ALMA MATER STUDIORUM
UNIVERSITA' DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
Sede di Forlì

Corso di Laurea in
INGEGNERIA AEROSPAZIALE
Classe L-9

ELABORATO FINALE DI LAUREA

in SATELLITI E MISSIONI SPAZIALI

PROGETTO ED IMPLEMENTAZIONE DI STRUMENTI SOFTWARE DEDICATI AL MEMORY BUDGET DI PIATTAFORME SATELLITARI

CANDIDATO
Jacopo Villa

RELATORE
Paolo Tortora

CORRELATORE
Gilles Mariotti

Anno Accademico 2016/2017

Alla mamma Dany e al babbo Max,
grazie ai quali tutto questo è stato possibile.

A mio fratello Brando,
perché i tuoi sogni siano sempre la tua bussola.

Indice

Indice delle figure

Indice delle tabelle

1. CAPITOLO 1
INTRODUZIONE AL MEMORY BUDGET

1.1 Introduzione

Nel progetto di una missione spaziale satellitare è di fondamentale importanza la valutazione della capacità della piattaforma satellitare di rispettare i requisiti di missione. Questa valutazione passa (anche) attraverso la definizione di una serie di bilanci (o *budget*), che descrivono le prestazioni ed i limiti operativi della piattaforma e permettono di verificare se lo spacecraft che si sta progettando è effettivamente in grado di rispettare i suddetti requisiti. La definizione di questi budget interessa tutte le fasi di missione, dalla fase di studio di fattibilità fino a quella di definizione delle operazioni. Alcuni esempi di questi bilanci sono:

- *Mass budget* – stima le quantità e la distribuzione di massa all'interno del veicolo, così da definirne l'inerzia, ingombri, peso.
- *Power budget* – quantifica il bilancio tra l'energia elettrica fornita dai sistemi di generazione a bordo (es. pannelli solari o RTG) e quella richiesta per alimentare i vari sottosistemi della piattaforma ed i payload.
- *Link budget* – definisce i requisiti necessari per il sistema di telecomunicazioni, partendo da informazioni sull'orbita, sulle stazioni di terra e sui componenti utilizzati a terra e/o a bordo.
- *Memory budget* – dimensiona i componenti che permettono di immagazzinare e trasmettere dati, prima all'interno del satellite e poi verso le stazioni di terra.

Per effettuare la maggior parte dei budget, è necessario conoscere i parametri orbitali e la loro evoluzione nel corso della missione. Infatti, poiché tra questi dati sono comprese le grandezze fisiche in ingresso nei bilanci stessi, essi rappresentano le fondamenta su cui vengono sviluppata tutta la serie di analisi successive.

Alcune delle informazioni che si possono estrapolare dall'evoluzione di un'orbita sono, a titolo di esempio: quota, inclinazione orbitale, traccia a terra, condizione di luminosità (luce, *umbra*, *penumbra*), posizione relativa satellite-stazioni di terra e così via. Per quanto detto, una delle prime fasi dell'analisi di una missione consiste nella determinazione dei dati orbitali riferiti alla traiettoria del satellite in oggetto. Nelle primissime fasi della missione vengono solitamente stilati budget basati su parametri orbitali di tentativo, tracciati a grandi linee in base al tipo di missione che si sta andando

a progettare. Via via che il progetto di missione avanza verso stadi più di dettaglio, anche i budget di sistema vengono raffinati diventando alle volte delle vere e proprie simulazioni. A tal proposito, gli input orbitali possono essere generati usando dei cosiddetti propagatori orbitali che, una volta definite le condizioni iniziali della missione, calcolano la traiettoria del satellite attraverso metodi numerici. In questo caso, le simulazioni di budget possono essere definite come un vero e proprio *post-processing* dei dati orbitali di cui sopra.
Il lavoro svolto in fase di tesi, e descritto nel presente lavoro, può essere localizzato nel processo appena descritto: partendo dalla propagazione orbitale si è realizzato, attraverso lo sviluppo completo di un software dedicato, un post-processing dei dati volto alla definizione del memory budget di una piattaforma micro-satellitare, per trarre infine delle conclusioni sul dimensionamento dei componenti.

1.2 Introduzione a Sitael S.p.A.

Fig. 1.1 – Logo aziendale di Sitael S.p.A.

Il lavoro di tesi è stato svolto presso Sitael S.p.A., azienda italiana che opera in campo scientifico, dell'elettronica industriale, dell'Internet of Things oltre che in ambito spaziale, focalizzato nel settore dei piccoli satelliti (massa inferiore ai 200 kg), in particolare per missioni in orbita bassa. Sitael propone ricerche innovative quali la propulsione elettrica, oltre ad occuparsi di propulsione chimica, strumentazioni ed avionica, servizi di osservazione della Terra e payload scientifici, vantando una qualità certificata dagli standard ESA e NASA.

1.3 Obiettivo del lavoro di tesi

Per le operazioni pertinenti alla fase di *post-processing*, la divisione *Mission Analysis* di Sitael si serve ad oggi di un software proprietario in fase di sviluppo, scritto in linguaggio MATLAB. Grazie ad esso, è possibile importare i dati in uscita da un propagatore orbitale (ad esempio GMAT - si veda Cap. 1.6.1), che sono raccolti in un documento appositamente formattato. Una volta effettuato il caricamento dei dati nel programma, è possibile svolgere una serie di analisi che portano a risultati

numerici e/o grafici (sotto forma di diagrammi). Nel programma è presente, ad esempio, la *subroutine* dedicata al *power budget.* Dato l'interesse di Sitael nell'implementazione di una funzionalità analoga dedicata al *memory budget* all'interno del software, per poterne così espandere le potenzialità, nel lavoro di tesi si è preso in carico lo sviluppo di un tale programma. In particolare, l'obiettivo della tesi si identifica con lo sviluppo di un software dedicato al memory/data budget per una missione spaziale in fase A, B e C di categoria microsatellite, attraverso la modellizzazione del bus di scambio dati della piattaforma comprensiva di sottosistemi, payload, bus, memorie e trasmettitori.

Nel periodo in cui è stato svolto il lavoro, Sitael è stata direttamente impegnata nella missione ESEO (si vede il seguente Cap. 1.4) e si è scelto in fase di sviluppo che la validazione finale del software realizzato durante la tesi venisse effettuata proprio attraverso la simulazione del memory budget di tale missione. Pertanto si può dire che l'obiettivo finale del lavoro si identifichi con la stima, attraverso il programma sviluppato, del memory budget della missione sopracitata.

Il lavoro di tesi è stato preceduto da un'attività di tirocinio ad esso propedeutica, che ha permesso di sviluppare le conoscenze e le competenze richieste per il suo completamento (si veda il paragrafo 1.7).

1.4 Introduzione alla missione ESEO

ESEO (European Student Earth Orbiter) è un programma dell'Education Office di ESA che vede impegnate alcune università europee, tra cui l'Università di Bologna, i cui studenti lavorano sullo sviluppo delle tecnologie da integrare a bordo di un satellite di classe micro (massa circa 50kg) in qualità di payload. Sitael funge da partner commerciale per lo sviluppo della piattaforma satellitare. La missione consiste in un satellite (in Fig. 1.2) da collocare ed operare su di un'orbita bassa (LEO) ed eliosincrona. Tra i payload figurano una microcamera, due sensori di radiazioni e diverse tecnologie tra cui un ricevitore GPS, un trasmettitore in banda S e un dispositivo di de-orbiting (la cui funzione è garantire, a fine vita operativa, il rientro del satellite che verrà disintegrato in atmosfera).

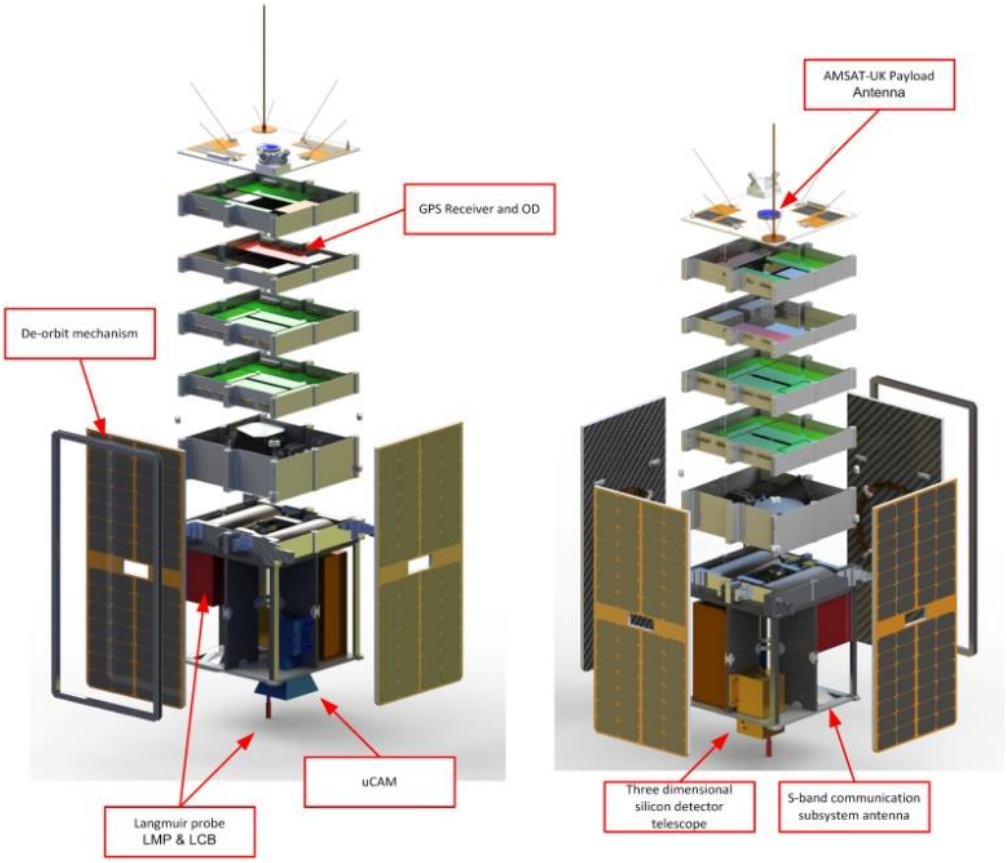

Fig. 1.2 - La vista esplosa di ESEO

1.5 Il memory budget

Come già accennato, il *memory budget* si identifica con l'insieme di operazioni che determinano i flussi di dati all'interno del satellite, in particolare la quantificazione dei dati accumulati nelle memorie e nei *buffer* dei sensori. Esso ha come fine ultimo il dimensionamento dei componenti dedicati all'immagazzinamento ed alla trasmissione dei dati stessi.

1.5.1 L'importanza del memory budget

Nella maggior parte dei casi, l'obiettivo di una missione spaziale satellitare in orbita terrestre è quello di raccogliere, grazie al satellite che funge da tramite, una o più tipologie di informazioni. Queste possono essere di interesse scientifico, meteorologico, di carattere commerciale (ad es. per le telecomunicazioni), bellico e così via.

Ne consegue che gli strumenti ed i sensori a bordo del satellite (il cosiddetto *payload*), necessari per l'adempimento della missione, sono strettamente collegati alla tipologia della missione e sono quindi fortemente variabili, da un satellite ad un altro, in numero, prestazioni e funzionalità.

A prescindere dal campo di applicazione, il memory budget è comunque fondamentale perché la piattaforma stessa deve essere in grado di generare e trasmettere una serie di dati di *housekeeping*[1] che informano gli operatori sullo stato di salute della missione.
In generale, il memory budget serve ad assicurarsi che vengano soddisfatti, per tutto l'inviluppo della missione, due requisiti fondamentali:

- I dati non devono saturare le memorie di bordo, ed eventuali memorie interne dei payload
- Tutti i dati immagazzinati a bordo devono essere trasmessi verso la/le stazione/i di terra

Qualora uno di questi punti non venisse soddisfatto, una parte delle informazioni potrebbe andare perduta, compromettendo il successo della missione.
All'interno di un satellite, l'informazione viaggia e viene immagazzinata sotto forma di bit. Pertanto, è opportuno introdurre due grandezze fondamentali per modellizzare le operazioni svolte con i bit: il *Data Rate* ed il *Data Volume*.
Si definisce *Data Rate* il prodotto:

$$Data\ Rate = Sample\ Rate * Bit\ per\ Sample$$

in cui con *Sample Rate* si intende la frequenza (dimensionalmente Hz, s^{-1}) con cui avvengono i campionamenti da parte di un determinato strumento, mentre i *Bit per Sample* (misurati in *bit*) sono la quantità di bit presente in ogni singolo campionamento. Le dimensioni del *Data Rate* sono dunque *bit/s* (solitamente indicate con *bps*).
Per valutare la mole di dati trasmessi o immagazzinati in un certo tempo Δt, si definisce il *Data Volume*, con la relazione:

$$Data\ Volume = Data\ Rate * \Delta t$$

Quest'ultimo ha come unità di misura il *bit*.

[1] Vengono definiti parametri di *housekeeping* tutti quei dati, contenuti nella telemetria inviata dal satellite, che caratterizzano lo stato di salute del satellite stesso. Alcuni esempi sono tensioni elettriche, correnti e temperature, che vengono rilevati all'interno del satellite grazie ad appositi sensori e che devono rimanere entro determinati range di sicurezza.

1.5.2 La dinamica di scambio dati in un satellite

Lo scambio di dati nelle piattaforme satellitari, su un singolo ramo, è schematizzabile come in Fig. 1.3:

- La sorgente dei dati è un sensore che acquisisce informazioni (e quindi bit) e le immagazzina inizialmente all'interno di una memoria interna del sensore, detta *buffer*. Il *buffer* garantisce che non vi siano perdite di dati in caso di differenza di velocità tra il trasferimento a valle e l'acquisizione a monte del sensore.
- I dati vengono trasmessi, attraverso un collegamento detto *bus*, ad una memoria esterna che solitamente vanta una capacità maggiore di quella del *buffer*.
- Nel momento in cui il satellite si trova in una posizione dell'orbita tale per cui una o più stazioni di terra sono in vista, i dati immagazzinati nella memoria vengono inoltrati, tramite un secondo *bus*, al trasmettitore. Quest'ultimo, attraverso opportune codifiche e modulazioni, converte l'informazione in onde elettromagnetiche sotto forma di segnale in radiofrequenza e la trasmette alla/e stazione di terra.

  Per semplicità, in Fig. 1.3 è riportato un unico ramo che parte dall'informazione e giunge alla conversione in segnale. Tuttavia, come si vedrà nel seguito, sono di uso comune architetture con un numero maggiore di sensori, *bus*, memorie e trasmettitori, collegati tra loro in modo più complesso, in modo tale da soddisfare i requisiti di missione. Si può osservare che ogni componente del diagramma a blocchi è soggetta ad entrate ed uscite di dati, la cui entità dipende, come verrà approfondito, dalle caratteristiche del satellite, dai requisiti di missione, dalla tipologia di orbita e così via.

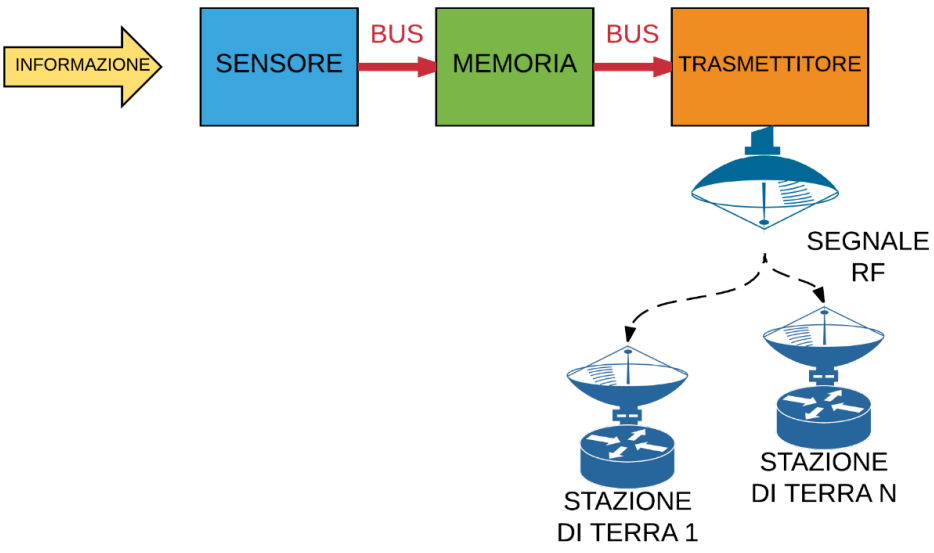

Fig. 1.3 – Diagramma di flusso concettuale sul movimento dei dati in un satellite

1.5.3 Sensori

Ad introdurre i dati a bordo sono i sensori, il cui ruolo è interfacciare il satellite con l'ambiente esterno e raccogliere le informazioni necessarie per il caso specifico. Essi possono essere parte del *payload* (ad es. strumenti di osservazione della Terra) oppure strumenti necessari al mantenimento delle condizioni ottimali per il satellite (ad es. strumenti per la determinazione d'assetto e/o sensori che raccolgono i cosiddetti dati di *housekeeping*[2]), che vengono poi trasmessi a terra attraverso la telemetria. A seconda della propria funzionalità e dei requisiti di missione, l'entità (*Bit per Sample*) e la frequenza (*Sample Rate*) di campionamento possono variare considerevolmente, dando origine a *Data Rate* molto variabili tra loro. Come regola generale, si può pensare che il *Data Rate* relativo alla telemetria sia sempre di qualche ordine di grandezza minore del *Data Rate* proveniente dal *payload.*

1.5.4 Bus

I *bus* rappresentano l'elemento attraverso cui è possibile scambiare dati all'interno del satellite, passando da un componente ad un altro. Concettualmente, essi sono rappresentati in Fig. 1.3 dalle frecce rosse che collegano i vari blocchi. Nel caso in oggetto, il flusso di dati passante nei *bus* è monodirezionale, partendo da monte (dai sensori) e procedendo verso valle (i trasmettitori). Così come per i sensori, anche per i *bus* esistono varie categorie, caratterizzate da diverse velocità di trasmissione dati, un diverso tipo di *overhead*[3] e di compressione[4] (se presenti), e in generale diversi protocolli. Tra i protocolli più diffusi è presente il CAN-bus[5], utilizzato anche per il satellite della missione ESEO. In campo spaziale, i *bus* utilizzati devono soddisfare stringenti requisiti e norme dedite a certificare che i componenti possano mantenere un certo livello di performance anche nello spazio e per un periodo più o meno prolungato.

