CAPÍTULO 1

CÓDIGO

El objetivo de esta sección es explicar como utilizar el código con el que vuela el cuadricóptero.

Para el entender en detalle que hace cada función, referirse a los comentarios en el código fuente.

El código se encuentre en el repositorio git, en la carpeta `src/`. Todas las referencias a archivos en este anexo son respecto a la raíz del repositorio. Todos los programas están pensados para compilarse y ejecutarse en un entorno linux.

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. Por claridad, no se muestran bloques intermedios, utilizados para intercomunicar las distintas partes.

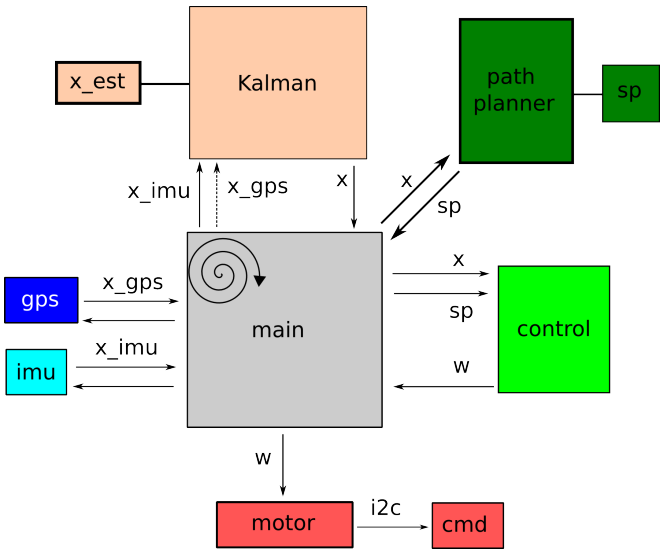

Figura 1.1: Estructura general del código.

Un loop normal, ejecutado por el programa principal (de ahora en más: *main*) consiste en las siguiente etapas:

1. **imu:** Obtener una muestra nueva de la IMU, un dato nuevo de cada sensor.
2. **gps:** Si el GPS tiene un dato nuevo, leerlo.
3. **kalman:** Alimentar el filtro de Kalman con la información actualizada proveniente de los sensores.

4. **path planner:** Comparar el estado actual con el objetivo, y determinar si se completó el objetivo actual. En caso afirmativo, actualizar el estado objetivo.

5. **control:** Usando el estado actual y el estado deseado, determinar la acción de control a realizar.

6. **motor:** Actuar sobre los motores.

Por información relativa a bloques, configuración, compilación, ejecución, etc, referirse a:

```
src/README
```

1.1.1. Git

El código está almacenado en un repositorio git disponible en el DVD adjunto a esta documentación y en *github*: `git://github.com/rlrosa/uquad.git`.

El repositorio ocupa aproximadamente 4GB, por lo que no conviene poner otras cosas en la tarjeta SD de la beagleboard, ya que sino se excederá de su capacidad.

1.1.2. Comunicación

La forma básica de comunicarse con la beagleboard es mediante el puerto serie, usando un conversor *RS2232* a USB. Durante el vuelo, la comunicación se establece mediante *WiFi*, usando un dongle USB. El procedimiento para hacerlo funcionar es el siguiente:

1. Copiar el firmware `scripts/rt73.bin` a la beagleboard en `/lib/firmware/`.

2. Instalar el software necesario haciendo
   `opkg install kernel-module-rt73usb rt73-firmware`.

Una forma de conectarse es usando una red *Ad-Hoc*, también se puede usar un servidor *DHCP*. Para configurar la interfaz `wlan0`[1] para trabajar en modo *Ad-Hoc* agregar las siguiente líneas a `/etc/network/interfaces`:

```
auto wlan0
iface wlan0 inet static
address 10.42.43.2
netmask 255.255.255.0
wireless-mode ad-hoc
wireless-essid uquad
wireless-key s:uquaduquad123
```

Luego, levantar una red en una laptop, en modo *Ad-Hoc*, con nombre *uquad* y contraseña *uquaduquad123* en *WEP 40/128-bit Key(Hex or ASCII)*.

La laptop tendrá la IP *10.42.43.1*, y la beagleboard *10.42.43.2*. El usuario en la beagleboard es *root* y la clave cualquiera. Para conectarse hacer ver la sección 1.1.2

[1] El dongle puede ser asociado a otra interfaz, como *wlan1*, configurar la apropiada.

