DronLink: Funciones de la clase Dron

Se describen en la tabla siguiente las funciones de la clase Dron en la versión actual de la librería DronLink. En muchas de ellas aparecen los tres parámetros requeridos para implementar la modalidad no bloqueante (`blocking, callback` y `params`).

Al final de este documento hay una tabla que indica en qué módulo de la librería está el código de cada una de las funciones.

Instanciación de objetos de la clase Dron

El constructor de la clase está definido así:

```
class Dron(object):
    def __init__(self, id = None, verbose = False):
```

El parámetro `id` permite identificar al dron. Es de utilidad cuando se trabaja con un enjambre de drones. El identificador puede ser cualquier valor de cualquier tipo.

El parámetro `verbose` indica si queremos que los métodos escriban en consola información sobre lo que van haciendo.

Métodos básicos

Lo básico para el ciclo más elemental: conectar, armar, despegar, aterrizar y desconectar.

`def connect(self,`
`  connection_string, baud,`
`  id=None, freq = 4,`
`  blocking=True, callback=None, params = None)`

Conecta con el dron. Los parámetros `connection_string, baud` indican si hay que conectar con el simulador o con el dron real, y la velocidad de comunicación. Los strings de conexón habituales son: Simulador: `'tcp:127.0.0.1:5763'` y velocidad de 115200. Directamente al dron por radio de telemetría: `'com??'` y velocidad 57600 Con el dron a través de mavproxy: `'udp:127.0.0.1:14551'` y velocidad de 115200. La conexión admite una identificador para el dron, que la librería añadirá como primer parámetro en todas las funcione callback. El parámetro `freq` indica la frecuencia con la que se van a enviar datos de telemetría. Por defecto se envían 4 paquetes de datos por segundo. Con frecuencias mayores puede conseguirse, por ejemplo, mayor fluidez en el movimiento del icono del dron sobre un mapa, aunque puede saturarse un poco el computador en el que se ejecuta la aplicación. La conexión suele ser rápida. Por eso es habitual usar el modo bloqueante al usar esta función.
`def arm(self, blocking=True, callback=None, params = None)`
Arma el dron. El armado es rápido. Por eso es habitual usar el modo bloqueante al usar esta función.

`def takeOff(self,`
`  aTargetAltitude,`
`  blocking=True, callback=None, params = None)`

Despega el dron hasta alcanzar la altura indicada en el parámetro.
`def Land (self, blocking=True, callback=None, params = None)`

Ordena al dron que aterrice en el punto que está sobrevolando.
`def RTL (self, blocking=True, callback=None, params = None)`
Ordena al dron que retorne a casa (Return to launch).
`def disconnect (self)`
Desconecta el dron. Además, detiene el envío de datos de telemetría.

Navegación

Comandos para indicarle al dron hacia dónde tiene que navegar o a que punto geográfico debe ir.

`def changeNavSpeed (self, speed)`
Cambia la velocidad de navegación.

`def change_altitude(self,`
`  alt,`
`  blocking=True, callback=None, params=None)`

Cambia la altura a la que se encuentra el dron.
`def go(self, direction)`
Hace que el dron navegue en la dirección indicada. Las opciones son: `'North', 'South', 'West', 'East', 'NorthWest', 'NorthEast', 'SouthWest', 'SouthEast', 'Stop', 'Forward', 'Back', 'Left', 'Right', 'Up', 'Down'.`
`def setMoveSpeed (self, speed):`
Fija la velocidad (m/s) para las operaciones de movimiento

`def goto(self,`
`  lat, lon, alt,`
`  blocking=True, callback=None, params=None)`

Dirige al dron al punto geográfico indicado.