[3] L'overhead consiste in una serie di informazioni aggiuntive che è necessario fornire al ricevente (umano o macchina che sia) per specificare informazioni ulteriori sui dati ricevuti. Per ulteriori dettagli, si veda il Par. 2.6.
[4] La compressione è un processo che, tramite opportuni algoritmi, permette di ridurre la quantità di dati richiesta per trasmettere un certo numero di informazioni. Si vedano gli approfondimenti nel Par. 2.6.
[5] Il protocollo CAN, invenzione della Robert Bosch GmbH, è nato in ambito automotive ma viene ad oggi impiegato per una vasta serie di applicazioni, tra cui i sistemi spaziali. Ha il vantaggio di essere più economico rispetto ai principali protocolli utilizzati in ambito satellitare; vanta inoltre una bassissima sensibilità ai disturbi elettromagnetici.

1.5.5 Memorie

Per satelliti in orbita bassa (LEO[6]) o più in generale per satelliti non geostazionari, la posizione relativa tra suolo (ad es. il punto sotto-satellite[7]) e satellite varia nel tempo. Pertanto, per la maggior parte del tempo non è possibile trasmettere i dati di bordo alle stazioni di terra dato che queste non sono in vista dal veicolo in orbita. Per non perdere i dati in arrivo dal *payload* e dagli altri sensori è quindi fondamentale disporre di una o più memorie nelle quali conservare i dati, in attesa di poterli scaricare verso le *ground station*.

1.5.6 Trasmettitori

I trasmettitori rappresentano il terminale nella rete di trasporto dei dati. Infatti, nel momento in cui i bit giungono al trasmettitore, l'informazione viene codificata (tramite un *encoder*), modulata (tramite un modulatore) ed infine trasmessa tramite l'antenna di bordo, sotto forma di onde elettromagnetiche, ad una o più antenne riceventi a terra. Anche il protocollo di trasmissione utilizzato può prevedere l'utilizzo di *overhead* e compressione, così come avviene all'interno dei *bus*. Come già detto, la comunicazione tra antenna trasmittente e ricevente può avvenire soltanto se le due si trovano in vista l'un l'altra; la frequenza con cui questa condizione si verifica è strettamente correlata alla tipologia di orbita ed alla posizione geografica della stazione di terra.

1.6 I programmi utilizzati

Come detto in precedenza, le due fasi fondamentali per stimare il memory budget per un satellite sono, così come rappresentato in Fig. 1.4.:

1) La propagazione dell'orbita e il calcolo della traiettoria del satellite. Tali dati vengono raccolti in un *report file*, ovvero un documento adeguatamente formattato per essere leggibile da un secondo programma.
2) Il *post-processing* dei dati contenuti nel *report file*, che in questo caso consiste nelle simulazioni di memory budget, per trarre delle conclusioni sul dimensionamento dei componenti.

[6] Sono dette LEO (Low Earth Orbit – orbita bassa terrestre) tutte le orbite dai 100 km (linea di Karman) ai 2000 km circa.
[7] Il cosiddetto punto sotto-satellite è rappresentato dall'intersezione tra la superficie terrestre e la congiungente satellite-centro della Terra.

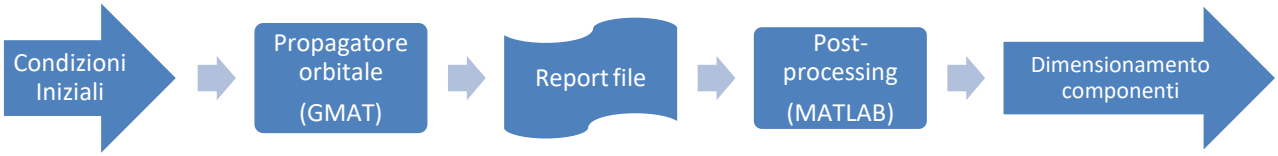

Fig. 1.4 – Diagramma di flusso del processo di Memory Budget

Per effettuare la fase (1) è stato scelto il software *GMAT*, mentre per la fase (2) si sfruttano le funzionalità presenti in *MATLAB*.

1.6.1 GMAT – General Mission Analysis Tool

Fig. 1.5 - Logo di GMAT

GMAT è stato studiato approfonditamente nelle fasi iniziali del presente lavoro, grazie alla guida utente ed alle simulazioni tutorial ad essa allegate.

Il *General Mission Analysis Tool* è un software open-source fornito e sviluppato da NASA. È una piattaforma progettata per fornire all'utente un vasto assortimento di strumenti essenziali per l'analisi di missione, con la possibilità di simulare un ampio spettro di scenari, dalle LEO (Low Earth Orbit) a missioni lunari, interplanetarie o deep space. GMAT è pensato tanto per scopi didattici quanto per un utilizzo professionale e viene regolarmente utilizzato per l'analisi di missione in progetti NASA e di altri enti.

Il software include un'ampia gamma di modelli fisici e matematici e permette all'utente un alto livello di personalizzazione per la scelta e/o l'aggiunta dei modelli stessi. È inoltre possibile estendere le funzionalità di base di GMAT con vettori, stringhe, funzioni e routine programmabili dall'utente, con la possibilità ulteriore di integrare funzioni esterne scritte in linguaggio MATLAB o Python.

1.6.2 MATLAB

MATLAB è un potente manipolatore numerico, che fornisce un ventaglio di elementi dalle diverse proprietà e funzionalità, tra cui variabili, vettori, vettori cella, strutture e funzioni, che possono essere definite nell'ambiente attraverso un linguaggio di programmazione dedicato. *MATLAB* non è solo il nome del programma di cui sopra, ma anche del linguaggio di programmazione, con il quale si può scrivere codice nell'ambiente omonimo.

La maggior parte del lavoro di tesi è stata strutturata attorno a questo tool, alla luce dei suoi numerosi punti di forza in applicazioni di questo tipo. In primis, su MATLAB è semplice per l'utente modellizzare scenari di varia natura e grandezze fisiche. Inoltre, essendo un linguaggio di alto livello con compilazione *just in time*[8], risultano rapidi ed intuitivi sia la scrittura del codice sia l'eventuale *debugging*[9]. Un altro importante vantaggio di questo software consiste nella possibilità di progettare e realizzare delle interfacce grafiche (*GUI – Graphical User Interface*) attraverso la piattaforma nativa *GUIDE*. Il prodotto finale del lavoro qui descritto è in effetti proprio una *GUI* sviluppata attraverso *GUIDE*, alla base della quale si trova il codice di calcolo, anch'esso scritto in linguaggio MATLAB.

1.7 Cenni sul tirocinio curriculare ed interfaccia con il lavoro di tesi

Per raggiungere il livello di conoscenze e competenze richieste per lo svolgimento della tesi, si è ritenuto anzitutto opportuno radicare alcuni concetti e determinati strumenti dell'analisi di missione attraverso l'esperienza di tirocinio curriculare. Il tirocinio, anch'esso svolto presso Sitael, è infatti stato strutturato in modo tale che la tesi ne fosse un naturale sviluppo e che il lavoro complessivo risultasse organico e funzionale agli obiettivi descritti nel Cap. 1.3. A tal proposito, il lavoro svolto in precedenza è stato suddiviso in quattro fasi:

- Apprendimento ed applicazione delle competenze per l'utilizzo di GMAT
- Simulazione della missione ESEO (si veda Cap. 1.7.2) in GMAT

[8] Se un software è dotato di compilazione *just in time*, o compilazione dinamica, il programma viene compilato durante l'esecuzione stessa del codice.
[9] È la fase di individuazione e correzione di errori presenti nel codice del software.

- *Post-processing* dei dati in uscita dalla propagazione su MATLAB
- Progettazione ed implementazione di una GUI MATLAB per la consultazione dei dati

È stata redatta, in precedenza, una relazione completa sul lavoro di tirocinio svolto; pertanto, in questa sede verranno riassunti soltanto gli aspetti prettamente utili per lo sviluppo della tesi.

1.7.1 Apprendimento ed applicazione delle competenze per l'utilizzo di GMAT

GMAT è un propagatore orbitale ampiamente utilizzato e dalle estese funzionalità, grazie alle quali è possibile simulare un'ampia varietà di scenari nel campo delle missioni spaziali, in orbita terrestre o interplanetaria. Il programma si è rivelato una risorsa fondamentale per apprendere i fondamenti dell'analisi di missione nel campo della meccanica orbitale.

Nella prima parte del tirocinio è stato svolto uno studio di letteratura sul manuale utente di GMAT, per comprendere le caratteristiche e le potenzialità del programma. In seguito, grazie ai casi di studio forniti dai *tutorial* presenti nel manuale utente stesso, sono stati configurati e simulati diversi scenari di interesse pratico, tra cui trasferimenti alla Hohmann, rilevazione delle eclissi e dei contatti con le stazioni di terra, utilizzo di sistemi di propulsione e studio delle perturbazioni orbitali causate dalla disomogeneità del campo gravitazionale terrestre. Come è possibile vedere in Fig. 1.6, è possibile operare in GMAT sia attraverso una interfaccia grafica che attraverso uno *script*, scritto in un linguaggio dedicato. Le informazioni contenute nei due ambienti sono totalmente equivalenti, dato che la GUI viene programmata dal contenuto dello *script* e ne mostra gli elementi in una forma più intuitiva per l'utente. Pertanto è necessario che i due elementi restino sincronizzati tra di loro, e che un eventuale aggiornamento in uno degli ambienti venga comunicato (tramite l'apposito pulsante) all'ambiente parallelo.

Come si può notare in figura, con GMAT è possibile creare dei modelli per le caratteristiche del veicolo spaziale di interesse, le stazioni di terra, le manovre (impulsive o finite) da eseguire, i solutori numerici utilizzati per il computo delle equazioni differenziali e così via. È peraltro possibile definire le uscite che interessa raccogliere al termine della propagazione, come diagrammi o documenti di testo (i cosiddetti *report file*) contenenti i vettori in uscita dalla propagazione. Grazie alla possibilità di creare un *report file* accessibile anche al di fuori di GMAT, si può in seguito interfacciare questo software con un altro programma dedicato alla fase di *post-processing* (nel nostro caso MATLAB).

Attraverso il propagatore è possibile ricavare il vettore di stato del veicolo spaziale, e quindi la sua posizione al variare del tempo. In altre parole, il programma è in grado di tracciare l'orbita dell'oggetto (e di conseguenza anche la posizione reciproca dello stesso con luoghi geografici o corpi celesti), tenendo in conto tutte le condizioni al contorno che sono state inserite all'interno della

simulazione, i metodi numerici utilizzati per integrare le equazioni differenziali e la precisione richiesta al simulatore.

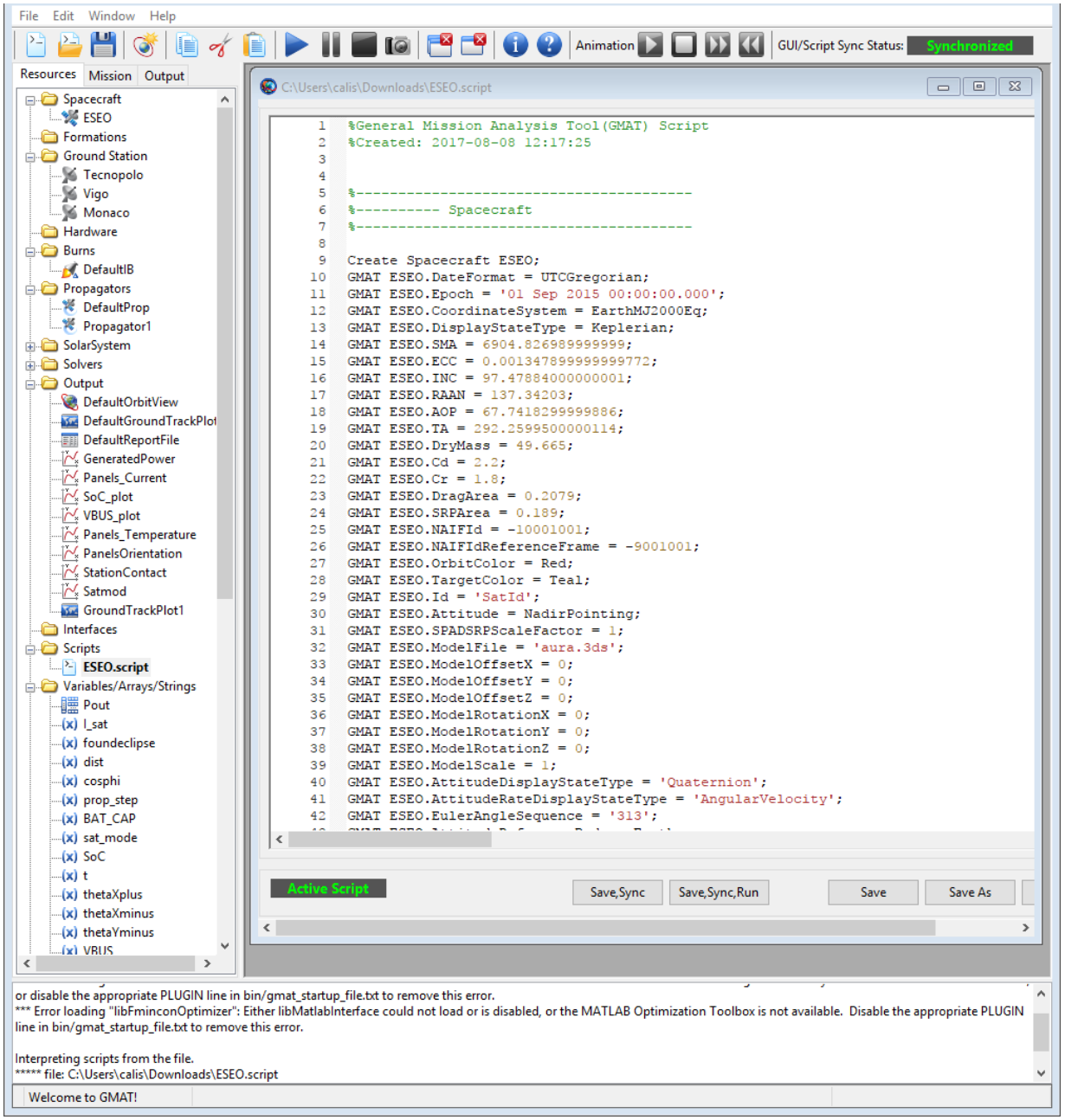

Fig. 1.6 - La finestra di GMAT che mostra il documento dello script (al centro) all'interno dell'interfaccia grafica

Per condizioni al contorno, non solo ci si riferisce alle caratteristiche orbitali del corpo (velocità iniziale, tipologia di orbita, ecc.) ma anche alla risoluzione imposta dall'utente per perturbazioni come i disturbi gravitazionali dati dalla terra e/o da altri corpi celesti, i cui effetti sono verificabili attraverso la propagazione. Ovviamente, essendo GMAT un solutore numerico, i risultati forniti sono correlati ai modelli scelti (ad es. modelli di campo gravitazionale, di atmosfera, di sistema solare, ecc.),

presenti nel programma o importabili in esso da sorgenti esterne. Infine, è possibile propagare nel corso della missione una serie di variabili e/o funzioni personalizzate, definite dall'utente prima della simulazione.

1.7.2 Simulazione della missione ESEO

Tramite gli strumenti di GMAT è stato possibile simulare lo scenario della missione ESEO, a partire dalla configurazione del satellite e delle sue specifiche, l'assetto, la caratterizzazione dell'orbita, la definizione delle stazioni di terra, ecc.

In sintesi, le peculiarità della missione e le ipotesi imposte sul modello sono:

- Orbita bassa (*SMA*[10] di 6905 km) ed eliosincrona[11] (*LTAN* 10:30). Si veda la Fig. 1.7 per ulteriori dettagli sui parametri orbitali.
- Satellite *Nadir-pointing*[12]
- Modellizzato come un cubo dal punto di vista aerodinamico
- In contatto con le stazioni di terra di Forlì, Monaco e Vigo
- Influenza gravitazionale di Terra (campo gravitazionale ad armoniche sferiche 10x10), Sole e Luna
- Resistenza atmosferica simulata tramite modelli di densità.

[10] *Semi-Major Axis*, o semiasse maggiore. È uno dei parametri orbitali fondamentali ed indica la dimensione dell'orbita, oltre che a caratterizzare molte grandezze d'interesse, tra cui l'energia orbitale.

[11] Questa tipologia di orbita trova impiego per il cosiddetto 'remote sensing' e cioè per effettuare rilevazioni di dati puntando la strumentazione (ad esempio una fotocamera) verso la Terra. Per questo tipo di missioni sono spesso necessarie condizioni di illuminazione costanti, dal punto di vista del satellite ed è per questo che risulta così importante l'inclinazione della luce solare sulla Terra. Esiste infatti un parametro, chiamato LTAN (Local Time of the Ascending Node, ovvero 'ora locale del Nodo Acendente') che posiziona il Nodo Ascendente dell'orbita in modo da imporre l'ora locale nel punto in cui il satellite passa sull'equatore. Ad esempio, un'orbita eliosincrona con LTAN 12:00 indicherà un satellite che passa sull'equatore a mezzogiorno e a mezzanotte (in ora locale). ESEO è una missione con LTAN 10:30, che significa che passa sull'equatore alle 10:30 in ascesa ed alle 22:30 in discesa.

[12] Un satellite *Nadir-pointing* mantiene, durante l'orbita, un assetto tale per cui un asse del satellite punta sempre verso il suo Nadir locale. Questo vincolo garantisce il puntamento costante di uno o più strumenti (ad esempio, una fotocamera) verso la Terra, in particolare in direzione della verticale locale.