Proxy

Para trabajar con la beagleboard en el laboratorio de medidas, es necesario configurar el proxy. Para ello, ejecutar los siguiente comandos en la beagleboard:

```
echo "option http_proxy http://httpproxy.fing.edu.uy:3128/" \
>> /etc/network/options.conf
echo "option http_proxy http://httpproxy.fing.edu.uy:3128/" \
>> /etc/opkg/arch.conf
```

ssh

El usuario *root* acepta cualquier contraseña. Para evitar que la beagleboard pida contraseña, se puede agregar la clave pública de una laptop a la beagleboard, en[2].:

```
/home/root/.ssh/authorized_keys
```

Para conectarse hacer:

```
ssh root@10.42.43.2
```

ethernet

La beagleboard sabe conectarse a una red en la que haya un servidor *DHCP*. Elegirle un nombre en `/etc/hostname`, por ejemplo *beagle*, y luego hacer:

```
ssh root@beagle.local
```

1.1.3. Tiempos

El *main* corre sobre linux, lo cual simplifica algunas cosas, pero complica otras. Correr arriba de un sistema operativo que no es RT[3] puede introducir demoras inaceptables. No conviene ejecutar programas ni establecer conexiones (nuevas sesiones *ssh*) mientras se ejecuta el programa principal.

El acceso a memoria no volátil es lento, especialmente el acceso a la tarjeta *micro-SD*. El *main* guarda logs a una memora flash USB, en `/media/sda1/`, ya que es significativamente más rápido que trabajar con la *mirco-SD*. De cualquier forma, si se accede a memoria durante la ejecución del *main*, **NO** será posible tener un loop estable, y por lo tanto **NO** se podrá volar el cuadricóptero. El *main* crea, mediante `fork()`, un *logger* por cada log que se le pida, y dicho programa se encarga de pedir un bloque de RAM donde va almacenando información, y al finalizar el *main* la guarda a memoria.

Nuevamente, la performance del *main* es **INACCEPTABLE** si se accede a memoria o si se abren sesiones ssh durante su ejecución.

[2]Crear el archivo y/o el directorio, si estos no existen.

[3]RealTime.

1.1.4. Consideraciones de seguridad

Al comenzar el *main*, intentará establecer una conexión TCP con un servidor en la laptop, y abortará en caso de no tener éxito. Si en algún momento se pierde la conexión, por ejemplo por algún error en el driver del dongle *WiFi*, el *main* abortará, apagando los motores del cuadricóptero.

En caso de perderse ls conexión ssh, y que el cuadricóptero siga funcionando (porque la conexión TCP sigue abierta), se puede detener el servidor que corre en la laptop, lo cual hará que el *main* aborte y apague los motores.

1.2. Salida - Logs

El *main* genera varios logs durante su ejecución, que sirven determinar la causa de errores o comportamiento extraños durante un vuelo:

- Datos crudos provenientes de la IMU - *imu_raw.log*
- Datos utilizados para estimar el estado - *kalman_in.log*
- Estado estimado - *x_hat.log*
- Acción de control - *w.log*

La primer columna de *imu_raw.log*, *kalman_in.log* y *w.log* es el tiempo, en segundo, desde el comienzo del *main*. La diferencia de tiempo entre *kalman_in.log* y *x_hat.log* es despreciable, por lo que *x_hat.log* no tiene columna de tiempos. Referirse al código fuente por información sobre como habilitar logs, nombres, contenido, etc.

Cabe destacar que el espacio en RAM utilizado por el *logger* es limitado, y dejará de guardar información una vez que se acabe.

1.3. Debugging

Para probar el *main* sin volar el cuadricóptero, puede resultar cómodo apagar el control de connectividad. Para ello, setear `CHECK_NET_BYPASS` a 1 en

`src/common/uquad_config.h`

ADVERTENCIA: **NO** es recomendable hacer esto para una prueba con los motores prendidos, ya que la pérdida conexión implicaría la pérdida del control sobre el cuadricóptero.

1.3.1. Ejecución a partir de un log

Para debuggear el *main*, se lo puede correr con un log conocido como entrada, lo cual permite analizar el efecto de cambios concretos. Para esto se debe configurar el módulo que lee de la IMU, avisándole que debe leer de un log. Esto se logra seteando `IMU_COMM_FAKE` a 1, en `src/imu/imu_comm.h`.