`def gotoLocal(self,`
`  X, Y, Z,`
`  blocking=True, callback=None, params=None)`

Dirige al dron al punto X,YZ (especificados en metros) en relación al sistema de coordenadas con origen en el punto de despegue. Se usa el esquema NED, es decir X corresponde a dirección North-South, Y corresponde a East-West y Z corresponde a DOWN-UP.
`def move distance(self, direction, distance,` `  blocking=True, callback=None, params = None):`
Mueve el dron (a la velocidad establecida con `setMoveSpeed`) los metros indicados por `distance` y en la dirección indicada. Los valores de `direction` pueden ser: `'Forward', 'Back', 'Left', 'Right', 'Up', 'Down', 'Stop', 'North', 'South', 'West', 'East'.`

Misiones y planes de vuelo

Comandos para cargar y ejecutar misiones y planes de vuelo. Una misión es una secuencia de waypoints que el autopiloto va a ejecutar de manera autónoma. Un plan de vuelo es una secuencia de waypoints (con el mismo formato que la misión) pero que la función de la librería ejecuta uno a uno, pudiendo incluso ejecutar una función indicada por el usuario cada vez que llega a uno de los waypoints del plan.

```
def uploadMission(self,
   mission,
   blocking=True, callback=None, params = None)
```

Carga en el autopiloto la misión indicada, que debe especificarse en ese formato:

```
{
    "speed": 7,
    "takeOffAlt": 5,
    "waypoints":
        [
            {
                'lat': 41.2763410,
                'lon': 1.9888285,
                'alt': 12
            },
            {'rotAbs': 90},
            {
                'lat': 41.27623,
                'lon': 1.987,
                'alt': 14
            },
            {'rotRel': 90, 'dir': -1}
        ]

}
```

El dron despegará en la posición en la que esté y al llegar al último waypoint harán un RTL. `rotAbs` indica una rotación absoluta para colocar el heading en los grados indicados. `rotRel` indica que hay que rotar tantos grados como los indicados en sentido antihorario (`'dir'=1`) o antihorario (`'dir'=-1`).

```
def getMission(self,
   blocking=True, callback=None)
```

Retorna la misión que está cargada en ese momento en el autopiloto (o None si no hay misión). En el caso de que la llamada sea no bloqueante llamará a la función callback pasándole como parámetro la misión. El formato en que retorna la misión es el indicado en la descripción del método `uploadMission`.

```
def executeMission(self,
   blocking=True, callback=None, params = None)
```

El dron ejecuta la última misión que se haya cargado.

```
def executeFlightPlan(self, flightPlan, inWaypoint =None,
   blocking=True, callback=None, params = None)
```

El dron ejecuta, paso a paso, el plan de vuelo, que debe especificarse en el mismo formato que en el caso de la mision. Si se ha especificado una función `inWaypoint` entonces se ejecutará esa función cada vez que se llegue a un waypoint, pasándole como parámetro a esa función el número de secuencia del waypoint y el propio waypoint.

Escenarios

Los escenarios incluyen un geofence de inclusión y varios geofences de exclusión que representan obstáculos en la zona de vuelo (puede no haber obstáculos).

```
def setScenario(self,
   scenario,
   blocking=True, brench= None, callback=None, params = None)
```

El escenario se recibe en forma de lista. En cada posición hay un fence que representan áreas.
El primer elemento de la lista es un fence de inclusión, que representa el área de la que el dron no va a salir. El resto de elementos de la lista son fences de exclusión que representan obstáculos dentro del fence de inclusión, que el dron no puede sobrevolar. El escenario debe tener un fence de inclusión (solo uno y es el primer elemento de la lista) y un número variable de fences de exclusión, que puede ser 0.
Un fence (tanto de inclusión como de exclusión) puede ser de tipo 'polygon' o de tipo 'circle'. En el primer caso el fence se caracteriza por un número variable de waypoints (lat, lon). Deben ser al menos 3 puesto que representan los vértices del polígono. Si el fence es de tipo 'circle' debe especificarse las coordenadas (lat, lon) del centro del círculo y el radio en metros.
Un ejemplo de scenario en el formato correcto es este:

```
scenario = [
        {
            'type': 'polygon',
            'waypoints': [
                {'lat': 41.2764398, 'lon': 1.9882585},
                {'lat': 41.2761999, 'lon': 1.9883537},
                {'lat': 41.2763854, 'lon': 1.9890994},
                {'lat': 41.2766273, 'lon': 1.9889948}
            ]
        },
        {
            'type': 'polygon',
            'waypoints': [
                {'lat': 41.2764801, 'lon': 1.9886541},
                {'lat': 41.2764519, 'lon': 1.9889626},
                {'lat': 41.2763995, 'lon': 1.9887963},
            ]
        },
        {
            'type': 'polygon',
            'waypoints': [
                {'lat': 41.2764035, 'lon': 1.9883262},
                {'lat': 41.2762160, 'lon': 1.9883537},
                {'lat': 41.2762281, 'lon': 1.9884771}
            ]
        },
        {
            'type': 'circle',
            'radius': 2,
            'lat': 41.2763430,
            'lon': 1.9883953
        }
    ]
```