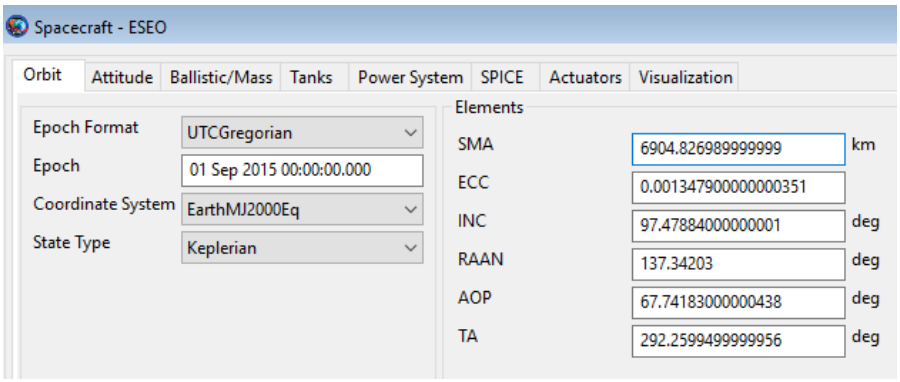

Fig. 1.7 - A sinistra: il calendario, la data di inizio ed il sistema di riferimento fisso selezionati. A destra: i parametri orbitali, in questo caso kepleriani.

Come anticipato, su GMAT è possibile creare delle varabili e delle funzioni non presenti di default nel programma, per calcolare, attraverso la simulazione, altre grandezze di interesse. Nel caso della simulazione di ESEO sono state definite una serie di variabili e funzioni ad hoc per valutare l'andamento dei parametri principali di ESEO nel corso della missione. Tra le variabili compaiono tensioni, correnti, capacità delle batterie, assetto, visibilità sulle stazioni di terra e così via. I valori di queste vengono calcolati, passo dopo passo, dagli algoritmi delle funzioni appositamente scritte in fase di inizializzazione. Tra queste funzioni compare *gs_contact*, che è stata fondamentale per la stima del memory budget nel software sviluppato in MATLAB. Infatti, quest'ultima funzione definisce la visibilità sulla/e stazione/i di terra, distinguendo quindi gli istanti in cui il trasmettitore può scaricare dati (e quindi svuotare le memorie) da quelli in cui ciò non è possibile (nei quali le memorie continuano a riempirsi). Data l'importanza di questa funzione nell'analisi di un memory budget, è stata riportata, nel Cap. 1.7.4, una serie di considerazioni geometriche che ne giustificano l'algoritmo. L'ultimo passo prima di avviare la simulazione è quello di definire la sequenza di missione, inserendo nell'apposita sezione di GMAT tutte le azioni che il simulatore deve svolgere nel corso del calcolo. La sequenza di missione può quindi contenere richiami delle funzioni, manovre da effettuare (grazie ai relativi *burn*[13]), passaggi da una modalità operativa ad un'altra e così via. A titolo di esempio, la Fig. 1.8 mostra una porzione di sequenza di missione di ESEO, nella quale vengono richiamate le funzioni sopracitate e viene propagata l'orbita.

[13] Accensione di un propulsore del satellite, spesso per compiere una manovra orbitale o di controllo d'assetto.

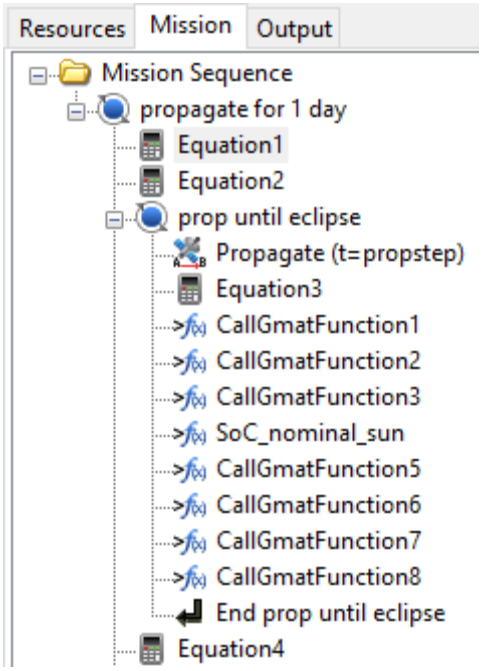

Fig. 1.8 - La prima parte della sequenza di missione di ESEO

Al termine della propagazione, settata dall'utente in base al periodo di tempo da simulare (per ESEO la missione nominale è di 6 mesi), si dispone di tutte le variabili in uscita richieste, raccolte nel *report file*. L'utente ha la possibilità di selezionare le variabili da conservare nel documento e in che modo quest'ultimo debba essere formattato. Per stimare il memory budget, come si vedrà nel seguito, ci si è serviti delle variabili

- *UTCGregorian*, che rappresenta il calendario gregoriano. È una variabile predefinita in GMAT, usata per localizzare nel tempo ogni intervallo della simulazione.
- *foundeclipse*, un *flag*[14] che segnala le fasi dell'orbita che il satellite spende in eclissi, ricavato attraverso la funzione custom *Eclipse_Finder*.
- *gscontact*, che come già detto è un *flag* utilizzato per segnalare la visibilità, dal punto di vista del satellite, delle stazioni di terra.

Dalla figura Fig. 1.9, che rappresenta la traccia a terra del satellite sulla *proiezione di Mercatore*[15], si può notare graficamente che l'orbita rispecchia le caratteristiche selezionate in precedenza e si ha dunque una ulteriore conferma della validità dei risultati, che verranno utilizzati dal Cap. 2 in poi.

[14] Un indice booleano che identifica nel codice se una particolare circostanza si è verificata (1) o no (0) in un dato istante.
[15] La proiezione di Mercatore è uno dei modi più utilizzati per rappresentare la Terra su un piano bidimensionale. È una derivazione della proiezione cilindrica, a cui applica alcune correzioni.

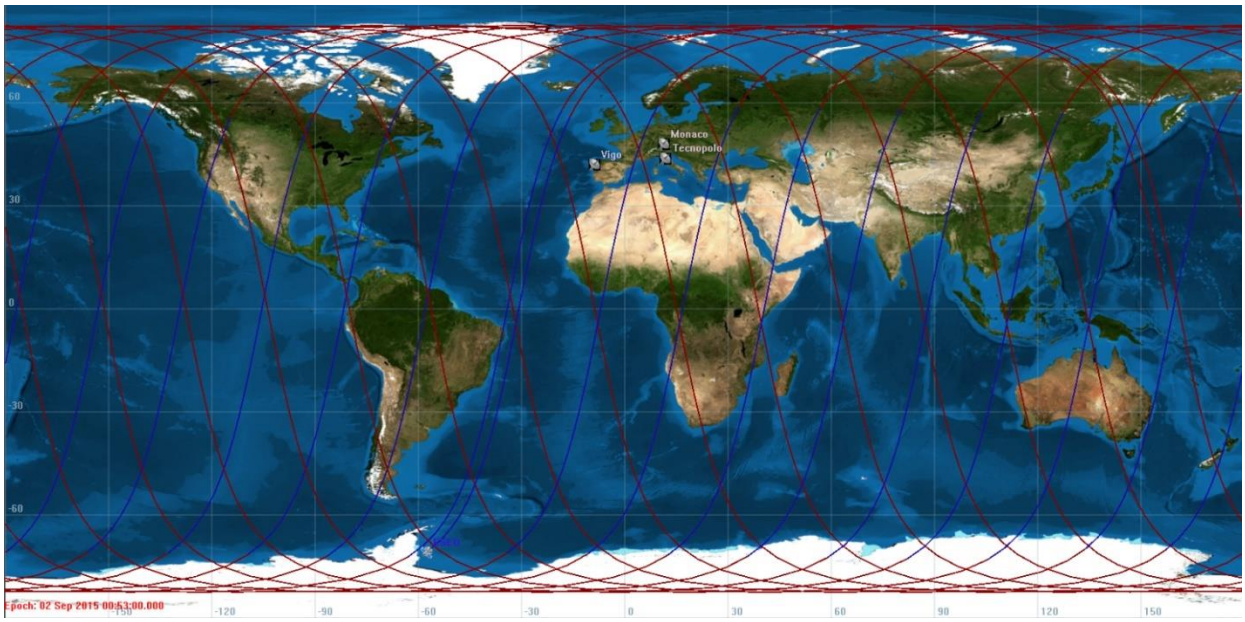

Fig. 1.9 - L'orbita di ESEO proiettata sul Mercatore. In rosso: le fasi in daylight; in blu: le fasi in eclissi

1.7.3 Il memory budget e la fase di eclissi

Come si vedrà nel seguito, nel computo del memory budget è necessario anche determinare le fasi in cui il satellite si trova in eclissi. Se la stazione di terra è in vista del satellite mentre quest'ultimo è eclissato, necessariamente il passaggio avviene mentre è notte nella zona della stazione di terra. A seconda di come sono strutturate la stazione di terra (*ground segment*) e l'intero scheduling delle operazioni, la trasmissione di dati in notturna potrebbe essere non realizzabile. Le simulazioni di memory budget devo quindi poter tener conto di questa particolare eventualità, permettendo all'utente di discriminare tra passaggi diurni e notturni.

Per quantificare il periodo in cui il satellite si trova eclissato, è possibile servirsi di alcune considerazioni geometriche.

Si supponga di essere nelle condizioni mostrate in Fig. 1.10: il satellite orbita attorno alla Terra (di raggio medio R_t) ad una quota h. La radiazione solare proviene da destra, pertanto nella parte ad est sul pianeta è giorno, ad ovest è notte. Dato che la terra si frappone tra il Sole e lo spazio dietro di essa, questa proietterà la sua ombra, detta appunto cono d'ombra, nella direzione opposta rispetto a quella della stella. Trascurando le zone di cosiddetta *penumbra*, il cono d'ombra può essere modellizzato come un cilindro.

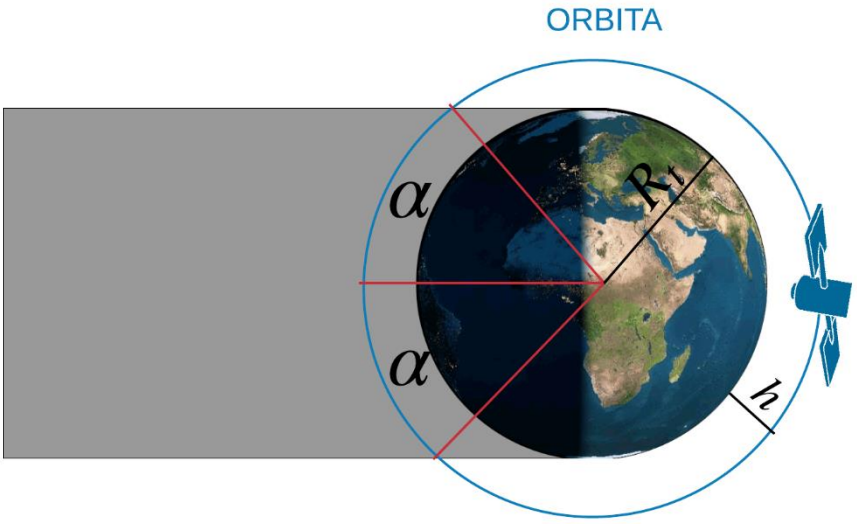

Fig. 1.10 - Considerazioni geometriche sulla fase di eclissi

Come mostrato dalla figura, il satellite risente del cono d'ombra terrestre ed una fase della sua orbita giace dentro di esso. Per determinare la lunghezza dell'arco di orbita passato in eclissi, si può sfruttare la relazione:

$$R_t = (R_t + h) * \sin(\alpha)$$

da cui si ricava

$$\alpha = \sin^{-1}\left(\frac{R_t}{R_t + h}\right)$$

dove α rappresenta il semiarco dell'orbita in eclissi.

È facile vedere che α dipende dalla quota del satellite: più in alto è collocata l'orbita, minore è il periodo di eclissi. Per satelliti in orbita bassa, questa condizione occupa una porzione notevole del periodo orbitale.

Si supponga, ad esempio, un'orbita a quota 500 km. Il periodo orbitale (T) può essere calcolato come

$$T = 2\pi\sqrt{\frac{a^3}{\mu}} \cong 5668\ s = 1.57\ ore$$

dove a è il Semiasse Maggiore dell'orbita:

$a = R_t + h \cong 6371\ km + 500\ km = 6871\ km$

e μ la costante planetaria della Terra, che vale $398600.4\ \frac{km^3}{s^2}$.

Il periodo che il satellite passa in eclissi è uguale a

$$T_e = \frac{2\ \alpha}{360} * T \cong 2141\ s \cong 0.59\ ore$$

essendo α uguale a circa 68 gradi.

Con questo esempio è possibile notare che, per i satelliti in orbita bassa, la fase di eclissi non è trascurabile. Nel memory budget, quindi, si crea una sensibile differenza tra il caso in cui un satellite può trasmettere di notte ed il caso in cui non può farlo. Nel secondo caso sarà infatti necessario sovradimensionare la capacità delle memorie. Nella propagazione in GMAT, la fase di eclissi viene determinata dalla funzione custom *Eclipse_Finder* che, attraverso considerazioni geometriche sulla posizione relativa tra i corpi (1), assegna alla variabile *foundeclipse* il valore 1 se si è in fase di eclissi, il valore 0 altrimenti.

1.7.4 Il memory budget e le fasi di downlink

L'aspetto che più incide sui risultati del memory budget è la frequenza e la durata degli intervalli di orbita in cui è possibile effettuare il *downlink* (la fase di trasmissione dati dal satellite a terra) con le stazioni di terra. Infatti, solo quando le ground station sono in vista dal satellite è possibile scaricare i dati e quindi liberare spazio nelle memorie. In generale, un satellite è visibile per una stazione di terra soltanto se il suo angolo di elevazione (l'angolo misurato tra l'orizzonte locale e il punto nel cielo in cui si trova il satellite) è maggiore di zero. Per questo motivo, un satellite non è sempre visibile da terra (salvo rari casi specifici) perché la posizione relativa tra la stazione e il satellite cambia nel tempo.

Così come per le eclissi, anche per il downlink i satelliti in orbita bassa si trovano penalizzati. Infatti, data la velocità orbitale elevata (dovuta alla bassa quota), essi attraversano velocemente il cielo sopra alla stazione di terra e la visibilità dura circa 10-15 minuti. La durata del passaggio sopra alla stazione di terra è correlata all'angolo di elevazione: quest'ultimo parte da zero, aumenta, raggiunge un massimo e ritorna a zero nel momento in cui il satellite scende sotto l'orizzonte. Quando l'angolo di elevazione vale zero, il corpo in orbita appare quindi all'osservatore coincidente con l'orizzonte.

All’atto pratico, però, l’attenuazione del segnale dovuta all’atmosfera frapposta tra i due oggetti e/o alla presenza di eventuali ostacoli (rilievi montuosi, umidità, ecc.) impedirebbe la comunicazione tra satellite e stazioni di terra. Per questo motivo viene un *angolo di maschera*, che indica il limite angolare sotto il quale, nelle simulazioni, si considera impossibile la comunicazione terra-satellite, suppere quest'ultimo sia geometricamente in vista. Nelle simulazioni svolte in GMAT, ad esempio,l'angolo di maschera è stato posto uguale a 10 gradi. Una soluzione per aumentare la frequenza del downlink è quella di aumentare il numero di stazioni di terra e distribuirle sulla superficie terrestre La funzione custom di GMAT chiamata *gs_contact* e utilizzata per le simulazioni è in grado di determinare, sempre grazie ad una flag, quando la visibilità reciproca tra spacecraft e stazione di terra si verifica.

2. CAPITOLO 2
LO SVILUPPO DEL SOFTWARE

In questo capitolo verrà in primo luogo illustrata la struttura concettuale del software, ne verranno poi valutati i requisiti, gli ingressi e le uscite, e saranno chiariti i modelli utilizzati e le ipotesi adottate. In un secondo momento si passerà invece alla trattazione dettagliata del codice stesso: la descrizione del programma è stata suddivisa in paragrafi, ciascuno dei quali tratta nel dettaglio i criteri, gli accorgimenti e gli strumenti con i quali le righe di codice sono state scritte, nonché i riferimenti alle principali funzionalità MATLAB utilizzate.

2.1 Architettura generale del software

2.1.1 Requisiti generali

Come già introdotto nel Cap. 1.3 e nel Cap. 1.6, l'oggetto della tesi è la realizzazione di un programma scritto in linguaggio MATLAB. Tramite la relativa GUI l'utente può modellizzare il processo di scambio dati tra sensori, *bus*, memorie e trasmettitori del satellite, insieme con i loro parametri caratteristici. L'utente ha la possibilità di importare un *report file* creato con GMAT, impostare l'intervallo temporale di simulazione, configurare un modello personalizzato dei componenti dedicati allo scambio dei dati ed infine ricavare le grandezze d'interesse. Attraverso queste ultime, è possibile verificare che i dimensionamenti eseguiti per le quattro categorie di componenti che operano con lo scambio dati siano idonei ai requisiti di missione, tra cui la mole di dati da elaborare e trasmettere, gli intervalli di tempo in cui è possibile effettuare il *downlink*, i criteri di attivazione dei payload e così via.

2.1.2 Gli ingressi del programma

Il software richiede quindi in ingresso:

- Un *Report File* contenente i dati orbitali che descrivono gli intervalli di visibilità tra spacecraft e stazione di terra
- L'architettura del sistema di data handling: sensori → bus → memorie → trasmettitori, ovvero il criterio con cui questi quattro gruppi di elementi sono stati collegati tra loro
- Le specifiche tecniche e le caratteristiche di ogni elemento caricato nel modello
- L'intervallo temporale da simulare

2.1.3 Le uscite del programma

Al termine della simulazione, il software restituisce:

- L'andamento nel tempo dello stato di occupazione dei buffer di ogni sensore
- L'andamento nel tempo dello stato di occupazione nelle memorie di bordo
- Il *Data Volume* totale acquisito (cioè generato dal satellite)
- Il *Data Volume* totale scaricato verso le stazioni di terra durante le fasi di *downlink*
- Il *Data Rate* medio di acquisizione dati di ogni sensore

2.1.4 La funzione *Runtime*[16]

Il computo del memory budget vero e proprio (e quindi il calcolo delle uscite) è a carico della cosiddetta funzione *runtime* (descritta nel dettaglio nel paragrafo 2.9), che è stata definita come un programma a sé stante, e rappresenta il vero centro di calcolo ed elaborazione dei dati. In questo caso, la funzione richiede in ingresso tutti i parametri e le grandezze forniti tramite GUI, ma sarebbe possibile utilizzarla anche senza l'ausilio dell'interfaccia grafica, definendo manualmente, all'interno del codice di *runtime*, le grandezze richieste oppure collegando la funzione ad uno script contenente i dati necessari.