C Vs. MatLab

Existe un script en *MatLab* que reproduce el comportamiento del *main*. Es muy útil para hacer pruebas y verificar el correcto funcionamiento del código en `C`.

```
kalman/kalman_main.m
```

Como argumento toma un log de datos crudos de la imu (*imu_raw.log*), como los generados por el *main* o por *imu_comm_test*:

```
src/test/imu_comm_test/imu_comm_test.c
```

1.4. Kernel

La beagleboard corre linux *2.6.37*, de la distribución *Angstrom*:

- `http://www.angstrom-distribution.org/`

El kernel que viene por defecto es suficiente para *casi* todo, hace falta configurar el puerto *i2c-2* para que trabaje a 333kHz (en lugar de 400kHz).

Para compilar el kernel se utilizó *Bitbake+OpenEmbedded*, y se lo compiló desde *Ubuntu* 11.10 (64bits).

La información que se presenta a continuación se obtuvo de [**?**], [**?**] y del canal IRC `#oe`.

1.4.1. *Compilación - Bitbake+OpenEmbedded*

Para poder compilar programas para la beagleboard se puede usar un entorno de desarrollo como OpenEmbedded (de ahora en más *OE*) y la herramienta para compilar, *bitbake*. La herramienta *bitbake* maneja recetas que listan programas y sus dependencias, y se encarga de compilar las cosas en el orden apropiado.

Todo lo que usa *OE* se baja de internet mediante *git*. A veces hay problemas con los servidores, y es cuestión de probar nuevamente en otro momento (o cambiar de servidor, revisar proxy, etc). Compilar una imagen entera, como para una SD, no es fácil, lleva tiempo y requiere que todos los servidores funcionen, y un poco de suerte.

Para compilar en *Ubuntu*:

1. Ejecutar `sudo dpkg-reconfigure dash` y en el menu elegir la opción *no*. Por más información ver `http://wiki.openembedded.org/index.php/OEandYourDistro`.

2. Descargar, configurar y actualizar el repositorio:

```
git clone git://git.angstrom-distribution.org/setup-scripts
cd startup-scripts
MACHINE=beagleboard ./oebb.sh config beagleboard
MACHINE=beagleboard ./oebb.sh update
source ~/.oe/environment-oecore
```

3. Si se quisiera compilar ahora[4], hacer:

[4]Compilar la imagen lleva mucho tiempo, mejor configurar todo antes de compilar.

```
bitbake console-image
```

El script `~/.oe/environment-oecore` es responsable de generar variables que se utilizan durante la compilación. Cada vez que se abre una consola se cargan las variables globales y las declaradas en `~/.bashrc`, una almacena las rutas a los ejecutables instalados, otra las bibliotecas, etc. Las variables en `~/.oe/environment-oecore` hay que cargarlas cada vez que se abre una nueva consola. Una de las cosas que hace el script es indicar la ruta al programa *bitbake*. Para verificar que el script fue correctamente ejecutado se puede escribir el comienzo del comando `bit` y apretar *tab* ver sugerencias. Entre las opciones debería aparecer el comando *bitbake*.

Las recetas que utiliza *bitbake* (y nos interesan más) se encuentran en:

```
setup_scripts/sources/meta-ti/
```

Algunos ejemplos:

- Configuración del kernel:

```
setup-scripts/sources/meta-ti/recipes-kernel/\
linux/linux-3.0/beagleboard/defconfig
```

- Receta para el kernel:

```
setup_scripts/sources/meta-ti/recipes-kernel/\
linux/linux-omap_3.0.bb
```

- Receta para el u-boot:

```
setup-scripts/sources/meta-ti/recipes-bsp/\
u-boot/u-boot_2011.12.bb
```

Luego de haber incorporado las variables de `~/.oe/environment-oecore` ya no es necesario usar `MACHINE=beagleboard ./oebb.sh`, se puede y debe usar directamente *bitbake*.