El escenario tiene 4 fences. El primero es el de inclusión, de tipo 'polygon'. Luego tiene 3 fences de exclusión que representan los obstáculos. Los dos primeros son de tipo 'polygon' y el tercero es de tipo 'circle'.

El parámetro brench es una función de callback que proporciona el usuario para el caso de que se produzca una violación de cualquiera de los fences.

```
def getScenario(self,
   blocking=True, callback=None)
```

Retorna el escenario que en ese momento está cargado en el dron. En el caso de que la llamada sea no bloqueante, llamará a la función de callback pasándole el escenario. El formato del escenario es el indicado en la descripción del método `setScenario`.

Control del heading

`def fixHeading(self)`
Hace que en las siguientes operaciones de navegación el dron se mueva en la dirección correspondiente sin cambiar el heading.
`def unfixHeading(self)`
En las siguientes operaciones de navegación el heading puede cambiar dependiendo de la dirección en la que se mueva el dron.
`def changeHeading (self, absoluteDegrees, blocking=True, callback=None, params = None)`
Hace que el dron rote hasta alcanzar el heading indicado.
`def rotate (self, offset, direction = 'cw', blocking=True, callback=None, params = None):`
Hace que el dron rote los grados indicados por `offset` en el sentido indicado por `direction` (sentido horario o antihorario).

Sensor de distancia

Naturalmente, estas funciones solo son operativas si se ha instalado en el dron un sensor de distancia.

`def send_distance_sensor_info (self, process_distance_info, freq = 4):`

Hace que el dron llame a la función indicada, con la frecuencia indicada, y le entregue a la función un paquete con los datos de distancia captados por el sensor. El paquete de datos tiene este formato:
`distance_info = {`
`  'distance': self.distance,`
`  'orientation': self.orientation,`
`}`
El campo `orientation` solo tiene información válida si se usa un sensor de 360º (como el `"RPLIDAR C1"`).

`def stop_sending_distance_sensor_info (self):`
Detiene el envío de los datos de distancia.
`def ConfigureDistanceSensor (self, sensor)`
Actualmente se consideran dos posibles tipos de sensores. El sensor `"RPLIDAR C1"` proporciona datos en 360º. En este caso, los paquetes de distancia contienen la distancia y también la orientación a la que se obtuvo esa distancia. El sensor `"TFmini"` es unidireccional y solo proporciona información sobre la distancia medida.

Gestión de parámetros

```
def getParams(self,
   parameters,
   blocking=True, callback=None)
```

Pide al dron el valor de los parámetros indicados. En el caso de que la llamada sea o bloqueante, ejecutará la función del callback pasándole como parámetro la lista de valores recibida (y el identificador del dron como primer parámetro en el caso de que el dron haya sido identificando en el momento de la conexión). Este ejemplo muestra el formato en el que debe pasarse la lista de parámetros y el formato de la respuesta.

```
parameters = [
    "RTL_ALT",
    "PILOT_SPEED_UP",
    "FENCE_ACTION"
]
result = dron.getParams(parameters)
print('Valores:',result)
```

El resultado podría ser:

```
Valores: [{'RTL_ALT': -1.0}, {'PILOT_SPEED_UP': 100.0}, {'FENCE_ACTION': 4.0}]
```

```
def setParams(self,
   parameters,
   blocking=True, callback=None, params = None)
```

Establece el valor de los parámetros que se le pasan en una lista. Un ejemplo de uso es este:

```
parameters =[
        {'ID': "FENCE_ENABLE", 'Value': 1},
        {'ID': "FENCE_ACTION", 'Value': 4}
]
dron.setParams(parameters)
```

Telemetría

Se diferencia entre telemetría y telemetría local. Los datos de telemetría son los que pueden obtenerse cuando se recibe la señal GPS. Los de telemetría local son los que pueden obtenerse cuando se navega en interiores y los datos de posición provienen de un Optical Flow, porque no se recibe señal GPS.