2.1.5 L'architettura del sistema da modellizzare

A seconda dei requisiti di missione, un satellite viene progettato con un certo numero di sensori, memorie e trasmettitori. Questi componenti possono essere collegati tra loro attraverso i bus, formando diverse architetture. Nel progettare una configurazione attraverso il programma, l'utente deve quindi essere libero di collegare questi oggetti tra loro con un criterio arbitrario. In altre parole, tutti i sensori possono essere collegati a tutti i bus, ogni bus può essere allacciato ad una qualunque memoria e così via. Per permettere all'utente una tale libertà, gli elementi del programma sono stati suddivisi in quattro sezioni. Sarà poi l'utente a decidere in che modo collegare tra loro gli oggetti contenuti nelle sezioni, attenendosi allo schema sensori→bus→memorie→trasmettitori. A titolo di

[16] Il termine *runtime* indica il momento in cui il programma viene eseguito. In questo caso ci si riferisce all'esecuzione del codice dedicato alla stima del *memory budget*, in contrapposizione al codice che invece si occupa, attraverso la GUI, di caricare in MATLAB il modello fornito dall'utente.

esempio, vengono riportati in Fig. 2.1 due diversi criteri di collegamento per una stessa configurazione di componenti.

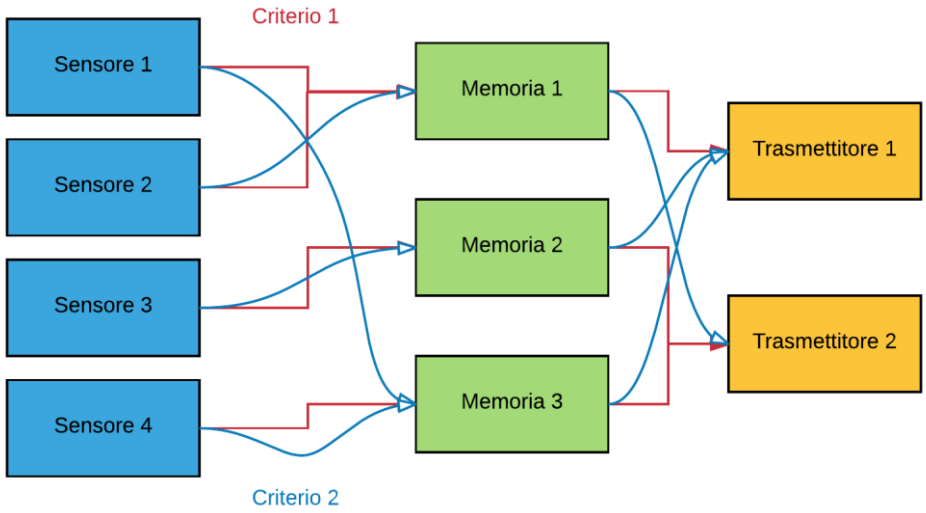

Fig. 2.1 – Due esempi di reti logiche realizzabili per uno stesso satellite

2.1.6 Mappa e spiegazione della logica del programma

Alla luce di tutte le osservazioni di cui sopra, l'architettura del codice è stata realizzata secondo lo schema in Fig. 2.2.

La mappa va letta da sinistra a destra e il codice di colori, come illustrato dalla legenda, caratterizza ogni blocco secondo la propria funzionalità:

- *Grigio* – sono documenti di vario genere e con diverse estensioni, che verranno specificate in seguito
- *Giallo* – sono i valori e le voci inseriti manualmente dall'utente, attraverso la GUI
- *Verde chiaro* – sono i risultati finali del memory budget, cioè le uscite della funzione *runtime*
- *Blu* – sono i valori e le grandezze che vengono allocate e memorizzate, grazie al software, nel *Workspace*[17] di MATLAB.
- *Rosso* – è la funzione *runtime* (si veda Cap. 2.1.4)

[17] Si veda Cap. 2.2.2.1

- *Celeste* – Sono gli elementi visualizzabili a video (in questo caso si tratta di diagrammi)
- *Verde acqua* – È la memoria su cui vengono salvati, alla chiusura del programma, i dati ricavati

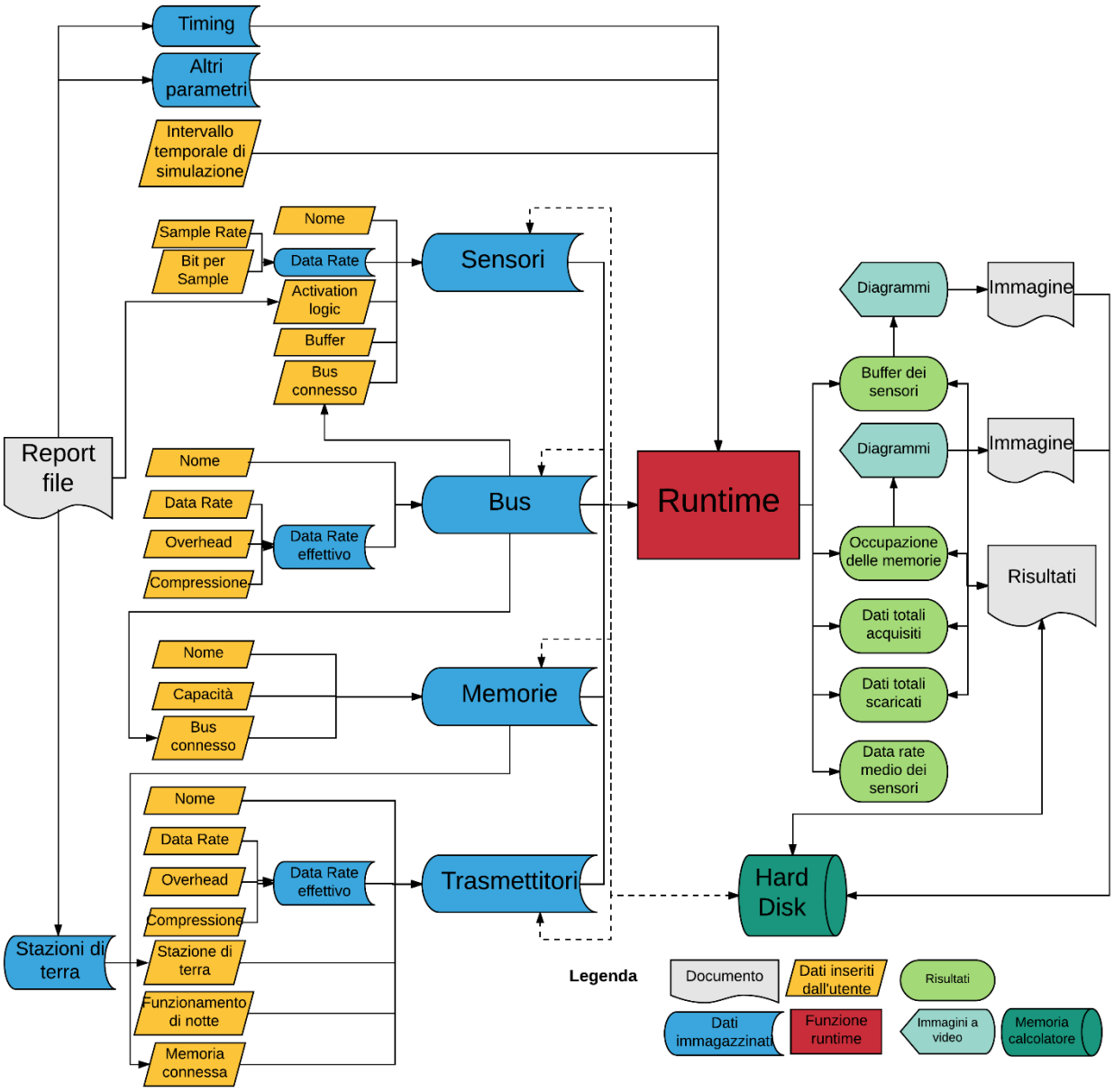

Fig. 2.2 - Il diagramma di flusso del funzionamento del programma

Nel seguito verrà illustrata la mappa più nel dettaglio. Si ricorda, tuttavia, che in questa prima parte del capitolo la trattazione è prettamente concettuale e che le metodologie utilizzate per implementare nel software lo schema logico di cui sopra verranno trattate nei prossimi paragrafi.

2.1.6.1 Caricamento dei dati nel programma

Come si può osservare, il punto di partenza nella mappa sono le informazioni contenute nel *Report File*, il documento nel quale sono stati salvati, tramite la simulazione di GMAT, i parametri di interesse in uscita dal propagatore, tra cui il vettore tempo indicato con *Timing*. Esso è fondamentale per referenziare ogni passo della simulazione eseguita da *runtime* con la rispettiva posizione sull'asse tempo (definita da data e ora), ed è quindi un ingresso che viene valutato dal blocco in rosso. Pertanto, il primo passaggio da eseguire dopo l'apertura del programma è il caricamento di tale *Report File* all'interno del programma.

Il secondo gruppo di operazioni da svolgere consiste nella compilazione delle caratteristiche inerenti ai sensori, ai *bus*, alle memorie ed ai trasmettitori.

In particolare, per i sensori vengono richiesti: un nome identificativo, il *Data Rate* (calcolabile inserendo singolarmente *Sample Rate* e *Bit per Sample*), un operatore logico che definisca le istanze in cui il sensore si trova acceso e funzionante, la capacità della memoria interna del sensore (il cosiddetto *buffer*) e il *bus* in uscita dal sensore, cioè il connettore attraverso il quale i dati possono essere scaricati a valle nello schema (cioè verso le memorie).

Per definire le caratteristiche del bus, l'utente deve poi inserire un nome identificativo, il *Data Rate* massimo che il bus è in grado di fornire ed eventuali overhead e compressione attuati dal bus (i quali possono variare il *Data Rate*, pertanto vanno presi in considerazione).

Nel terzo blocco partendo dall'alto vengono caratterizzate le memorie, per le quali si richiede, di nuovo, il nome, la capacità, ed il *bus* in arrivo alla memoria (ovvero il bus a monte della memoria, si veda la Fig. 1.3).

Infine vengono inseriti i dati anche per i trasmettitori di bordo: nome, *Data Rate*, eventuali overhead e compressione, le stazioni di terra con le quali il trasmettitore effettua il downlink, l'eventuale possibilità di trasmettere anche durante le ore notturne e la memoria a cui il trasmettitore è collegato, dunque la memoria che viene svuotata grazie al trasmettitore ad essa correlato.

2.1.6.2 Ipotesi semplificativa e applicazioni

Per quanto riguarda memorie e trasmettitori, va riportata una importante osservazione: come visto in Fig. 1.3, questi due elementi sono nella realtà spesso collegati tra loro tramite uno dei bus di bordo, dunque sarebbe logico caratterizzare ogni trasmettitore a seconda del bus ad esso collegato e non a seconda della memoria ad esso collegato, come invece detto poc'anzi. Tuttavia, per semplificare

l'approccio alla simulazione, supponendo che la velocità di trasmissione dei dati tra la memoria ed il trasmettitore annesso sia pressoché infinita. In altre parole, tutti i bit che arrivano alla memoria, sono immediatamente visibili anche per il trasmettitore, come se i due rappresentassero di fatto un unico elemento. È ovvio che la modellizzazione realizzata così come appena descritta introduce un'ipotesi semplificativa di cui è necessario tenere conto durante le analisi. Tuttavia, come si vedrà nel Cap. 4, è possibile trovare casi reali di satelliti nei quali la memoria in cui vengono immagazzinati i dati è un componente del trasmettitore stesso e pertanto non soffre di alcun rallentamento dovuto ad un limitato *Data Rate* che caratterizza un qualunque bus reale.

2.1.6.3 Elaborazione dei dati e risultati

Una volta completato il caricamento dei dati necessari alla simulazione, il programma conosce le caratteristiche e le specifiche del satellite necessarie a produrre le uscite richieste. Si passa dunque all'esecuzione del codice *runtime*: il blocco vede in ingresso le caratteristiche dei componenti (sensori, bus, memorie e trasmettitori) e i tutti i valori contenuti nel report file. Dopo il calcolo, il programma fornisce l'andamento nel tempo di occupazione dei buffer e delle memorie, la quantità totale di dati acquisiti dai sensori e scaricati dai trasmettitori e il *Data Rate* medio dei sensori. Inoltre, il software permette di visualizzare, sotto forma di diagrammi, l'andamento relativo all'occupazione dei buffer e alle memorie oltre ai valori delle altre voci sopracitate.

2.1.6.4 Caricamento e salvataggio dei risultati

Per conservare in memoria i risultati ottenuti dopo la chiusura del software, l'utente può seguire due strade: salvare l'intera gamma di uscite come variabili MATLAB, in un formato tale da poter essere, eventualmente, ricaricate nel programma, oppure procedere con il salvataggio dei soli *plot* in formato immagine. È inoltre possibile salvare la configurazione dei componenti (cioè quella descritta nel Cap. 2.1.6.1), per poterla ricaricare e modificare in futuro (le frecce che partono dai quattro blocchi blu principali possono condurre a *runtime,* oppure direttamente all' *Hard disk).* Infatti, il programma è in grado di caricare dei risultati o uno scenario creati in precedenza.

2.2 L'interfaccia grafica

Come accennato nel Cap. 1.7, nell'ultima parte del lavoro di tirocinio erano stati studiati ed utilizzati gli strumenti forniti da MATLAB per l'implementazione di una interfaccia grafica. Uno dei modi in cui si può realizzare una GUI con MATLAB è quello di utilizzare la piattaforma *GUIDE* (*Graphical User Interface Design Environment*), presente all'interno del programma. *GUIDE* fornisce un ambiente grafico rapido ed intuitivo per posizionare e caratterizzare gli elementi della GUI da realizzare. Con essa è possibile posizionare pulsanti, liste, caselle di testo, finestre per diagrammi e tutti gli altri elementi messi a disposizione. È inoltre possibile la personalizzazione delle caratteristiche di ogni elemento grafico, in modo tale da rendere l'elemento stesso funzionale per l'operazione da eseguire.

La versione finale dell'interfaccia, completamente realizzata con GUIDE, è mostrata in Fig. 2.3. L'immagine mostra la schermata attraverso cui GUIDE permette di configurare l'interfaccia grafica, pertanto sono visibili tutti gli elementi inseriti (compresi i messaggi e le finestre di errore, che sono invisibili di default nella GUI operativa) e la griglia sullo sfondo, utile per posizionare ordinatamente gli elementi.

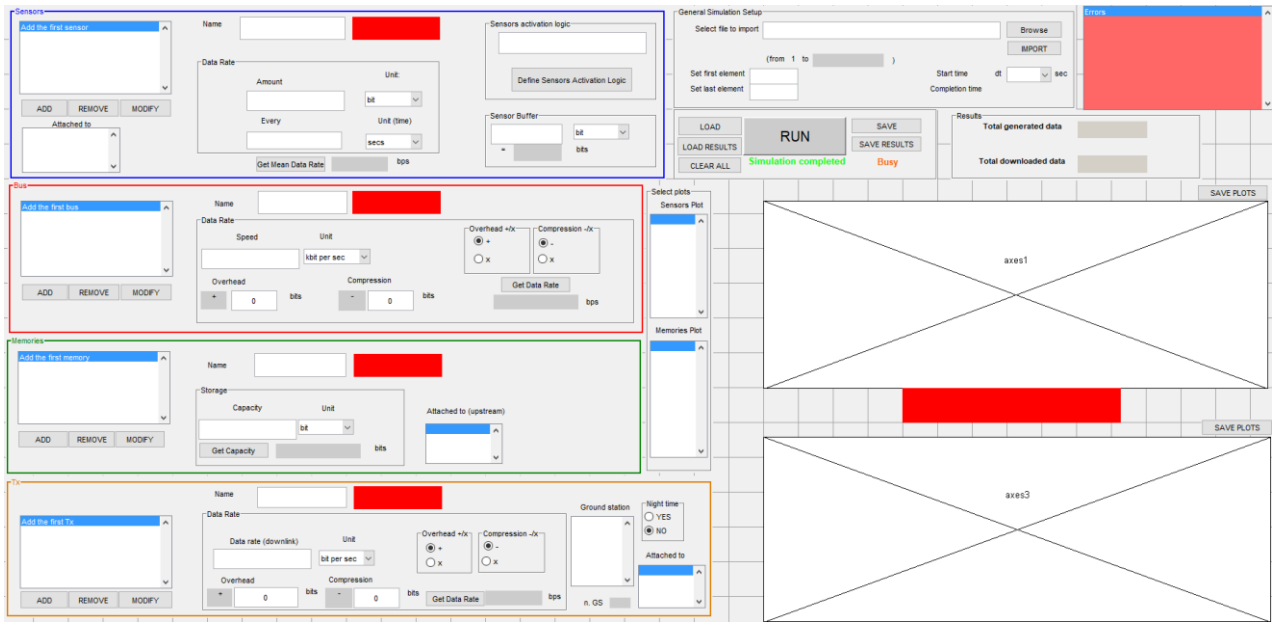

Fig. 2.3 – L'interfaccia grafica finale, caricata all'interno di GUIDE. Sulla sinistra in alto: gli elementi che è possibile aggiungere alla GUI

Come è possibile vedere, l'interfaccia è stata suddivisa in due aree principali: a sinistra la parte dedicata alla configurazione dei componenti del satellite (in particolare, dall'alto verso il basso, di

sensori, bus, memorie e trasmettitori), mentre quella di destra è pensata per il caricamento del report file, la selezione dell'intervallo temporale di simulazione, il caricamento ed il salvataggio dei dati e la visualizzazione grafica dei risultati. Ciascuna delle due aree si suddivide a sua volta in vari blocchi che d'ora in avanti verranno chiamati *sezioni*. Ogni sezione permette all'utente la configurazione di parte dei modelli necessari al memory budget.

