
CAPÍTULO 1

SOFTWARE

El objetivo de esta sección es realizar una breve introducción sobre cómo utilizar la implementación en software del vuelo autónomo del cuadricóptero. Para entender en detalle qué hace cada función, referirse a los comentarios en el código fuente, disponible en el repositorio *Git* en la carpeta `src/`. Todas las referencias a archivos son relativas a la raíz del repositorio. Los programas están pensados para compilarse y ejecutarse en un entorno *Linux*.

En el anexo ?? se explica como compilar y configurar las partes involucradas.

1.1. Esquema general

El código tiene una estructura modular, está escrito en C, y cada bloque está implementado como una biblioteca. La estructura general del código se resume en la figura 1.1.

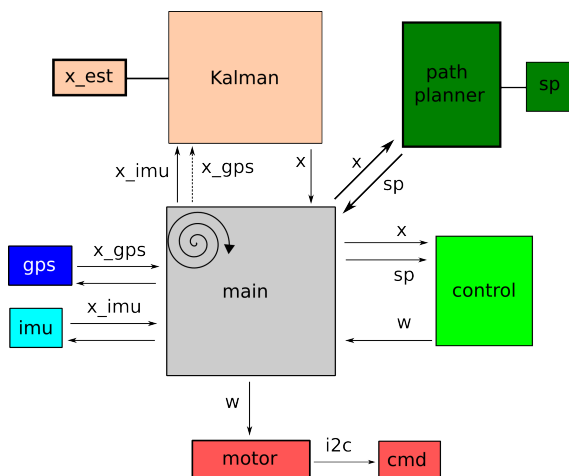


Figura 1.1: Estructura general del código.

1.2. Inicialización

Para correr el programa principal (de ahora en más: *main*) debería bastar con ejecutar el script `src/go.sh`. Durante la inicialización el *main* debe encontrar lo siguiente:

- `imu_calib.txt`: Parámetros de calibración, la biblioteca *imu_comm* los necesita para convertir los datos crudos provenientes de la IMU.
- `K*.txt`: Matrices de control utilizados en el modo *hover*.
- `lqr-*.txt`: Parámetros del algoritmo *LQR*.

- **IMU:** La IMU envía datos a través de una *UART* que es mapeada por el sistema operativo a un “archivo” `/dev/tty*`. El *main* recibe como parámetro la ruta a este archivo, o en su defecto un log `imu_raw.txt` generado por el *main* en una ejecución previa.
- **GPS:** Los datos provenientes del GPS (*USB*) son mapeados por el sistema operativo a `/dev/ttyUSB*`. No se interactúa directamente con este archivo, se utiliza la biblioteca *gps_comm* para iniciar un cliente que se comunice con el *gpsd*, que es el programa que se encarga de leer y analizar los datos crudos provenientes del GPS. El *gpsd* es iniciado por el script `go.sh`. Si no se dispone de señal del GPS se puede configurar un modo de prueba en el que se simulan los datos del GPS (a 1Hz) generando ceros, o números al azar dentro de un rango dado.
- **cmd:** El driver de los motores, encargado exclusivamente de enviar continuamente comandos *i2c* a los ESCs con la última velocidad configurada¹. Durante pruebas, se puede configurar el driver para que simule la presencia de los motores, o para que lea de la entrada estándar. Ver `src/i2c_beagle/README` por información sobre como compilar los distintos modos. La comunicación entre el driver y el *main* se realiza mediante la biblioteca *motor*, que a su vez se comunica con el driver mediante colas de kernel (IPC²), utilizando la biblioteca *uquad_kernel_msgq*.

1.3. Loop

A continuación se describe un loop normal ejecutado por el *main*, explicando brevemente las funcionalidades de cada una de las bibliotecas involucradas:

1. **imu:** La IMU genera datos nuevos cada 10ms. Al comienzo del loop, el *main* revisa si hay datos nuevos, y en caso afirmativo llama a la biblioteca para que los lea. Cuando se completa una trama, los datos crudos se almacenan en una cola circular. Al terminar de recibir una trama, se vuelve al principio del loop para verificar que no hay más nada para leer. En caso de haber más datos entonces hay que leerlos para evitar atrasarse respecto a la IMU, en caso contrario se convierten los datos y se avanza.
2. **gps:** El GPS genera datos nuevos a una tasa mucho menor que la IMU. Cada vez que se dispone de una muestra nueva en la IMU, el *main* revisa si también hay un dato nuevo del GPS. Avanza aunque no se disponga de datos nuevos del GPS.
3. **kalman:** El filtro de Kalman está implementado en la biblioteca *kalman*. Recibe una estructura de datos generada por *imu_comm* y otra (opcional) generada por *gps_comm*. Mantiene una estructura de datos que almacena el estado estimado y las matrices de covarianza.
4. **path planner:** El módulo generador de rutas está implementado en la biblioteca *path_planner*. Compara el estado actual con el objetivo, y determina si

¹Los motores se apagan si no reciben comandos continuamente.