`def send_telemetry_info(self, process_telemetry_info)`

Pide al dron que envíe los datos de telemetría. Cuando el dron tiene un nuevo paquete de telemetría llama al callback y le pasa ese paquete (y el identificador del dron como primer parámetro en el caso de que el dron haya sido identificando en el momento de la conexión). El dron va a enviar tantos paquetes por segundo como indique el parámetro `freq` en el momento de la conexión. El ejemplo siguiente muestra el formato en que se reciben los datos de telemetría:

```
def process_telemetry_info(self, telemetry_info):
    print ('info: ', telemetry_info)
```

El paquete de datos de telemetría contiene los datos siguientes:

```
telemetry_info = {
    'lat': self.lat,
    'lon': self.lon,
    'alt': self.alt,
    'groundSpeed':  self.groundSpeed,
    'heading': self.heading,
    'state': self.state,
    'flightMode': self.flightMode,
    'voltage_battery': self.voltage_battery,
    'current_battery': self.current_battery,
    'battery_remaining': self.battery_remaining
}
```

Los posibles estados en los que puede estar el dron son: `'connected'`, `'disconnected'`, `'arming'`, `'armed'`, `'takingOff'`, `'flying'`, `'returning'`, `'landing'`.

`def send_local_telemetry_info(self, process_local_telemetry_info)`

Pide al dron que envíe los datos de telemetría local. Cuando el dron tiene un nuevo paquete de telemetría llama al callback y le pasa ese paquete (y el identificador del dron como primer parámetro en el caso de que el dron haya sido identificando en el momento de la conexión). El dron va a enviar tantos paquetes por segundo como indique el parámetro `freq` en el momento de la conexión. El ejemplo siguiente muestra el formato en que se reciben los datos de telemetría:

```
def process_local_telemetry_info(self, local_telemetry_info):
    print ('info: ', local_telemetry_info)
```

El paquete de datos de telemetría contiene los datos siguientes:

```
local_telemetry_info = {
    'posX': self.position[0],
    'posY': self.position[1],
    'posZ': self.position[2],
    'velX': self.speeds[0],
    'velY': self.speeds[1],
    'velZ': self.speeds[2]
}
```

El valor `posX` es el desplazamiento en metros en el que se encuentra el dron en la dirección del Norte, respeto al home (o en la dirección de Sur si el valor es negativo). De manera similar, `posY` indica el desplazamiento en la dirección Este (u Oeste para valores negativos) y `posZ` el desplazamiento hacia abajo (o hacia arriba para valores negativos).

`def stop_sending_telemetry_info(self)`	
	Detiene el envío de datos de telemetría.
`def stop_sending_local_telemetry_info(self)`	
	Detiene el envío de datos de telemetría local.

Radio Control

Se trata de funciones que permiten enviar al dron, por programa, las mismas señales que se enviaría al dron mediante la emisora de radio control.

`def send_rc (self, roll, pitch, throttle, yaw ):`
Envía valores para cada uno de los ejes de control. Los valores deben estar entre 1000 y 2000. En la posición central todos los stics envían en valor 1500.

Vuelo en interiores

Este grupo de funciones permiten volar el dron en espacios interiores en los que no se recibe la señal GPS. Es ese escenario, el dron necesita un optical flow para tener información sobre posicionamiento y para poder navegar a puntos específicos, por ejemplo usando la función gotoLocal.