2.2.1 Gli elementi di una interfaccia grafica

Per facilitare la comprensione dei paragrafi a seguire, viene qui riportata una presentazione dei principali elementi presenti in una interfaccia grafica di MATLAB. GUIDE fornisce una serie di oggetti, ognuno dei quali vanta caratteristiche grafiche e funzionali proprie. Gli elementi utilizzati per lo sviluppo di questo programma sono (utilizzando il codice di colori dei riquadri in Fig. 2.4):

- *Push Button* – sono dei pulsanti che l'utente può premere con il puntatore del mouse, attivando così determinate funzioni.
- *Radio Button* – bottoni che permettono solitamente di selezionare una sola voce tra una serie di opzioni proposte. Restituiscono al codice una variabile booleana (0 o 1), a seconda che siano stati o meno selezionati.
- *Edit Text* – casella in cui l'utente può inserire una stringa di testo
- *Static Text* – casella utilizzata per mostrare all'utente una stringa di testo, senza però che l'utente possa modificarla
- *Pop-up Menu* – una lista di elementi che è consultabile per intero cliccando sulla apposita icona a forma di freccia; l'utente può selezionare la voce desiderata dalla lista. Ha il vantaggio di poter contenere molte informazioni in uno spazio ridotto. Restituisce il valore della voce selezionata dall'utente (ad es. 7, se l'utente ha selezionato la settima voce della lista)
- *Listbox* – simile a *Pop-up Menu*, ma può essere consultata per intero scorrendo la lista stessa con le apposite icone. È meno compatta e mostra più voci nella stessa schermata.
- *Axes* – una schermata che mostra i plot eseguiti dal codice. Come si vedrà, è possibile visualizzare più plot contemporaneamente.
- *Panel* – raccoglie un gruppo di elementi in uno stesso pannello. È utile ad esempio per suddividere ordinatamente l'interfaccia.
- *Radio Group* – simile a *Panel* ma utilizzato per raccogliere una serie di *Radio Button*.

Si noti che lo stesso codice di colori verrà utilizzato anche nei diagrammi concettuali presenti nei successivi paragrafi di questo capitolo, così da poter identificare il tipo di elemento in oggetto.

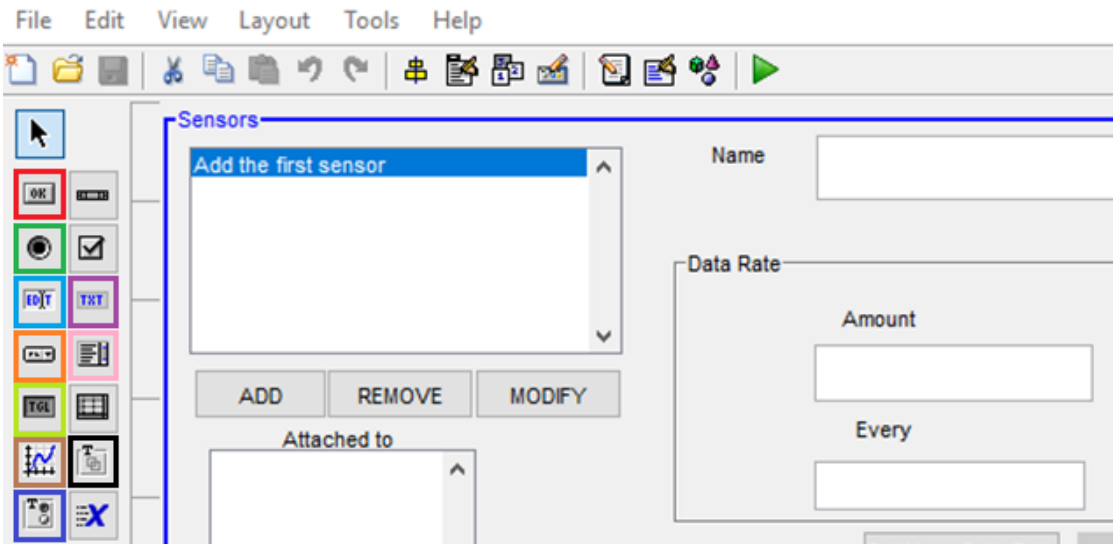

Fig. 2.4- Sulla sinistra: le icone degli elementi che è possibile inserire in una GUI MATLAB

2.2.2 Il codice di una interfaccia grafica

La programmazione di una GUI in MATLAB presuppone la conoscenza del criterio con cui il programma crea l'interfaccia grafica e gli strumenti attraverso i quali è possibile passare informazioni da un elemento grafico ad un altro.

Il primo concetto da tenere in considerazione è il fatto che una interfaccia grafica lavora sulla base di un codice, dato che essa stessa è generata da un codice ad hoc e che compiendo operazioni su di essa, in realtà si effettuano manipolazioni nel codice stesso. Si può dunque dire che l'interfaccia grafica è uno strumento visivo, facilmente accessibile e funzionale per manipolare un programma e quindi svolgere delle operazioni. In altre parole, tutte le operazioni che è possibile compiere con una interfaccia grafica, possono essere svolte anche direttamente tramite un codice. Tuttavia, il grande vantaggio di una GUI consiste nel garantire una immediata accessibilità, per l'utente, alle funzionalità del codice, senza dover necessariamente passare attraverso il linguaggio di programmazione, risparmiando così in tempo e complicazioni tecniche. In particolare, per software dedicati alla configurazione di modelli piuttosto flessibili, come nel caso in oggetto, essa porta ad un grande risparmio di tempo da parte dell'utente, che tra l'altro non deve necessariamente conoscere nel dettaglio il codice collegato alla GUI.

Il criterio con cui gli elementi delle interfacce comunicano tra loro (che siano essi interni od esterni alla stessa sezione), verrà illustrato nei paragrafi dedicati alla configurazione del satellite, anche attraverso diagrammi concettuali. In seguito, verranno descritte invece nel dettaglio le funzioni e le operazioni svolte in MATLAB per garantire tale comunicazione.

2.2.2.1 I workspace, i callback e gli handles

Nello sviluppo di una interfaccia grafica è importante sapere che cosa sono e che gerarchia assumono i *workspace* all'interno di MATLAB. Un *workspace* è un'area di memoria in cui sono conservate tutte le variabili (matrici, strutture, vettori cella, stringhe, ecc.) generate dai comandi MATLAB. Ciò significa che gli elementi che operano in un *workspace* sono disponibili, di default, soltanto all'interno di esso. Principalmente, ci sono due categorie di workspace, il *workspace base* ed il workspace di una funzione. Nel primo è possibile caricare le variabili attraverso la *command window* di MATLAB, oppure eseguendo uno *script* dalla riga di comando. Le variabili caricate nel *base* rimangono salvate e disponibili fino alla chiusura di MATLAB o fintanto che l'utente non decide di cancellarle. Nei *function workspace* invece vengono salvate solo le variabili generate internamente ad una function, e sono accessibili solo durante il runtime di quella funzione. Il workspace viene poi cancellato una volta che la funzione ha completato la run e gli output vengono trasferiti al workspace base.

```
2332      % --- Executes on button press in pushbutton30.
2333      function pushbutton30_Callback(hObject, eventdata, handles)
2334      % hObject    handle to pushbutton30 (see GCBO)
2335      % eventdata  reserved - to be defined in a future version of MATLAB
2336      % handles    structure with handles and user data (see GUIDATA)
2337 -    t = evalin('base','t');
2338 -    index_selected = get(handles.ListTx,'Value');
2339 -    t(index_selected).name = char(get(handles.EditTxName,'String'));
2340 -    t(index_selected).data = str2num(get(handles.StaticDataRate3,'String'));
2341 -    memories = get(handles.ListMemToTx,'String');
2342 -    index_memory_selected = get(handles.ListMemToTx,'Value');
2343 -    memory_selected = memories(index_memory_selected);
2344 -    t(index_selected).attached_to = char(memory_selected);
2345 -    list= get(handles.ListTx,'String');
2346 -    list(index_selected) = {t(index_selected).name};
2347 -    set(handles.ListTx,'String',list);
2348 -    index_selected2 = get(handles.ListGS,'Value');
2349 -    GSs = get(handles.ListGS,'String');
2350 -    GS_selected = GSs(index_selected2);
2351 -    t(index_selected).GS = char(GS_selected);
2352 -    t(index_selected).night = get(handles.RadioNightYES,'Value');
2353 -    t(index_selected).index_GS_selected = get(handles.ListGS,'Value');
2354 -    t(index_selected).num_GS_selected = length(t(index_selected).index_GS_selected);
2355 -    t(index_selected).EditDataRateDownlink = str2num(get(handles.EditDataRateDownlink,'String'));
2356 -    t(index_selected).PopUpBit4 = get(handles.PopUpBit4,'Value');
2357 -    t(index_selected).StaticPlusOrTimes3 = get(handles.StaticPlusOrTimes3,'String');
2358 -    t(index_selected).StaticPlusOrTimes4 = get(handles.StaticPlusOrTimes4,'String');
2359 -    t(index_selected).EditOverhead2 = str2num(get(handles.EditOverhead2,'String'));
2360 -    t(index_selected).EditCompression2 = str2num(get(handles.EditCompression2,'String'));
2361 -    set(handles.StaticnGS,'String',t(index_selected).num_GS_selected);
2362 -    assignin('base','t',t);
2363
```

Fig. 2.5 - Un esempio di callback per un pushbutton. Si noti che tra gli ingressi compare la voce handles

In una GUI, ogni elemento è rappresentato nel codice da una propria funzione, chiamata *callback* (Fig. 2.5). Come suggerisce il nome, nel momento in cui l'utente interagisce con l'elemento (ad esempio, nel caso di un bottone, premendolo con il puntatore del mouse), tale funzione viene richiamata nel codice, attivando l'algoritmo in essa contenuta.
Questo concetto è facilmente immaginabile se, come detto in precedenza, si pensa che la GUI opera in sincronia con il proprio codice e di fatto richiama (ogni volta che l'utente interagisce con essa) le funzioni a cui gli elementi della prima sono collegati. I *callback*, come la maggior parte delle funzioni, richiedono degli ingressi, ma non hanno invece delle uscite specificate.
Per quanto detto riguardo alle differenze tra i due tipi di spazio di lavoro, ogni elemento della GUI possiede un *function workspace* a sé stante e separato dagli altri. Questo significa che le variabili definite nel *callback* di un elemento non sono conosciute dal resto del codice al di fuori di esso, e che una volta terminate le operazioni in tale funzione, se non si agisce per evitarlo, queste variabili vengono perse. Tuttavia, è spesso necessario trasmettere le informazioni raccolte da un elemento grafico ad un altro. In MATLAB, questo è possibile grazie ai cosiddetti *handles*, che rappresentano uno degli ingressi delle funzioni *callback*. Essi permettono il riferimento ad un altro oggetto nel programma: richiamando la funzione *handles* riferita ad una determinata variabile, è possibile accedere a tale variabile anche da uno spazio di lavoro diverso. In questo modo, i vari *workspace* degli elementi nella GUI possono scambiare tra loro informazioni, raccoglierle dal *workspace base* e così via. Come si vedrà nel seguito, per lo sviluppo di questo programma si è fatto largo uso di queste funzioni.

2.2.2.2 Le proprietà degli elementi in una interfaccia grafica

Ogni elemento contenuto in una GUI possiede delle proprietà che lo caratterizzano e lo distinguono, non solo dagli elementi di diversa tipologia, ma anche da elementi dello stesso tipo. Alcune di esse sono raccolte in Fig. 2.6.
Grazie ad esse è possibile, ad esempio, distinguere due bottoni inserendo una scritta sopra di essi, decidere il numero massimo di selezioni possibili in una List Box, impostare la visibilità (o l'invisibilità, utile per i messaggi di errore o di avvertimento) di una Static Text, e così via. Le due proprietà più utilizzate per effettuare manipolazioni all'interno del codice sono *String* e *Value*. La prima può contenere una stringa che compare sopra o all'interno dell'elemento. Ad esempio, per realizzare un pulsante 'OK', è sufficiente creare un Push Button che riporti *OK* come *String*. La seconda proprietà è invece caratterizzata da un numero.

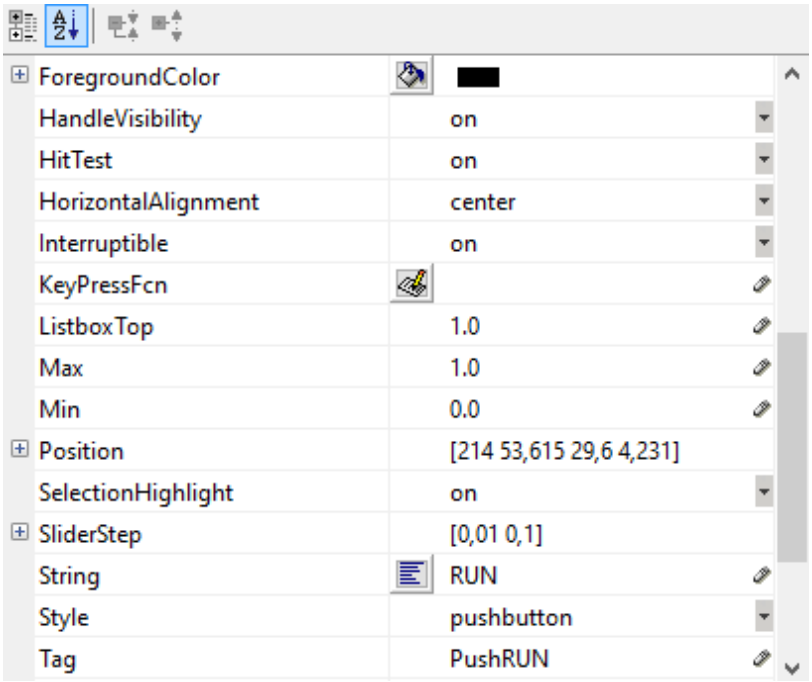

Fig. 2.6 - Alcune proprietà di un pushbutton

Nel caso delle liste (List Box o Pop-up Menu) tale valore definisce l'indice della voce selezionata all'interno della lista. Se, per esempio, una List Box riporta '2' come *Value*, significa che sulla GUI risulta selezionata la seconda voce dell'elenco. Nel corso della trattazione, verranno illustrate, volta per volta, le proprietà che sono state utilizzate e/o modificare per ottenere i risultati desiderati.

2.3 La sezione *General Simulation Setup*

Questa sezione dell'interfaccia è la prima che l'utente deve utilizzare per inizializzare il programma. Attraverso di essa è possibile importare nel software i dati orbitali del satellite e selezionare l'intervallo di tempo della simulazione. Quest'ultima operazione possa essere eseguita anche in un secondo momento.

Come visto nel Cap. 2.1.6, il primo passo da fare per stimare il memory budget attraverso il programma sviluppato è quello di caricare in esso (e dunque nel *workspace base* di MATLAB) le grandezze fornite dalla propagazione della missione, effettuata con GMAT. Esse vengono salvate nel report file fornito dal propagatore come una matrice di valori, con una formattazione tale che possano essere riconosciute, ad esempio, da una funzione MATLAB. In particolare, la formattazione utilizzata è *csv*, o *Comma-Separated Values*, con la quale è possibile i parametri secondo il seguente criterio: le componenti di uno stesso vettore sono riportate in colonna, dunque scorrono verticalmente lungo le righe; le componenti di vettori diversi sono invece tra di loro separate da una virgola, in orizzontale.

Il documento è quindi composto da vettori colonna, la cui componente i-esima è separata con una virgola dalla componente i-esima del vettore ad esso adiacente (si veda la Fig. 2.7).

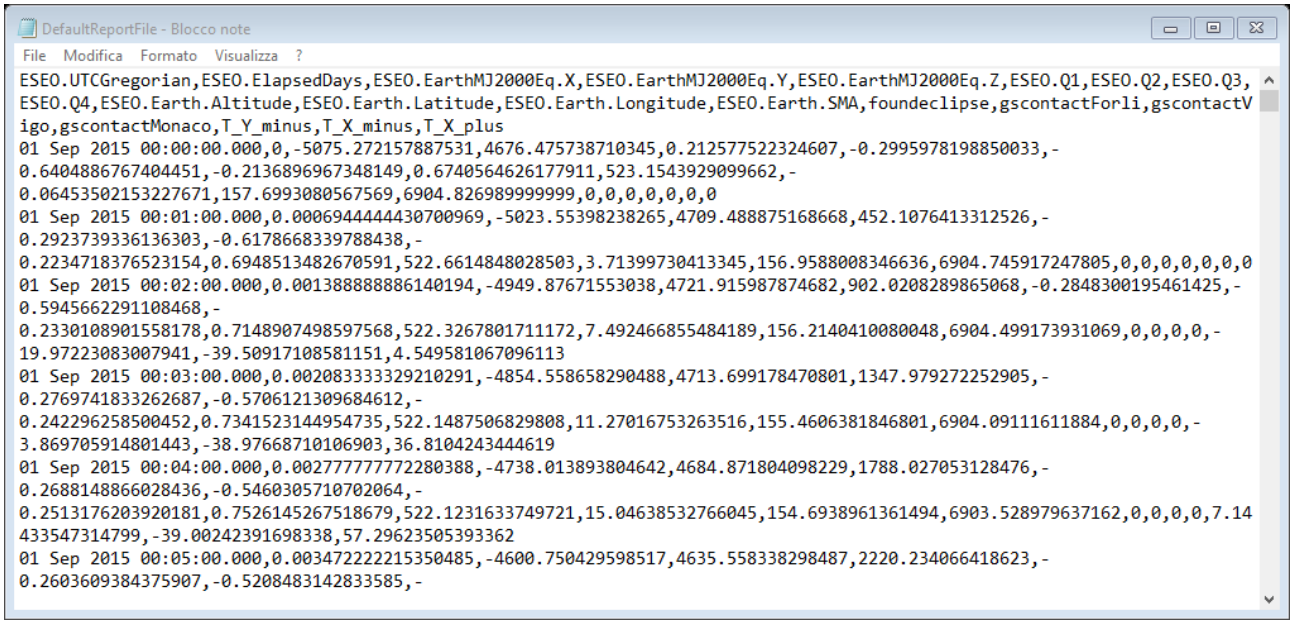

Fig. 2.7 - Un esempio di report file prodotto da GMAT

Si noti infine che la prima riga del documento riporta i nomi dei vettori sottostanti, anch'essi separati da una virgola (ad es. *ESEO.UTCGregorian, ESEO.ElapsedDays, …*).
L'importazione dei dati contenuti nel *report file* avviene grazie alla sezione denominata *General Simulation Setup*, la cui interfaccia è qui sotto riportata in Fig. 2.8.