²*Interprocess Communication*: <http://www.cs.cf.ac.uk/Dave/C>.

se completó el objetivo actual³. En caso afirmativo, devuelve una bandera que le indicará al módulo de control que debe actualizar la matriz de control para ajustarla a la nueva trayectoria.

5. **control:** El módulo de control está implementado en la biblioteca *control*. Mantiene una estructura con las matrices del control proporcional e integral (si corresponde). Recibe como argumento el estado estimado y la velocidad actual de los motores, y una estructura generada por *path_planner*, que indica el estado objetivo y la trayectoria a seguir. Devuelve la acción de control a aplicar sobre los motores.
6. **motor:** Se envía al driver una nueva velocidad deseada.

Por información relativa a bloques, configuración, compilación, ejecución, etc, referirse a `src/README`.

1.4. Módulo *imu_comm*

A continuación se describen algunas de las funcionalidades a destacar de la biblioteca *imu_comm*.

- **Calibración:** Se acumulan un conjunto de muestras que después se utilizan para estimar el offset de los giróscopos, la altura inicial, y pueden ser utilizados para inicializar el filtro de *Kalman*. Durante la calibración es crítico que el cuadricóptero no se mueva, ya que en caso de hacerlo el offset de los giróscopos será mal estimado.
La inclinación durante la calibración afecta la estimación inicial del offset en los acelerómetros, pero en caso de no estar perfectamente horizontal se acomodará luego de unos segundos, no es algo crítico.
- **Conversión:** Cargando parámetros de calibración, es posible convertir datos crudos provenientes de los sensores (cuentas de un ADC) a datos útiles:
 - Acelerómetro → Aceleraciones.
 - Giróscopo → Velocidad angulares.
 - Acelerómetro + Magnetómetro → Ángulos de Euler.
 - Barómetro → Altura y temperatura.
- **Filtrado:** Se disponen de funciones que permiten obtener el elemento más nuevo que aún no ha sido utilizado, o el resultado de aplicar un filtro FIR⁴ a los 6 elementos más recientes de la cola. Si por problemas de tiempo el *main* se retrasa, pueden haber datos que nunca sean etiquetados como “el dato más nuevo”, ya que se leerá hasta ponerse al día. De cualquier forma, serán tomados en cuenta en el filtro.
- **Modo *FAKE*:** Seteando `IMU_COMM_FAKE` a 1, la biblioteca leerá de un log `ascii` en lugar de utilizar el puerto serie. En el modo *FAKE* los tiempos no son un problema crítico, ya que no correrá en tiempo real.

³Solamente se implementó el modo *hover*.

⁴Los coeficientes del filtro están definidos en *imu_comm_init()*.

1.5. Módulo *kalman*

Aparte de implementar el filtro de Kalman descrito en la sección ??, la biblioteca *kalman* se encarga de suavizar la estimación del ángulo *theta* dada por los acelerómetros y los magnetómetros. El ruido presente en los acelerómetros, sumado a la mala performance del magnetómetro en lugares cerrados⁵, hace que sea necesario utilizar una lógica de suavizado más inteligente que un simple filtrado. Por detalles referirse a `src/kalman/uquad_kalman.c`.

1.6. Driver de los motores y módulo *motor*

La biblioteca *motor* y el driver `src/i2c_beagle/cmd_motores.c` (de ahora en más *cmd*) tienen una fuerte relación, y deben ser coherentes. El driver no se pudo incluir como una biblioteca más, ya que requiere de un encabezado que solamente está disponible en la *BeagleBoard*.

Algunas consideraciones relevantes:

- El *cmd* espera una velocidad superior a cierto mínimo, de lo contrario no arrancará los motores. Este umbral debe estar apareado, de lo contrario *motor* será incapaz de arrancar los motores en el momento apropiado. Luego del arranque, *motor* se encargará de no enviar valores por debajo de los valores definidos como mínimo y máximo. Usar valores por debajo del mínimo puede hacer que se apaguen los motores, y valores por encima del máximo pueden sobrecalentar los contactos de los cables que alimentan a los motores. El máximo también debe estar apareado entre el *cmd* y *motor*. El *cmd* reportará un error en caso de recibir valores fuera de rango.
- Al arrancar los motores, el *cmd* setea velocidades en torno una rampa⁶ desde 0 hasta el valor definido como mínimo, al cual el cuadricóptero no es capaz de levantar vuelo.
- Por cada comando que *motor* envía al *cmd*, este último responde con un *ack*. Así *motor* verifica que el *cmd* está funcionando⁷.

ADVERTENCIA: Cualquier mensaje de error reportado por el *cmd* es motivo suficiente para detener el vuelo y analizar el problema.

1.7. Comunicación

La comunicación con la *BeagleBoard* se hace mediante *ssh*. En el anexo ?? se explica como configurar las partes involucradas.

⁵Se probó en lugares con muchos materiales metálicos, distorsionan las lecturas del magnetómetro.

⁶La implementación son valores que saltan por encima y por debajo de la rampa, esta técnica ha demostrado ser eficiente para hacer arrancar los motores.

⁷Solamente se verifica que hay comunicación, pero la implementación es tal que si la comunicación es exitosa, entonces todo debería estar funcionando correctamente.