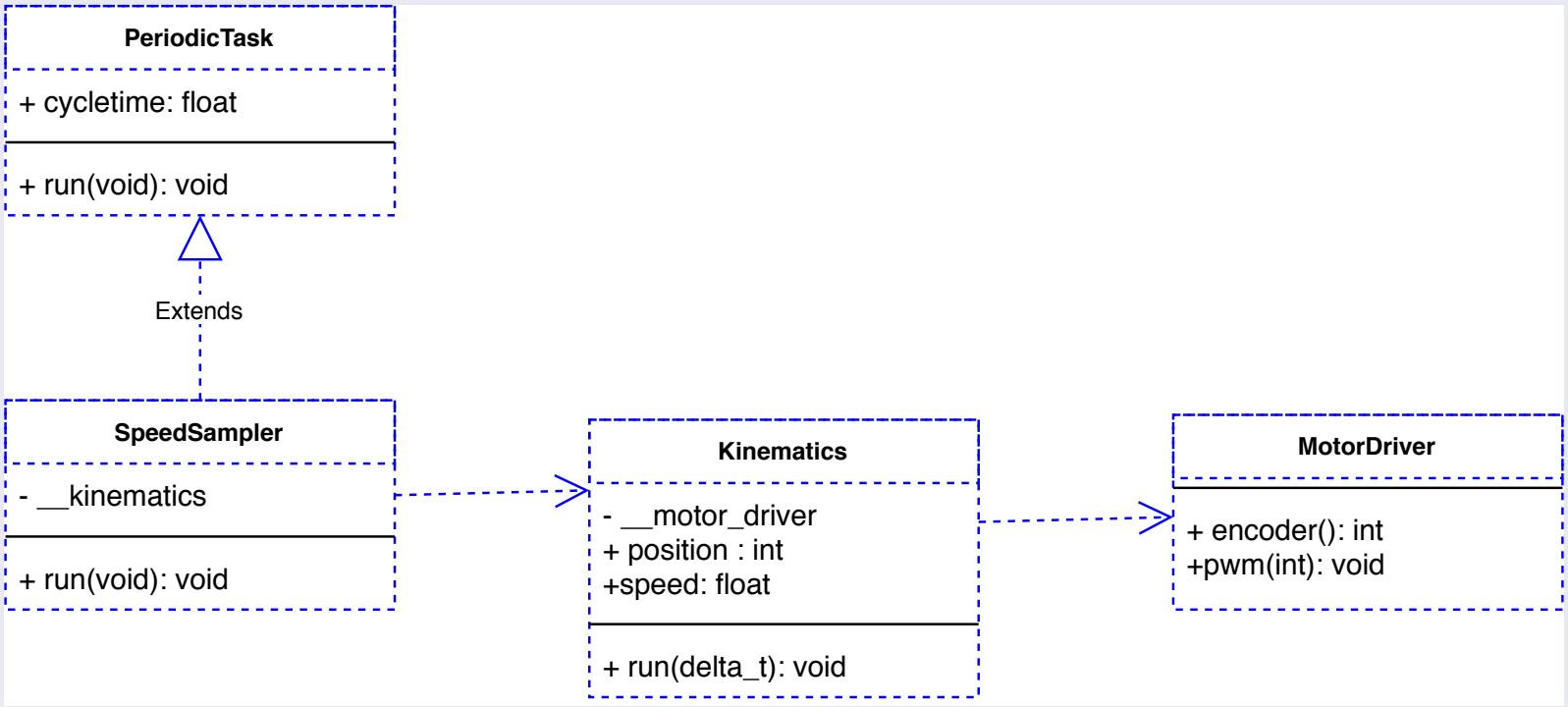
Test della Cinematica

Test

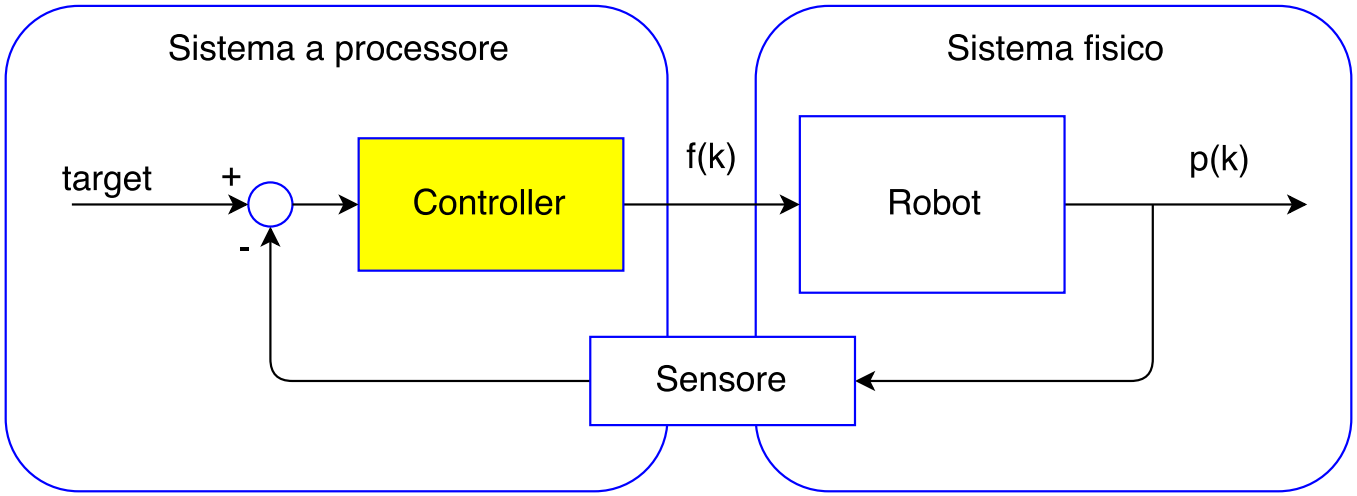
```
from motor_driver import *
from periodic_task import *
from simple_kinematics import *

class SpeedSampler(PeriodicTask):

    def __init__(self, driver):
        PeriodicTask.__init__(self, 0.01) # 10 ms of periodicity
        self.__kinematics = Kinematics(driver)

    def run(self):
        self.__kinematics.run(self.cycletime)
        print "Current Speed ", self.__kinematics.speed

if __name__ == "__main__":

    driver = MotorDriver()
    driver.open()

    sampler = SpeedSampler(driver)
    sampler.start()

    driver.pwm(0)
```