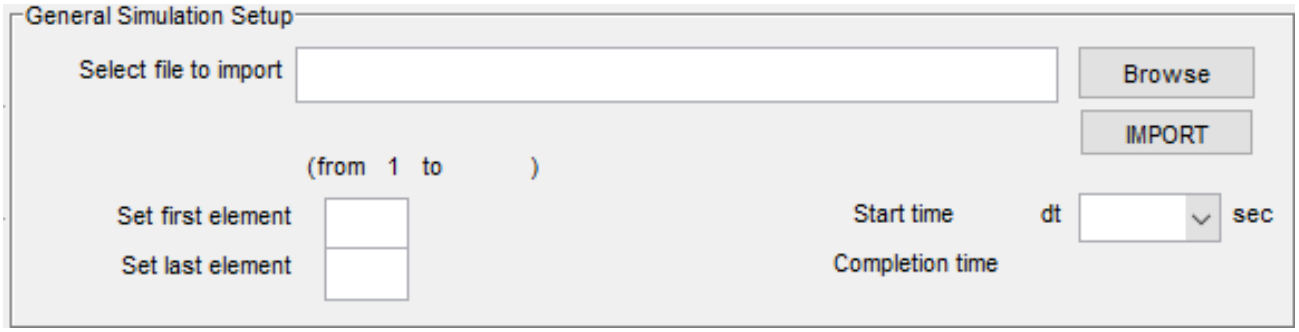

Fig. 2.8 - La sezione General Simulation Setup

Per facilitare al lettore i riferimenti tra GUI e codice, vengono riportati nella Tab. 2.1 i nomi degli elementi principali utilizzati nello script:

Elemento della GUI	Nome nel codice
Push Button ‘Browse’	PushBrowse
Push Button ‘IMPORT’	PushImport
Edit Text riferita a ‘Select file to import’	EditFileName
Edit Text riferita a ‘Set first element’	EditFrom
Edit Text riferita a ‘Set last element’	EditTo
Pop-up Menu riferito a ‘dt’	Listdt
Static Text contenente i passi della simulazione	StaticTo
Static Text mostrante ora e data di inizio	StaticStartTime
Static Text mostrante ora e data di fine	StaticCompletionTime

Tab. 2.1 – Gli elementi grafici della sezione General Simulation Setup

Poiché ogni elemento in una GUI possiede il proprio workspace ed il proprio algoritmo interno, è necessario utilizzare le funzioni *callback* e gli oggetti *handles* descritti nel paragrafo 2.2.2.1 per permettere di effettuare le operazioni di cui sopra. In particolare, la mappa che descrive le connessioni tra i vari elementi del codice è quella in Fig. 2.9. Le frecce indicano, in generale, un passaggio di informazioni tra i due elementi, solitamente da un workspace ad un altro.

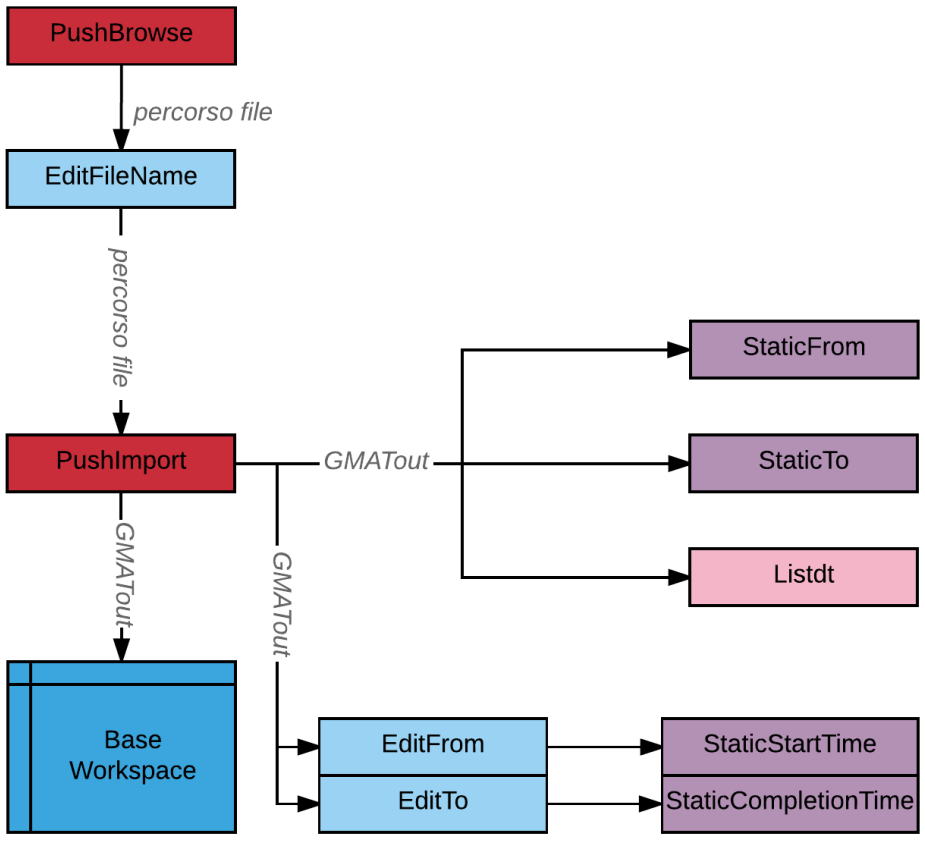

Fig. 2.9 - Il diagramma concettuale di General Simulation Setup

Per prelevare dalla memoria del computer il report, è necessario ottenere il suo percorso file all'interno del disco rigido. Ciò è possibile in due modi: l'utente può sia digitare direttamente il percorso file all'interno della casella di testo *EditFileName*, oppure servirsi del pulsante *PushBrowse* il quale, attraverso la funzione *built-in*[18] di MATLAB *uigetfile*, richiama la tipica interfaccia che il sistema operativo propone per cercare un documento. Tale funzione ha come prima uscita il nome del file e come seconda il percorso file ad essa correlato. Dopodiché, servendosi di un *handles*, viene rispedita la stringa contenente il percorso file a *EditFileName*, in particolare viene salvata nella proprietà '*String'*, all'interno dell'oggetto. Come anticipato, la proprietà *String* di un elemento della interfaccia grafica è quella che contiene la stringa da mostrare a video sull'oggetto stesso. Pertanto, nel momento in cui a tale voce viene assegnato il percorso file, l'utente può visualizzarlo nella Edit Box. Ovviamente, se l'utente sceglie di inserire a mano nella barra il percorso file, il pulsante *PushBrowse* può non essere utilizzato.

Dato che è stato assegnato alla proprietà *String* di una *EditBox*, il percorso file può essere richiamato tramite l'*handle* della *EditBox EditFileName*, in modo tale da essere in seguito accessibile da altri elementi. A tal proposito viene utilizzato il pulsante *PushImport* che, se premuto, legge il percorso file e svolge una serie di operazioni. Innanzi tutto, esso memorizza il percorso, che diventa l'argomento della funzione *plot_GMAT_results*. Quest'ultima funzione permette, nel nostro caso, di salvare i dati nel *workspace base*, cosicché essi siano disponibili per la configurazione del satellite. La funzione *plot_GMAT_results*, sviluppata in precedenza da Sitael e qui utilizzata come libreria, permette sia di salvare in MATLAB i dati contenuti nel report file fornito da GMAT. Tramite questa funzione, il report viene salvato come una variabile *struct*[19] i cui campi sono i titoli delle grandezze presenti nel file .csv, nella forma mostrata in Fig. 2.10.

[18] Le funzioni *built-in* sono quelle già presenti all'interno del database di MATLAB, quindi non create dall'utente o da terze parti.

[19] Gli *struct* sono oggetti di MATLAB nei quali è possibile memorizzare altri oggetti, anche diversi tra loro, ed organizzarli in vari campi. Ad esempio, una struct può contenere una stringa, un vettore cella, una matrice ed un'altra struct contemporaneamente.

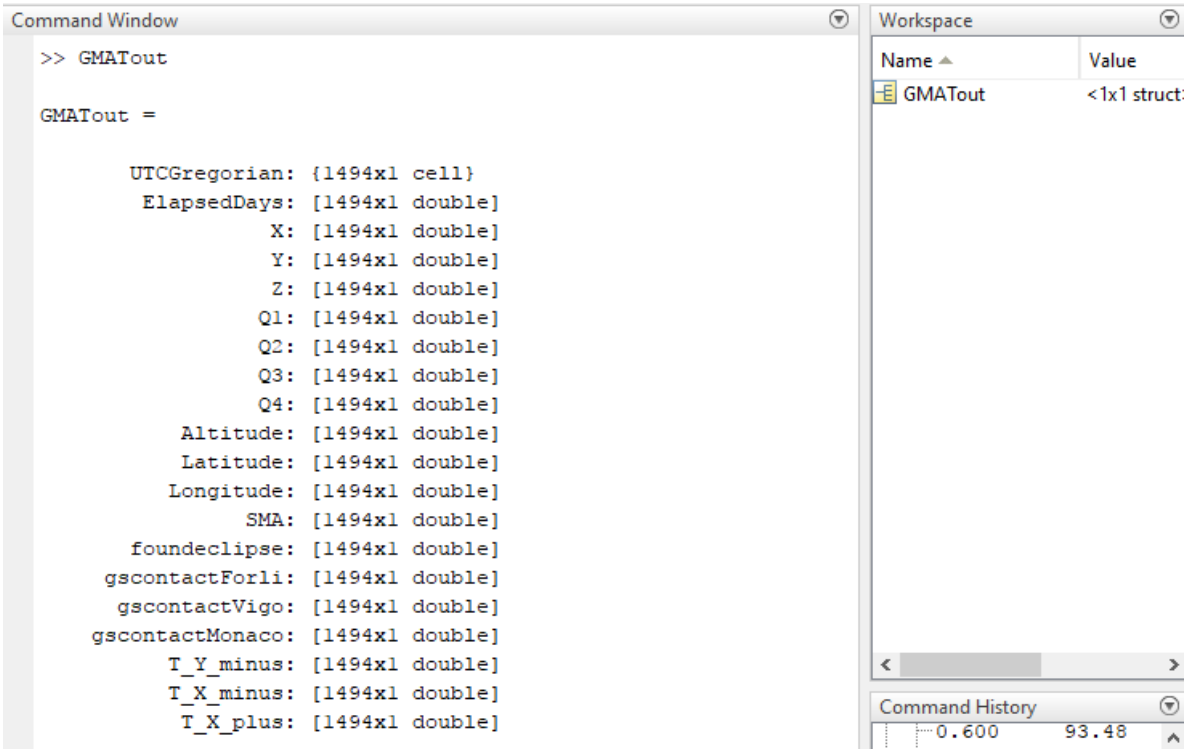

Fig. 2.10 - GMATout richiamato dalla command window

Il *report file* utilizzato per la trattazione proviene da una propagazione della missione ESEO. La durata della simulazione è di circa un giorno, con un passo temporale di 60 secondi, per un totale di 1494 passi temporali. Ogni campo (*GMATout.UTCGregorian, GMATout.ElapsedDays, ...*) è un vettore colonna di *double*[20] (eccetto *GMATout.UTCGregorian*[21] che, contenendo il calendario, utilizza i *vettori cella*[22]) che raccoglie l'evoluzione delle varie grandezze nel tempo. *GMATout* viene richiamato all'interno del *workspace* di *PushImport*, ma per essere utilizzato da altri elementi del programma, esso deve essere trasferito al *workspace base.* Per salvare un elemento nel *workspace base*, è stata usata la funzione *assignin*, con la quale si può salvare, in un altro spazio di lavoro, una variabile presente nel *workspace* dentro cui è definita la funzione.

Dopo aver caricato il report nel *workspace base*, è possibile aggiornare alcuni elementi della GUI in modo da fornire all'utente un maggior numero di informazioni. In particolare, sfruttando la funzione

[20] Il *double precision arrays* è il modo con cui MATLAB registra, di default, i valori numerici. I *double* sono numeri in virgola mobile ed in doppia precisione, cioè rappresentati utilizzando 64 bit.

[21] *UTCGregorian* fa riferimento al modo in cui GMAT rappresenta gli intervalli temporali: UTC è riferito al Tempo Coordinato Universale (il fuso orario basato sul tempo medio di Greenwich), Gregorian indica il calendario gregoriano, usato nella maggior parte dei paesi occidentali, compresa l'Italia.

[22] Così come le struct, anche i vettori cella possono contenere qualsiasi tipo di elemento al loro interno. La differenza sostanziale è che gli elementi dei vettori cella, come le matrici, sono indicizzati, mentre le struct sono suddivise in campi.

length si ottiene il numero degli elementi contenuti nel calendario, cioè il numero di passi della propagazione (1494 in questo caso), che viene mostrato nella casella *StaticTo*.
Inoltre sono valutate, con la funzione *diff,* le ampiezze di ogni passo della simulazione (ovvero dei *time-step*[23]), raccolte poi nella lista chiamata *Listdt*.
Per finire, con *PushImport* viene aggiornata la lista delle stazioni di terra (*ListGS*) presente nella sezione dedicata alla configurazione dei trasmettitori. Su *ListGS*, dopo aver premuto 'IMPORT', comparirà la lista con i campi presenti in *GMATout*, in modo tale che sarà possibile per l'utente selezionare quelli dedicati alle stazioni di terra. Una volta che il programma dispone delle variabili contenute in *GMATout*, l'utente può impostare a piacere l'intervallo di simulazione, che deve essere compreso tra i valori indicati da *StaticFrom* e *StaticTo*. Da notare che questi ultimi due elementi grafici indicano rispettivamente la prima e l'ultima componente del vettore *GMATout.ElapsedDays*; pertanto, l'intervallo deve essere impostato dall'utente in termini di componenti del vettore appena citato (ad esempio, all'istante iniziale si può assegnare 1 e all'istante finale 1494). Nel momento in cui le componenti per l'istante iniziale e quello finale vengono inseriti, i *callback* di *EditFrom* ed *EditTo* mostrano all'utente data e ora, sul calendario, dei tempi di simulazione selezionati. Quest'ultima informazione viene assegnata alla proprietà *String* degli elementi *StaticStartTime* e *StaticCompletionTime* e dunque compaiono a schermo di fianco a *EditFrom* ed *EditTo.* Se le operazioni vengono svolte correttamente, questa sezione si presenta compilata in modo simile a quello in Fig. 2.11.

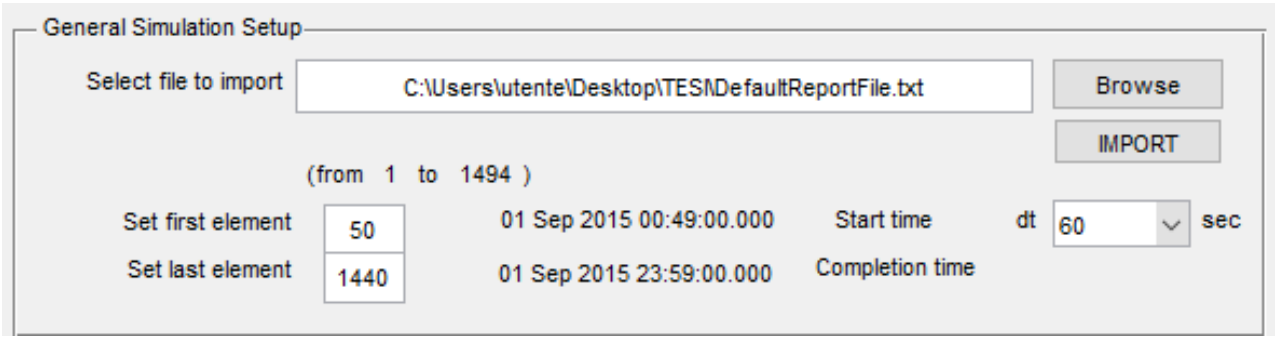

Fig. 2.11 – Un esempio di come può essere compilata General Simulation Setup

[23] I *time-step* sono i passi della simulazione, lungo il vettore tempo. Dato che la simulazione è discreta e non continua, essi sono separati da un certo lasso temporale. Minore è tale lasse temporale, maggiore è la risoluzione della simulazione.

Una volta completata questa prima sezione dell'interfaccia grafica, il software ha acquisito le informazioni necessarie per procedere con la configurazione. Nei paragrafi a seguire, pertanto, verranno illustrate le strategie di progetto utilizzate per creare il modello di scambio dati.

2.4 I componenti del satellite

Prima che il programma possa eseguire i calcoli veri e propri per il memory budget del satellite è necessario definire tutte le condizioni al contorno necessarie all'algoritmo *runtime* (chiamato nel codice *mem_bud_run*). Come visto il Fig. 2.2, le informazioni sono raccolte, con l'intervento dell'utente nella GUI, nelle quattro categorie 'sensori', 'bus', 'memorie' e 'trasmettitori'. Queste quattro categorie vengono salvate sotto forma di *struct* nel *workspace base* di MATLAB. Così facendo, esse possono essere richiamate da eventuali funzioni esterne, come i *callback* degli elementi nella GUI o la funzione *runtime*. Analogamente a quanto fatto per il codice descritto nel paragrafo 2.3, anche in questo caso si farà ampio uso della funzione *assignin* per salvare le grandezze nel *workspace base*. La scelta di salvare tali informazioni all'interno di una *struct* è supportata dal fatto che, queste ultime, possono contenere in un unico elemento MATLAB informazioni di diverso tipo: double, vettori, vettori cella, altre struct, stringhe, ecc. Questa caratteristica delle *struct* è particolarmente utile nel nostro caso, in cui si vogliono salvare in un unico oggetto (ad es. quello dei sensori) informazioni di vario genere (ad es. nome del sensore, *Data Rate*, logica di attivazione, ecc.).
Il codice sviluppato, quindi, permette lo scambio di informazioni tra utente e macchina così come illustrato in linea generale nel diagramma di Fig. 2.12: le quattro sezioni della GUI raccolgono le informazioni sulla configurazione del satellite, all'interno dei vari workspace degli elementi grafici. In seguito, le informazioni vengono passate al workspace base e rese disponibili per il memory budget. I dettagli a riguardo verranno trattati nei successivi paragrafi.