Es recomendable hacer `MACHINE=beagleboard ./oebb.sh update` frecuentemente. Algunos paquetes necesarios para poder compilar correctamente (pueden faltar otros):

```
sudo apt-get install\
   python-ply python-progressbar\
   texi2html cvs subversion gawk\
   chrpath texinfo diffstat
```

Como modificar el contenido de OE

Para modificar un programa, como por ejemplo el *u-boot*:

```
bitbake -c devshell u-boot
# se abre una consola en el dir temporal del u-boot
emacs board/ti/beagle/beagle.h
# editar, por ejemplo habilitar la UART2
git add board/ti/beagle/beagle.h
git commit -m 'uquad: habilitando UART2'
```

```
git format-patch HEAD~1
cp 001-uquad:-habilitando-UART2.patch $OE_BASE/
# incrementar la línea que dice "PR = "r4"", ponerle r5
bitbake u-boot
# buscar el resultado en setup-scripts/build/tmp*/deploy/images
```

Para modificar el kernel y setear el *i2c-2* a 333kHz:

```
bitbake -c devshell virtual/kernel
# se abre una cosola en el dir temporal del kernel
# ANTES de cambiar nada, hacer:
quilt new uquad-set-i2c-2-333kHz.patch
# Si se quiere hacer cambios en board-omap3beagle.c:
quilt add arch/arm/mach-omap2/board-omap3beagle.c
emacs arch/arm/mach-omap2/board-omap3beagle.c
# editar, por ejemplo setear i2c a 333kHz
# Ahora pedirle a quilt que arme un patch
quilt refresh
# El patch queda en patches/uquad-set-i2c-2-333kHz.patch
# Se copia a meta-ti/recipes-kernel/linux-3.0/
# Se edita meta-ti/recipes-kernel/linux_3.0.bb, agregando una
# línea antes de la q dice defconfig:
file://uquad-set-i2c-2-333kHz.patch;patch=1 \
# Ahora se compila haciendo
bitbake virtual/kernel
```

Comandos útiles - *bitbake*+OE

Algunos comando que pueden ser de utilidad:

- Para compilar un paquete individual, sin tomar en cuenta las dependencias:

  ```
  bitbake -b receta.bb
  ```

  Ejemplo:

  ```
  bitbake -b sources/meta-ti/recipes-bsp/u-boot/u-boot_2011.12.bb
  ```

- Para ver todas las recetas que se ejecutan como dependencias, hacer:

  ```
  bitbake <receta> -g
  ```

  y luego mirar en `task-depends.dot`.
  Ejemplo:

  ```
  bitbake console-base-image -g
  ```

- Para borrar todo lo compilado sobre un paquete, por ejemplo el u-boot, hacer:

  ```
  bitbake -c clean u-boot
  ```

1.5. IMU

La Mongoose viene cargada con un *bootloader Arduino*, que permite bajarle código mediante el puerto serie.

El código original que trae la Mongoose fue modificado. El problema principal que tenía era que la frecuencia de muestreo no era estable.

Algunos cambios:

- Se implementó transmisión de datos en segundo plano, mediante interrupciones, evitando trancar el loop principal al momento de transmitir.
- Se agregó la posibilidad transmitir en binario (se mantuvo el modo `ASCII`, pero no es posible transmitir a 10ms en dicho modo).
- El barómetro demora varios milisegundos entre que se le pide un dato y que lo tiene disponible. El código original esperaba durante este tiempo. La versión modificada sigue trabajando y vuelve para recoger el dato luego que transcurrió el tiempo en cuestión.
- Se modificó el formato de los datos envíados al puerto serie.

1.5.1. Compilación

Para programar la Mongoose se utiliza *Arduino*. Para instalarlo hacer:

```
sudo apt-get install arduino
```

El código utiliza una biblioteca que no viene con *Arduino*. Asumiendo que el repositorio git fue descargado a `~/uquad`, para agregar la biblioteca al *Arduino*, hacer:

```
sudo ln -s ${HOME}/uquad/src/mongoose_fw/HMC58X3/\
 /usr/share/arduino/libraries/HMC58X3
```

1.6. Trabajo a futuro

- **Control:** La matriz de control se carga de un archivo de texto. Para automatizar el cálculo de dicha matriz, permitiendo cambiar de trayectoria en tiempo real, habría que implementar el cálculo en `C`. Está lista una implementación del algoritmo LQR en `C`, pero falta implementar correctamente la linealización del sistema según la trayectoria elegida.
- **Path planner:** Solamente está implementada la modalidad de *hovering*. Queda pendiente implementar rectas y círculos, y un sistema de *waypoints* que permita ir recorriendo trayectorias.
- **Visión:** Queda pendiente implementar en `C` el algoritmo descrito en **??**.
- **logger:** Una mejora sería que el *logger* guarde información en binario en lugar de usar `ASCII`, lo que permitiría ahorrar RAM. De cualquier forma, el RAM no es un problema por ahora.