Test della Cinematica

UML



Realizzazione del Sistema di Controllo

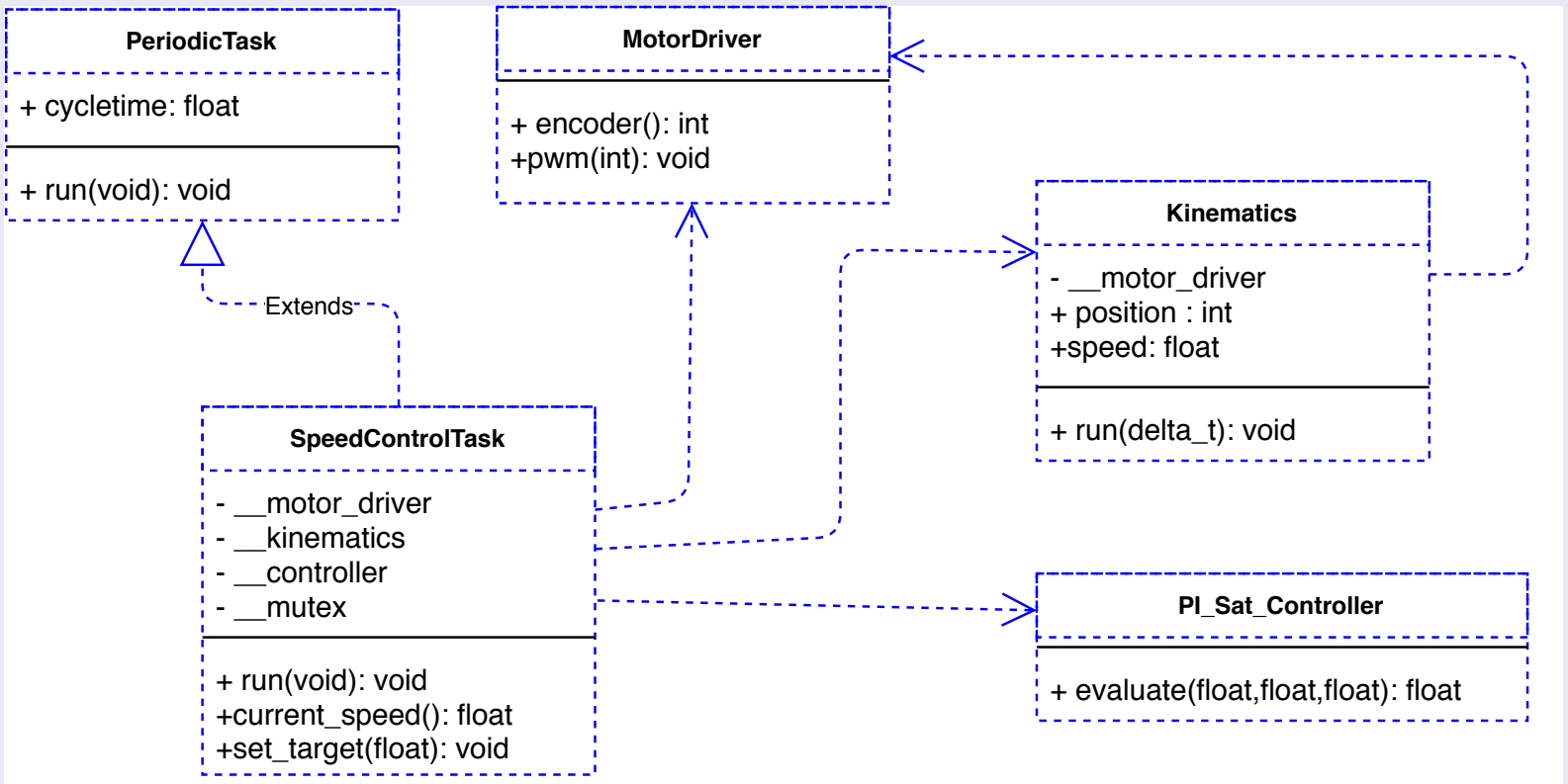


Sistema di Controllo

Realizziamo una classe **`SpeedControlTask`** che

- è ereditata da **`PeriodicTask`**
- è associata a **`Kinematics`** e **`MotorDriver`**
- utilizza **`PI_Sat_Controller`**
- implementa il loop di controllo nel proprio metodo **`run`**

UML


PeriodicTask
+ cycletime: float
+ run(void): void
Extends
MotorDriver
+ encoder(): int
+pwm(int): void
Kinematics
- __motor_driver
+ position : int
+speed: float
+ run(delta_t): void
SpeedControlTask
- __motor_driver
- __kinematics
- __controller
- __mutex
+ run(void): void
+current_speed(): float
+set_target(float): void
PI_Sat_Controller
+ evaluate(float,float,float): float

Speed Control Task

```
import threading

from periodic_task import *
from simple_kinematics import *
from controllers import *
from motor_driver import *

class SpeedControlTask(PeriodicTask):

    def __init__(self):
        PeriodicTask.__init__(self, 0.01) # 10ms
        self.__motor_driver = MotorDriver()
        self.__kinematics = Kinematics(self.__motor_driver)
        self.__target_speed = 0
        # KP = 50, KI = 400, Saturation = 4200
        self.__controller = PI_Sat_Controller(50, 400, 4200)