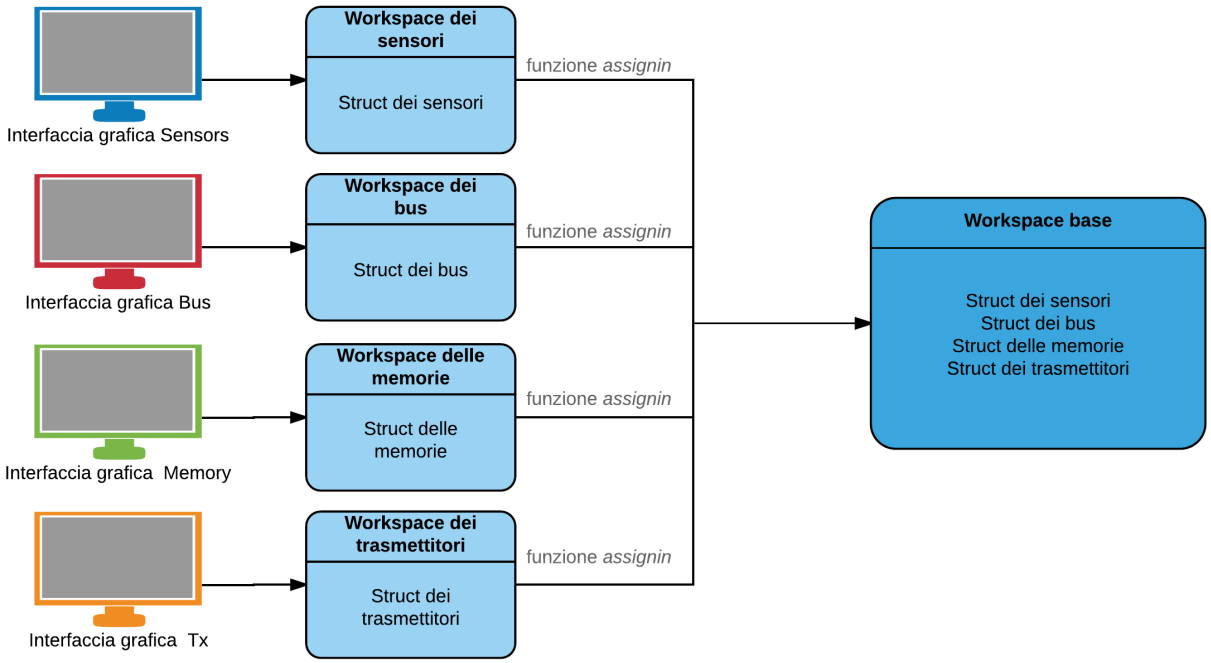

Fig. 2.12 – Diagramma di alto livello che mostra la modalità di caricamento dei dati nel programma

2.5 La sezione *Sensors*

Nel paragrafo 1.5 è stato illustrato il criterio con cui si è scelto di modellizzate la rete di scambio dati del satellite da analizzare. In accordo con lo schema mostrato allora, i sensori sono il primo elemento della catena poiché sono di fatto la sorgente delle informazioni (e dunque dei bit) a bordo del satellite. Si precisa che, con il termine 'sensore', in questo contesto si intende tutto l'insieme di strumenti che introducono a bordo del satellite una certa quantità di dati: payload, strumenti per *housekeeping*, sensori d'assetto, ecc. Dati i requisiti del software e le operazioni da svolgere in esso, la *struct* dei sensori (nominata *s* all'interno del programma) è stata suddivisa nei campi illustrati in Tab. 2.2. Oltre a questi, *s* contiene anche dei campi ausiliari il cui scopo verrà illustrato in un secondo momento.
Lo scopo della sezione *Sensors* della GUI è quello di permettere all'utente di compilare i campi appena descritti nel modo più semplice e rapido possibile a partire dalle specifiche tecniche e di missione del componente. Non appena si avvia il programma, la sezione *Sensors* si presenta come in Fig. 2.13.

Campo	Descrizione
s.name	Nome del sensore. È una stringa di caratteri
s.data	*Data Rate* di acquisizione del sensore, in *bps* sotto forma di numero *double*
s.activation_condition	Stringa logica di attivazione del sensore. È una stringa di caratteri
s.attached_to	Nome del *bus* attraverso cui il sensore scarica i dati. È una stringa di caratteri
s.buffer	Memoria interna del sensore, in bit. È un *double*

Tab. 2.2 – I campi della struct dedicata ai sensori

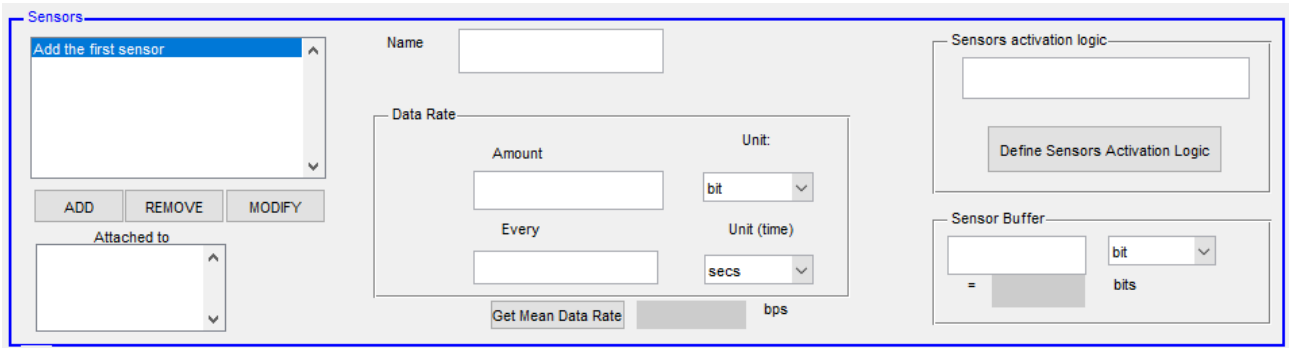

Fig. 2.13 – La sezione Sensors, dedicata alla configurazione dei sensori

Analogamente a quanto fatto per *General Simulation Setup*, vengono riportati in Tab. 2.3 i nomi dei principali elementi contenuti nella sezione *Sensors*.

Elemento della GUI	Nome nel codice
Edit Text riferita a ‘Name’	EditSensName
Edit Text riferita a ‘Amount’	EditAmount
Edit Text riferita a ‘Every’	EditEvery
Edit Text riferita al buffer del sensore	EditSensBuffer
Edit Text riferita alla logica di attivazione	pay_act_logic_edit
Pop-up List riferita a ‘Amount’ (unità di misura dei dati)	PopUpBit
Pop-up List riferita a ‘Every’ (unità di misura del tempo)	PopUpTime
Pop-up List riferita al buffer del sensore (unità di misura dei dati)	PopUpBit5
Listbox contenente i sensori aggiunti	ListSens
Pushbutton ‘ADD’	PushAddSens
Pushbutton ‘REMOVE’	PushRemoveSens
Pushbutton ‘MODIFY’	pushbutton27
Listbox contenente i bus aggiunti (riferita ad ‘Attached to’)	ListSensToBus

Pushbutton ‘Get Mean Data Rate’	PushGetDataRate
Static Text riferito a Get Mean Data Rate	StaticDataRate
Static Text riferito al buffer del sensore	StaticBuffer
Pushbutton ‘Define Sensor Activation Logic’	pushbutton26
Static Text segnalante un errore (invisibile di default)	StaticMissingError1

Tab. 2.3 - Gli elementi grafici della sezione Sensors

Per permettere all’utente una compilazione che sia la più rapida possibile, è stato necessario introdurre in questa sezione un elevato numero di elementi grafici, correlati tra loro come mostrato in Fig. 2.14. I cerchi grigi indicano il fatto che, le informazioni dei due oggetti in entrata, si combinano insieme generando un’unica uscita.

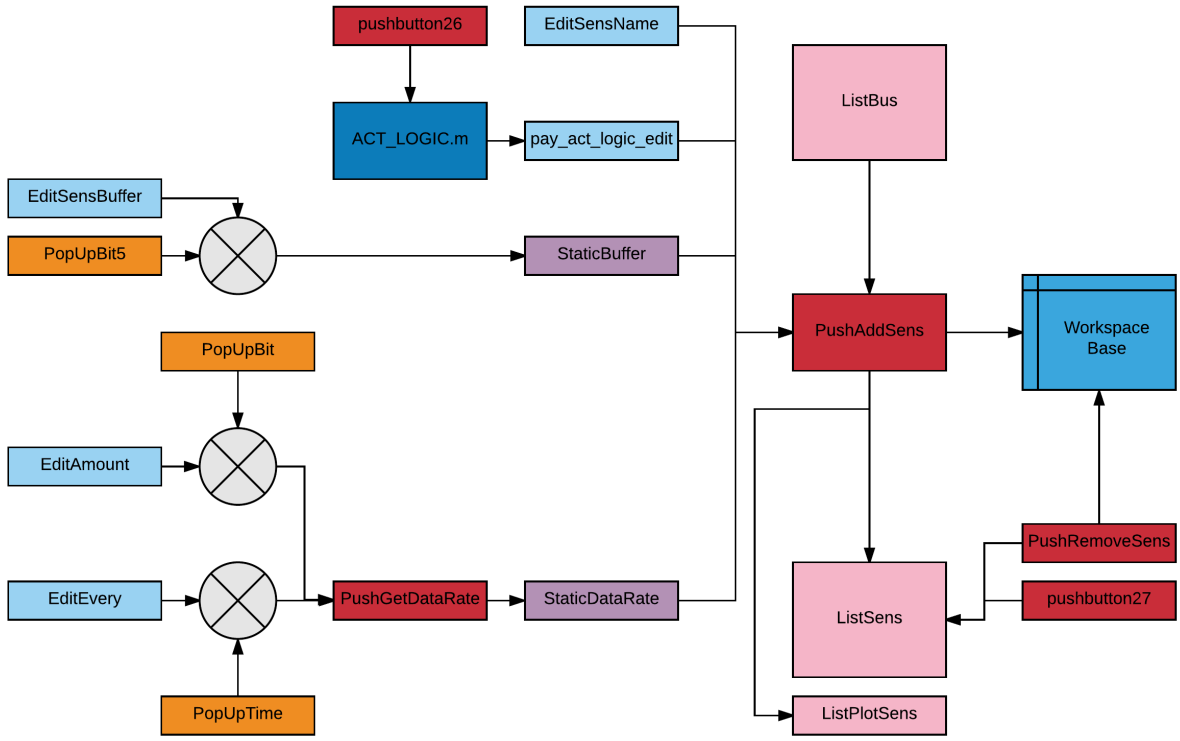

Fig. 2.14 – Il diagramma concettuale di Sensors

2.5.1 La compilazione di *Sensors*

Per configurare un sensore è necessario compilare tutte le voci presenti nella sezione. Per cominciare, è possibile inserire il nome del sensore che si vuole aggiungere al modello all’interno di *EditSensName*. L’utente deve poi definire il *Data Rate* con cui il sensore inserisce dati all’interno del satellite, servendosi delle Edit Text relative ad *Amount* ed *Every*, che sono rispettivamente

EditAmount ed *EditEvery*. Nella prima delle due si deve inserire il numero di bit acquisiti nell'intervallo di acquisizione, mentre nella seconda va inserito l'intervallo di acquisizione stesso. Grazie a *PopUpBit* e *PopUpTime,* è possibile servirsi dei vari multipli delle unità di misura utilizzate, così che si possa esprimere il valore all'interno di *EditAmount* anche in kbit, Mbit, Tbit, kbyte, ecc. ed il valore in *EditEvery* in minuti, ore, giorni. Quest'ultima funzionalità è resa possibile grazie a *PushGetDataRate,* nel cui *callback* è presente un algoritmo che moltiplica i valori presenti in *EditAmount* ed *EditEvery* a seconda dell'unità di misura scelta nelle suddette Pop-up List. In questo modo, a prescindere dalla scelta dell'utente, vengono convertiti i due valori rispettivamente in bit ed in secondi e viene calcolato il *Data Rate* con la semplice espressione

$$Data\ Rate(bps) = EditAmount(bit) * EditEvery(\text{sec})$$

Il *Data Rate* espresso in bps viene poi mostrato all'utente nella casella di testo *StaticDataRate*, servendosi degli *handles* che permettono di comunicare tra *PushGetDataRate* e la casella di testo appena citata. Si noti che, per come è stato definito, il *Data Rate* è un valore che riporta soltanto il numero di bit acquisiti nell'intervallo di simulazione specificato, pertanto non è sensibile ad eventuali picchi o valli presenti nell'andamento del flusso di dati stesso. Per questo motivo, il Pushbutton presente sulla GUI che determina il valore finale del *Data Rate* è stato etichettato con 'Get Mean Data Rate'. Nonostante ciò, se l'utente è interessato a modellizzare il sensore in modo tale che acquisisca le informazioni secondo 'picchi' discreti di dati, questo è possibile usando le opportune condizioni di attivazione, come si vedrà nel seguito.

Un'altra voce da compilare in questa sezione è quella riguardante il buffer del sensore, ovvero la memoria interna di cui esso dispone. Infatti, tra i requisiti del software rientra anche la modellizzazione dell'andamento nel tempo dei buffer, così che l'utente possa constatare che il flusso di dati non superi il limite fisico consentito da questa memoria interna. Per definire questa grandezza è sufficiente che l'utente inserisca nella casella *EditSensBuffer* la capacità del sensore, utilizzando i multipli dei bit e dei byte suggeriti dalla lista *PopUpBit5.* Dopo aver selezionato una delle voci in quest'ultima lista, il suo *callback* aggiornerà in automatico la stringa visibile su *StaticBuffer*, mostrando il valore del buffer convertito in bit.

Per questo modello è inoltre necessario conoscere gli istanti in cui il sensore in questione è attivo e dunque gli intervalli di tempo in cui vengono acquisiti i dati. Questo viene implementato nel codice attraverso una logica di attivazione, cioè una frase MATLAB formulata imponendo una condizione; tale condizione può essere vera o falsa. Se è vera, restituisce 1, altrimenti 0. Negli istanti in cui l'uscita

del logico vale 1, il sensore è attivo. Il modo in cui il software manipola la condizione di attivazione verrà meglio illustrato nel capitolo dedicato alla funzione *runtime*. La condizione di attivazione deve essere inserita dall'utente nella casella di testo chiamata *pay_act_logic_edit* e deve essere scritta in linguaggio MATLAB. Ad esempio, se venisse scritto '1==0', la logica di attivazione restituirebbe sempre '0' in uscita, dato che 1 è diverso da 0. Questo porterebbe ad avere il sensore spento per tutta la durata della simulazione. Secondo lo stesso criterio, scrivendo '1==1' il sensore rimarrebbe attivo dall'istante iniziale all'istante finale, poiché 1 è sempre identico a 1.

2.5.1.1 La GUI secondaria per la condizione di attivazione

Dato che il software dispone dei dati orbitali forniti da GMAT e caricati nella sezione *General Simulation Setup*, può risultare utile definire la condizione di attivazione di un particolare sensore sfruttando le variabili contenute nel *report file*. Poiché tali parametri variano nel tempo, in questo modo è possibile riprodurre una simulazione in cui il sensore si attiva solo in certi istanti temporali. Ad esempio, se si vuole modellizzare un sensore che si attiva solo qualora il satellite non è in eclissi, si può scrivere, in *pay_act_logic_edit*, la stringa:

$$(GMATfile.foundeclipse(t) == 0)$$

dove si ricorda che *GMATfile* è la *struct* contenente tutti i dati forniti dal propagatore, e *foundeclipse* è il campo contenente la funzione che vale 0 se il satellite non è in eclissi ed 1 altrimenti. Si noti che tale condizione viene definita istante per istante (questo è il motivo per cui *GMATfile.foundeclipse* ha come argomento *t*, che rappresenta l'intervallo temporale e quindi la dipendenza del tempo).
Per definire condizioni di attivazione più complesse l'utente può servirsi del Push Button che cita 'Define Sensors Activation Logic' (*pushbutton26*), grazie al quale viene avviata una seconda interfaccia grafica (Fig. 2.15) che permette di 'assemblare' la propria logica di attivazione in modo intuitivo. Quest'ultima è stata sviluppata da Sitael ed è stata implementata all'interno della GUI generale richiamandone il codice (ACT_LOGIC.m) nel *callback* di *pushbutton26*.

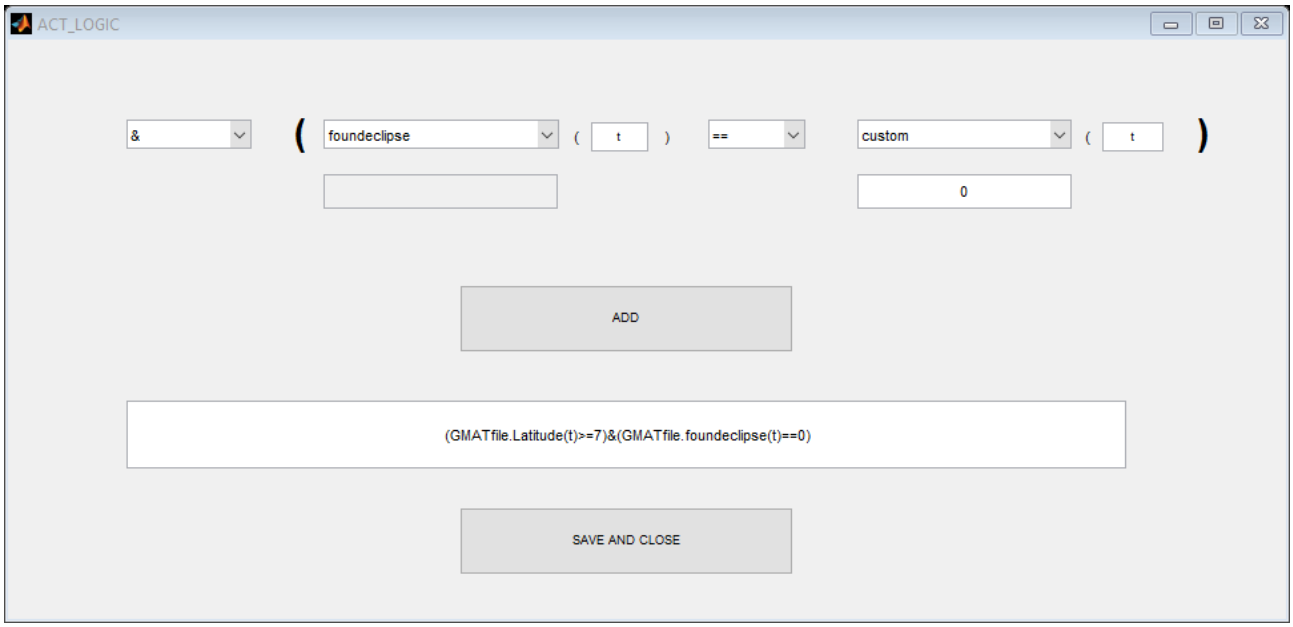

Fig. 2.15 – L'interfaccia grafica di ACT_LOGIC.m, compilata con un esempio

Una volta scritta la stringa, è possibile riportarla in *pay_act_logic_edit* con il pulsante 'SAVE AND CLOSE' presente sulla GUI secondaria.

