How our MADS plugin works

Questo documento serve per comprendere meglio la struttura del plugin che abbiamo creato per il corso di robotic perception and action. Per prima cosa, stabiliamo con quali file stiamo lavorando e a cosa servono:

- run_all.sh
- replay.cpp
- rerunner.cpp
- mads.ini
- odometry_filter.cpp
- visualizer.py

La struttura MADS, lavora come un ros, con alcune modifiche. Un controllo centrale (mads broker) gestisce tutti I file che vengono richiamati nell'esecuzione. Il flusso dati parte da una sorgente (source) tramite un comando "mads source replay.plugin -n …." da cui il broker si piglia I dati. Questi possono essere passati ad un filtro "mads filter odometry_filter.plugin -n …." che li elabora e a sua volta restituisce dei dati, che vengono raccolti da un visualizzatori python. Questo visualizzatore utilizza la classe rerun per ricreare tramite interfaccia video I dati elaborati.

Passiamo quindi a vedere le parti importante del codice per capire come e perchè funziona.

run_all.sh

Questo file si comporta come un terminale e può controllare la presenza di pacchetti essenziali per l'esecuzione (se provate ad aprirlo vedrete degli if).
Se avete puntato correttamente il percorso del file mads.ini del vostro percorso locale a quello interno di esecuzione di mads (spiegato nella user guide), run_all.sh controlla per voi se avete buildato I plugin ed esegue il codice.

Cosa fa?

In una prima fase, pulisce tutti I processi mads che potrebbero essere rimasti in esecuzione.
Fa partire il visualizzatore ed il broker di mads. Infine, esegue tutte le sorgenti (4 se c'è anche l'htc).
Il flusso di dati quindi proviene da possibilmente 4 diversi file in contemporanea, ecco perchè è importante allineare I tempi in cui arrivano I dati. Questo non si tratta solamente di allineare il timecode, ma di rallentare equamente il processo di lettura del file. Un piccolo esempio, I dati dell'encoder potrebbero essere letti a 100k righe/s, mentre quelli della realsense solamente a 5000 righe/s. Ipotizzando che I due sensori abbiano una frequenza simile, il mio codice ha già finito di leggere tutti I dati dell'encoder, mentre quelli della realsense no. Il codice che segue dopo la lettura dei dati può solamente salvare il dato precedente a quello nuovo che arriva, quindi la mia simulazione si rompe.

SI tratta questo problema all'interno di replay.cpp

replay.cpp

Questo file è stato creato semplicemente seguendo la guida che si trova per I plugin sul sito di MADS. È stato aggiunta questa sezione:

```
if (flat.contains("/message/timecode") &&
flat["/message/timecode"].is_number()){
current_sim_time = flat["/message/timecode"].get<double>();
time_found = true;
```

```
}
// search for "timecode"
else if (flat.contains("/timecode") && flat["/timecode"].is_number()) {
current_sim_time = flat["/timecode"].get<double>();
time_found = true;
}

if(time_found){
if (_first_packet){
_initial_sim_time = current_sim_time;
_start_real_time = std::chrono::steady_clock::now();
_first_packet = false;
} else {
double dt_sim = current_sim_time - _initial_sim_time;
auto dt_duration = std::chrono::microseconds(static_cast<long long>(dt_sim *
1e6));
auto target_time = _start_real_time + dt_duration;
// Sleep until target time
std::this_thread::sleep_until(target_time);
}
}
```

La prima parte è una ricerca all'interno dei dati, controllo se esiste il percorso /*message/*timecode o /timecode all'interno di essi per salvarmi il tempo attuale.
La condizione (first_packer) è per Il primissimo dato che viene passato, dato che si controlla successivamente la differenza dei tempi tra due pacchetti di dati.

REMINDER : viene lanciato un source per ogni file di dati, quindi si controllano I tempi con sè stessi, non con gli altri sensori. Con gli altri sensori si è già allineati in quanto registri la stessa simulazione in contemporanea.

Dopo aver ricevuto il primo dato, al sucessivo, controllo la differenza dei due timecode e faccio dormire il sistema fino a quel tempo. In questo modo faccio rispettare I tempi in cui sono stati rilevati I dati. (in parole povere rallento la lettura)

Questo file potrebbe essere modificato, in quanto il controllo del percorso /timecode nel dato di lettura probabilmente è inutile

rerunner.cpp

Non ho modificato nulla di questo file, l'ho importato seguendo le guide.