`def EstablecerLimites(self, limites, callback = None):`

Informa al dron de la geografía del espacio interior. Este es un ejemplo del formato de la estructura de datos necesaria:
`limites = {`
`'minAlt': 2,`
`'maxAlt':10,`
`'inclusion': [(3,5), (7,9), (-2,4)],`
`'obstaculos': [`
`[(2,9), (4,7), (8,8)],`
`[(-1,3), (3,3), (10,10)]`
`]`
`}`
Los valores de `minAlt` y `maxAlt` indica las alturas mínimas y máximas a las que puede volar el dron. El valor de `inclusion` especifica el polígono que define los limites externos del espacio de vuelo, según el sistema de coordenadas NED, con origen en el punto de despegue. En `obstaculos` tenemos una lista de polígonos que especifican los obstáculos del espacio de vuelo, que el dron no puede atravesar.
La función `callback` se llamará cada vez que el dron esté peligrosamente cerca de cualquiera de esos límites. La librería pasará tres parámetros a esa función: el `id` del dron y los valores `elemento` y `situacion`. El valor de `elemento` indica qué tipo de límite está implicado en la situación, de acuerdo con el siguiente criterio:
`-2: altura mínima`
`-1: altura máxima`
`0: limites externos`
`i: obstáculo i-ésimo de la lista`
 	
El valor de `situacion` indica la situación según el siguiente criterio:
`1: el dron está peligrosamente cerca del límite`
`0: el dron ha vuelto a zona segura`

	`2: el dron ha superado el límite y va a realiza una operación de RTL`
`def ActivaLimitesIndoor (self):`	
	Pone en marcha el sistema de control. En el caso de que el dron se acerque peligrosamente a alguno de los límites, además de activar la función de callback, la librería reducirá significativamente la velocidad del dron hasta que el piloto haya colocado el dron suficientemente lejos (zona segura). En el caso de que el piloto insista y supere el límite, entonces se realizará un RTL. En el caso de acercarse peligrosamente a los límites de altitud, la librería hará que el dron ascienda o descienda un poco para situarse en zona segura.
`def DesactivaLimitesIndoor (self):`	
	Detiene el sistema de control de vuelo indoor.
`def ConfiguraVueloIndoor (self):`	
	Configura los parámetros para que el dron use los datos del optical flow (y no los del GPS) para posicionamiento y navegación.
`def ConfiguraVueloExterior (self):`	
	Configura los parámetros para que el dron use los datos del GPS (y no los del optical flow) para posicionamiento y navegación.
`def SetHome (self):`	
	Hace que el dron establezca como home la posición en la que está en ese momento. Es muy importante hacer esto en el momento de conectar el dron antes de hacerle volar en interior porque de no ser así puede malinterpretar los datos que proporciona el optical flow, con fatales consecuencias para la estabilidad del dron.

Miscelánea

`def drop(self):`	
	Cambia la posición del servo, espera un segundo y regresa a la posición original.
`def reboot (self):`	
	Reinicia el autopiloto
`def setFlightMode (self,mode):`	
	Establece el modo de vuelo. Los más habituales son: ‘GUIDED’, ‘LAND’, ‘RTL’, ‘LOITER’, ‘BRAKE’.

Correspondencia entre módulos y funciones

Modulo	Funciones
dron_connect.py	connect disconnect reboot
dron_arm.py	arm setFlightMode
dron_takeoff.py	Takeoff
dron_RTL_Land.py	RTL Land
dron_parameters.py	getParams setParams
dron_telemetry.py	send_telemetry_info stop_sending_telemetry_info
dron_RC_override.py	send_rc
dron_nav.py	changeNavSpeed go
dron_move.py	move_distance setMoveSpeed
dron_mission.py	uploadMission executeMission getMission executeFlightPlan
dron_local_telemetry.py	send_local_telemetry_info stop_sending_local_telemetry_info
dron_heading.py	fixHeading uynfixHeading changeHeading rotate
dron_goto.py	goto gotoLocal
dron_geofence.py	getScenario setScenario
dron_drop.py	Drop
dron_distanceSensor.py	send_distance_sensor_info stop_sending_distance_sensor_info ConfigureDistanceSensor
dron_inDoor.py	EstablecerLimites ActivaLimitesIndoor DesactivaLimitesIndoor ConfiguraVueloIndoor ConfiguraVueloExterior SetHome
dron_altittude.py	change_altitude