        self.__mutex = threading.Lock()

    def start(self):
        self.__motor_driver.open()
        PeriodicTask.start(self)

    def run(self):
        ....
```

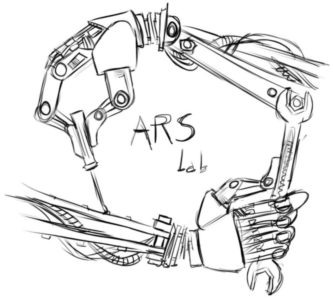
Speed Control Task

```
class SpeedControlTask(PeriodicTask):

    ....

    def run(self):
        self.__mutex.acquire()
        # run the kinematics
        self.__kinematics.run(self.cycletime)
        # do the control
        controller_output = self.__controller.evaluate(self.__target_speed,
                                                       self.__kinematics.speed,
                                                       self.cycletime)
        self.__mutex.release()
        # output PWM to driver
        self.__motor_driver.pwm(controller_output)

    def current_speed(self):
        self.__mutex.acquire()
        v =  self.__kinematics.speed
        self.__mutex.release()
        return v

    def set_target(self, newtarget):
        self.__mutex.acquire()
        self.__target_speed = newtarget
        self.__mutex.release()
```

Progetto e realizzazione di un framework per il controllo di un sistema robotico

Corrado Santoro

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



Programmazione Sistemi Robotici