Mads.ini

Questo file è molto importante per MADS e contiene il nome, I parametri e gli indirizzi in cui andare a pescarsi I vari plugin.

Se lo aprite, troverete molti percorsi file, questo perchè I prof han voluto caricarci una miriade di file lol.

Questo file è parecchio utile per I vari parametri perchè è possibile modificare e salvare mads.ini e runnare direttamente la simulazione senza dover buildare nuovamente il plugin modificato.

Al suo interno ho salvato I valori di tutte le variabili che sono state calcolate e le configurazioni del sistema che sono da cambiare a seconda del dataset che leggiamo 1st dataset ad esempio funziona con la aruco che riporta I dati secondo il centro del qr code, invece che il centro del walker.

Odometry_filter.cpp

Passiamo al file più importante: il nostro plugin. Per questioni di facilità di lettura, chiedo di aprire a lato di questo documento il codice, senza riportarvelo qui. (So che siete bravissimi e lo avevate già fatto)

Tralasciando tutti I vari include iniziali, Ho creato un po' di funzioni/classi utili: MovingAverage, Odometry_filterPlugin ekf_predict, std_dev, normalize_angle e ekf_update.

Nel codice, MovingAverage è posizionato sopra la classe odometry_filterPlugin, mentre le funzioni sono all'interno della classe principale, nella sua sezione public.

MovingAverage è una classe che permette di utilizzare una media a finestra mobile, utile per pulire I dati, ma aggiunge un ritardo.

Odometry_filterPlugin serve a ridefinire le dimensioni delle finestre mobili che ho usato

efk_predict e update sono le funzioni per il filtro kalman.

std_dev calcola la deviazione standard di un vettore di dati in input (usato per la calibrazione in fase stazionaria)

normalize_angle permette di mantenere l'angolo tra – e + pi

All'interno della classe Odometry_filterPlugin, alcuni dati sono definiti all'interno di strutture mentre altri no. Quelli all'interno delle strutture, vengono ridefiniti alla fine del file da mads.ini. Questo è stato fatto per raggruppare meglio I dati, non so se era la soluzione più efficace.

Arriviamo quindi all'elaborazione dati, partiamo da come li leggiamo:

return_type load_data

Dovete immaginare che questo filtro riceve I dati in contemporanea da tutte le sorgenti che abbiamo avviato in precedenza, quindi devo capire cosa mi sta arrivando e dove salvarlo.
Questa parte di codice si occupa di controllare il nome agente in arrivo e salvare I dati.

- Per primo salvo gli encoders inizializzando I tempi se è il primo pacchetto.
Mi salvo I tick ed il timecode a cui mi è arrivato.

- Estraggo la posizione dell'HTC (nota: se non ho l'HTC, il controllo dell'if non si avvera, quindi non ho problemi di esecuzione se mi manca un file sorgente). Ricavo posizione e angolo.

- Dati della IMU: creo delle variabili temporanee in cui salvo I valori corrispondendi dopo aver applicato la matrice di rotazione.
Un problema della Imu è che è applicata alla fronte del walker e nelle rotazioni risente di forze centripete e tangenziali.

Determino accelerazione angolare per ricavare il termine tangenziale. Calcolo quello centripeto e infine posso correggere il valore rilevato per ottenere quello effettivo.
In aggiunta, all'inizio della simulazione vi è sempre una fase stazionaria, la sfrutto per ottenere il bias dell'accelerometro. Stampo il bias alla fine del calcolo.

Infine, aggiorno le varibili che utilizzerò nella elaborazione.

- Aruco/Realsense (non ho ancora capito come si chiama)

Leggo I dati (I due dataset nuovi utilizzano un formato leggermente diverso da quello originale)
Con questo sensore abbiamo problemi di "buchi" nei dati perchè la aruco perde di vista il suo qr code. Per fortuna, quando questo accade, il sensore ritorna esattamente lo stesso valore dell'ultimo rilevato, quindi possiamo filtrare fuori I dati errati controllando la differenza con l'ultimo valore registrato.

Ora entra la logica del "il sensore riporta I dati secondo il centro del walker o secondo il centro del suo pannello?" Se è il centro del walker, aggiorna la posizione e normalizza l'angolo. Altrimenti, aggiunge una rotazione di 180° e applica una matrice di rototraslazione alla posizione per riprotarci al centro del walker.

_ekf_theta_rs è l'angolo che daremo all'aggiornamento del filtro kalman, quello che viene fornito grezzo dall'aruco è già corretto, serve solamente normalizzarlo.

Dopo aver letto I dati, c'è la sezione di calibrazione (non viene più utilizzata, ma serve per far vedere che l'abbiamo fatta) e, usando I dataset statici, ritornano I valori delle sigma dei vari sensori (IMU e ARUCO).

return_type process

Entriamo nella parte importante del codice.

SE riceviamo un aggiornamento dall'encoder, si procede con la predizione ecc, altrimenti si aspetta il dato, a prescindere dagli altri dati che riceviamo. Noi ci stiamo basando sul movimento dell'odometria, controllando con imu per eventuali slittamenti, controllando l'orientamento tramite giroscopio e aggiornando la posizione corretta con l'aruco.

La prima parte // -- prediction--- è l'odometria pura, da cui ricavo lo spostamento rilevato dall'encoder.
Da questo, derivo ed applico un filtro passa basso, derivo nuovamente ed applico un secondo filtro passabasso e salvo il dato tramite media mobile.

Questo doppio filtro può sembrare eccessivo, ma stiamo cercando di derivare due volte un dato altamente discreto. Senza controlli forti, l'accelerazione schizza alle stelle.

Questi filtri creano un ritardo nell'output, se confrontato con l'accelerometro, pertanto, applico I medesimi filtri ai dati dell'acc dell'imu.

Segue Il controllo slittamento in base all'accelerazione rilevata dei due sensori. Se la differenza è sostanziale, si controlla se l'accelerazione rilevata dalla imu non è 0, il che corrisponde a velocità costante (spostamento=spostamento precedente), altrimenti riduco lo spostamento del rapporto delle due accelerazioni.

In seguito, si vuole ricavare la stima dell'angolo tra imu e encoder utilzzando le sigma ricavate in fase di calibrazione. (attenzione, all'encoder ho applicato una sigma elevata in quanto si sa che è rumorosa)

Ricapitolando: ho ottenuto la stima dello spostamento che ho fatto nell'ultima frazione di tempo, ho ottenuto l'angolo di direzione in cui sto andando.

Ora calcolo l'incertezza sull'ultimo spostamento (con la flag is_slipping) e sull'angolo.

Fornitsco I dati ottenuti dentro alla predizione del kalman, il quale mi aggiorna lo stato di posizione e direzione della variabile di stato.

All'interno del codice, troverete delle varianti tipo "odometria pura" che servono a vedere la traiettoria tracciata mediante diverse configurazioni di sensori: solo encoder, encoder+imu, tutto insieme.

Se ho a disposizione il dato della realsense, aggiorno il vettore posizione e direzione Z e lo passo all'update del kalman.

Segue la parte finale del codice in cui, se è presente l'htc, si confronta la posizione del kalman con quella effettiva dell'htc, si ricavano le distanze attuali e l'errore quadratico medio.

La parte finale stampa in out tutti I dati di cui ho bisogno:
- stima posizione mediante encoder
- stima posizone encoder + imu
- stima posizione completa
- debug (accelerazioni, angoli.. tutto quello che era utile per capire come far funzionare meglio le cose)

stampa a video ogni 200 valori di aruco dell'errore quadratico della posizione e dell'angolo (angolo purtroppo sbagliato perchè la normalizzazione fa un cesso al momento)

void set_params

importa tutti I parametri che ho salvato dentro mads.ini

visualizer.py

Invece che utilizzare direttamente Il rerunner plugin, ho usato (grazie gemini) un visualizzatore python.

In parole povere, si allaccia all'host sul quale lavora il broker ed ascolta per I tag impostati.

La funzione get_nested serve a leggere I dati (ad esempio ricevo "/message/pose/position" legge quante "/" ci sono e naviga al suo interno fino ad arrivare alla parola "position" dove voglio estrarre il dato. Se invece ho direttamente il dato col nome giusto, usa quello)

Inizializzo rerun, mi collego a mads ed ai topics.

Controllo il dato ricevuto tramite il topic a cui sono collegato e successivamente con il tag che ho assegnato in out dentro al filtro. Controllo che non sia vuoto o nullo e stampo a video. Se è una traiettoria creando dentro al visualizzatore una traiettoria e definendo le sue caratteristiche tramite rr.Points3D e rr.LineStrips3D. Se è un grafico mediante Scalars.

Ci sono parecchi dettagli che ho probabilmente saltato e non ho parlato di tutta l'evoluzione che c'è stata dietro. Questo è il mio risultato finale.