2.5.1.2 L'interfaccia con i bus

Prima di poter aggiungere alla struct *s* il sensore configurato, è necessario anche definire il bus verso cui esso scarica i dati acquisiti. Se non fosse stato fatto prima, dunque, è necessario in primis configurare un bus ed aggiungerlo alla *struct* chiamata *b* (per ulteriori dettagli si veda il paragrafo 2.6). In questa trattazione, si supponga di aver già definito uno o più bus in precedenza. Essi saranno allora automaticamente elencati nella lista chiamata *ListSensToBus*, che definisce tutte le possibili connessioni che il sensore può effettuare con i bus a valle. Dopo aver selezionato il bus di interesse, viene attribuito alla proprietà *Value* della lista un valore numerico che coincide con la posizione nella lista del componente selezionato. Quindi, se viene selezionato il primo componente nella lista, *Value* conterrà il valore 1, il secondo componente assegnerebbe il valore 2 e così via. Si noti dalla Fig. 2.16 che di default *Value* assume il valore 1. Quindi, senza interazioni dell'utente, risulta selezionata dall'elenco la prima voce disponibile.

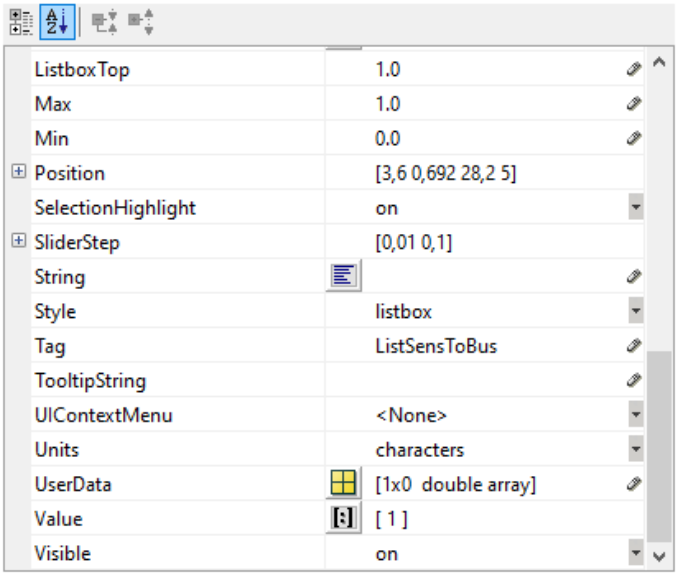

Fig. 2.16 - La lista delle proprietà di ListSensToBus

2.5.2 Aggiungere il nuovo sensore nel modello

Una volta che tutte le voci suddette sono state compilate, il programma è in grado di salvarle all'interno dei campi elencati in Tab. 2.2. Per fare ciò, è sufficiente che l'utente prema il tasto 'ADD' (*PushAddSens*), nel cui *callback* è contenuta una serie di operazioni.

In primo luogo, il codice verifica che non ci siano voci mancanti nelle caselle di testo fondamentali all'interno dell'interfaccia (*EditSensName, StaticDataRate, pay_act_logic_edit, EditSensBuffer, ListSensToBus*), dato che l'eventuale presenza di campi vuoti potrebbe portare ad avere errori in fase di simulazione. L'operazione di controllo viene eseguita attraverso cinque variabili chiamate *TF* (che sta per 'True or False'), che valgono 1 se il campo da esse osservate è vuoto e 0 altrimenti.

Nel caso in cui uno dei cinque elementi controllati non sia stato compilato dall'utente, viene mostrato a video un messaggio di errore correlato al campo in questione, come nel caso in Fig. 2.17, nel quale è stata tralasciata la stringa di attivazione del sensore.

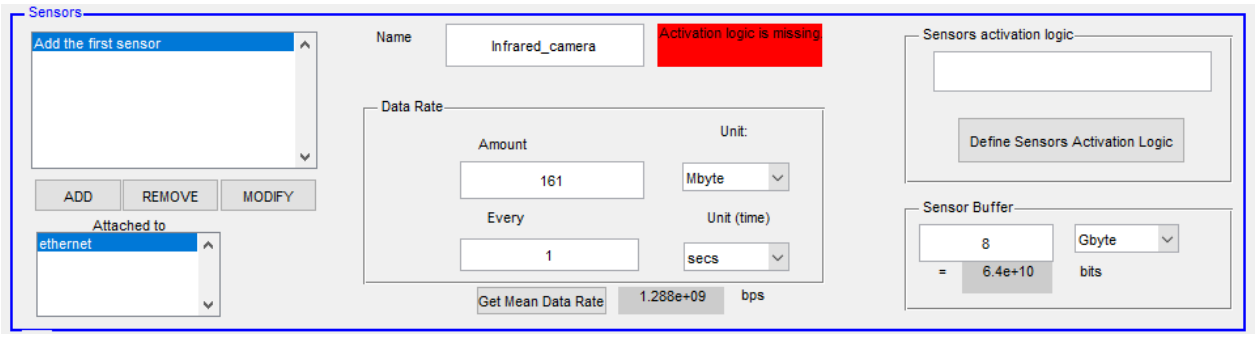

Fig. 2.17 - Nel momento in cui viene premuto il pulsante ADD, il programma avverte della mancanza della stringa logica di attivazione.

Dopo aver superato tutti i controlli, il codice del pulsante 'ADD' rende invisibili eventuali messaggi di errore comparsi precedentemente e procede ad assegnare ad ogni campo di *s* il valore corrispondente inserito dall'utente. Per comodità, prima di essere prelevate, le grandezze vengono convertite in bit ed in secondi, così che non si possa verificare alcuna ambiguità dovuta alle unità di misura.

2.5.3 Il criterio di salvataggio del sensore all'interno del workspace

L'algoritmo permette di salvare nel programma un generico numero *k* di sensori, in modo tale che il primo sensore caricato viene assegnato alla struct *s(1)* ed il *k*-esimo sensore viene assegnato ad *s(k)*. Per segnalare al codice in quale componente di *s* debba essere salvato il nuovo sensore, si sfruttano le proprietà della GUI.

2.5.4 Il caricamento del sensore nell'interfaccia grafica

Dopo che l'utente ha cliccato su *PushAddSens*, vengono inoltre aggiornate tutte le List Box contenenti l'elenco dei sensori nel modello, cioè *ListSens* e *ListPlotSens* (quest'ultima verrà trattata nel paragrafo 2.9.4).

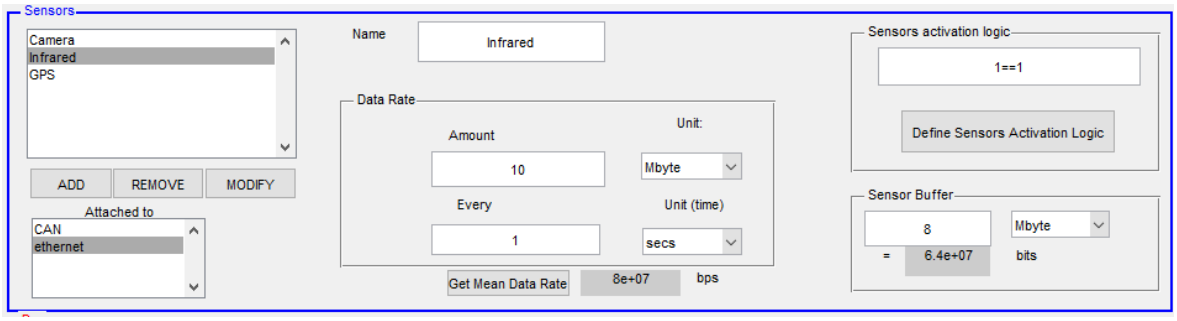

Fig. 2.18 - Un esempio di configurazione completa di tre sensori

In Fig. 2.18 è mostrato un esempio in cui sono stati caricati nel programma tre sensori. Si noti che, sulla sinistra, sono presenti le due List Box contenenti i sensori (*ListSens*) ed i bus (*ListSensToBus*) che sono presenti nel programma. Di default, all'interno dei vari campi dell'interfaccia grafica vengono visualizzate le informazioni dell'ultimo sensore configurato o aggiunto. Tuttavia, la GUI è stata programmata per far sì che, qualora l'utente clicchi su un qualunque sensore presente in *ListSens*, vengano visualizzate in essa tutte le informazioni relative a quello specifico sensore. In altre parole, se l'utente si sposta da un elemento ad un altro in *ListSens*, il codice aggiorna automaticamente i

restanti campi (nome, condizione di attivazione, bus collegato, ecc.) per renderli coerenti con la voce selezionata.

All'interno del *workspace base*, MATLAB salva i sensori caricati così come mostrato in Fig. 2.19.

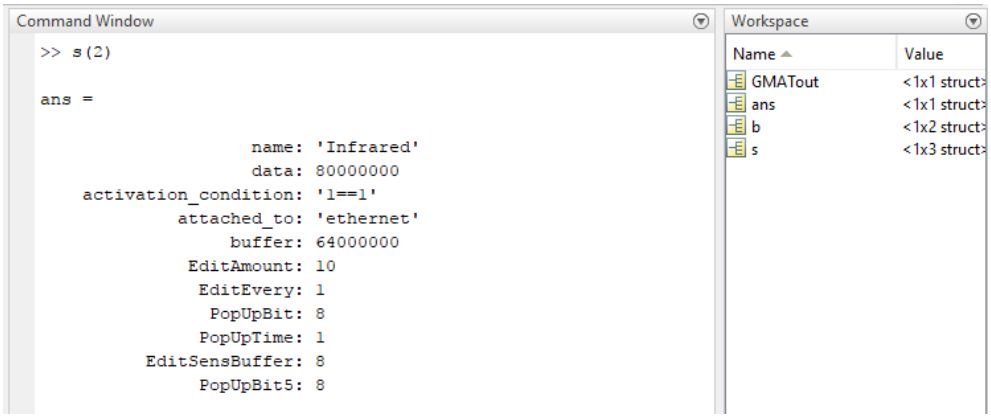

Fig. 2.19 - La seconda componente della struct 's' nel workspace base

Si noti che il nome, la condizione di attivazione ed il bus connesso vengono salvati come stringhe, i dati inseriti dall'utente sono salvati in bit o in secondi, a seconda del caso, mentre le voci selezionate dai Pop-up Menu (ad esempio per stabilire i multipli dei bit e dei secondi) sono rappresentate dalla posizione della voce selezionata all'interno della lista (viene quindi salvato il valore di *Value* della lista stessa).

2.5.5 Rimozione e Modifica degli elementi

Per espandere ulteriormente le funzionalità dell'interfaccia grafica, sono stati implementati due ulteriori pulsanti: il tasto 'REMOVE' (*PushRemoveSens*) ed il tasto 'MODIFY' (*pushbutton26*) che permettono all'utente rispettivamente di rimuovere e modificare la componente della struct selezionata in *ListSens*.

Il tasto 'REMOVE', se cliccato, valuta la struct *s* e l'indice dell'elemento attualmente selezionato nella lista *ListSens*. L'algoritmo genera dunque una nuova lista di sensori, nella quale viene esclusa la voce selezionata in precedenza, e la assegna alle List Box correlate a questa sezione (*ListSens* e *ListPlotSens*); inoltre, esso rimuove dalla struct la voce selezionata, così che l'elemento non esista più né all'interno dell'interfaccia grafica né nel *workspace base*.

Per cambiare una o più voci correlate ad un elemento aggiunto in precedenza, ci si serve del tasto 'MODIFY', che deve essere premuto al termine delle modifiche apportate ai campi della GUI. Infatti,

questo pulsante valuta la struct dei sensori ed assegna i valori modificati ai rispettivi campi, andando di fatto a sovrascrivere i valori definiti in precedenza. Il codice è molto simile a quello contenuto del *callback* del pulsante 'ADD': aggiorna tutte le voci dell'interfaccia grafica (e del workspace base) che sono in qualche modo collegate alle modifiche svolte.

2.6 La sezione *Bus*

L'obiettivo della sezione dedicata ai bus è quello di permettere all'utente la creazione di uno o più bus, ciascuno con il rispettivo *Data Rate*, eventualmente inserendo nell'interfaccia informazioni su overhead e/o compressione.
In un satellite, i BUS sono i canali di comunicazione attraverso cui i componenti possono scambiare informazioni e quindi bit. Rappresentano dunque il mezzo di trasporto dei dati tra sensori e memorie così come tra memorie e trasmettitori, come mostra la Fig. 1.3. Si è già visto nel paragrafo 2.5 che, per completare la configurazione di un sensore, è necessario specificare il bus ad esso connesso. Pertanto, nel programma è opportuno in primis effettuare la configurazione dei bus, così che sia possibile procedere al salvataggio anche di sensori e memorie.
La specifica principale di un bus è il *Data Rate*, cioè la quantità di informazioni per unità di tempo che il bus riesce a trasportare (si veda la definizione nel paragrafo 1.5). Il *Data Rate* può essere influenzato, positivamente o negativamente, da due parametri detti *overhead* e *compressione*. Infatti, i dati che deve trasmettere un bus non sempre sono i meri dati di acquisizione di un sensore, bensì è possibile che siano presenti delle aggiunte e/o delle modifiche di cui è necessario tenere conto all'interno del memory budget.

In particolare, l'overhead consiste in una serie di informazioni aggiuntive che è necessario fornire al ricevente (umano o macchina che sia) per rendere possibile la ricostruzione dei pacchetti di dati trasmessi. Come è logico pensare, l'overhead rappresenta una aggiunta di informazioni nel flusso di dati, pertanto riduce lo spazio disponibile per unità di tempo, dedicato ai dati di acquisizione.

La compressione è invece un processo che, tramite opportuni algoritmi, permette di ridurre la quantità di dati richiesta per trasmettere un certo numero di informazioni. Esiste la compressione cosiddetta *lossy*, che porta alla perdita di una parte dei dati originari e la compressione *lossless* che, nonostante riduca il numero di bit durante la trasmissione (ad esempio all'interno di un bus), non provoca perdite.

Al termine della configurazione, gli unici elementi che vengono salvati nella struct dedicata ai bus (chiamata *b*), sono quelli in Tab. 2.4.

Campo	Descrizione
b.name	Nome del bus. È una stringa di caratteri
b.data	*Data Rate* di trasmissione permesso dal bus, in *bps* sotto forma di numero *double*.

Tab. 2.4 – I campi della struct dedicata ai bus

La sezione dedicata ai bus (Fig. 2.20) è graficamente meno complessa di quella pertinente ai sensori, perché in questo caso è necessario un numero inferiore di informazioni caratteristiche.

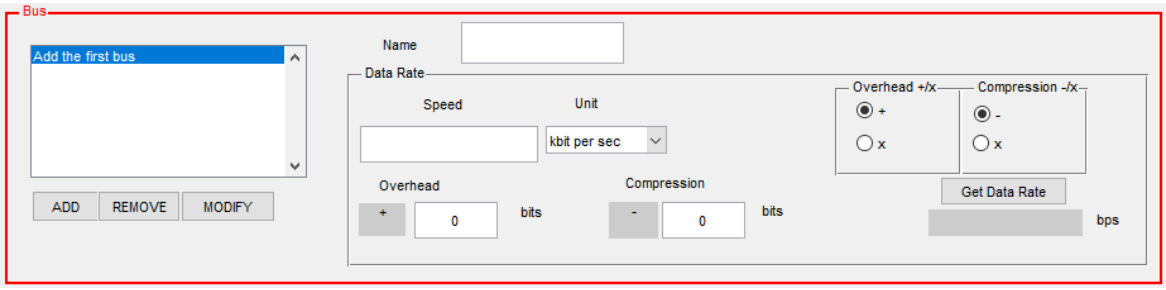

Fig. 2.20 - La sezione Bus, dedicata alla loro configurazione

Le specifiche tecniche quali *Data Rate*, overhead e compressione sono state raccolte grazie alla creazione degli elementi grafici descritti in Tab. 2.5.

Elemento della GUI	Nome nel codice
Edit Text riferita a 'Name'	EditBusName
Edit Text riferita a 'Speed'	EditSpeed
Pop-up List riferita a 'Unit' (unità di misura del *Data Rate*)	PopUpBit2
Radio Button in alto a sinistra	RadioPlus1
Radio Button in basso a sinistra	RadioTimes1
Radio Button in alto a destra	RadioPlus2
Radio Button in basso a destra	RadioTimes2
Static Text riferita a 'Overhead', sulla sinistra	StaticPlusOrTimes1
Static Text riferita a 'Overhead', sulla destra	StaticBitOrByte1
Static Text riferita a 'Compression' sulla sinistra	StaticPlusOrTimes2
Static Text riferita a 'Compression', sulla destra	StaticBitOrByte2
Edit Text riferita a 'Overhead'	EditOverhead
Edit Text riferita a 'Compression'	EditCompression
Push Button 'Get Data Rate'	PushGetDataRate2

Static Text riferito a ‘Get Data Rate’	StaticDataRate2
List Box contenente i bus	ListBus
Pushbutton ‘ADD’	pushbutton3
Pushbutton ‘REMOVE’	pushbutton45
Pushbutton ‘MODIFY’	pushbutton28
Static Text segnalante un errore (invisibile di default)	StaticMissingError2

Tab. 2.5 - Gli elementi grafici della sezione Bus

Il criterio con cui gli elementi interagiscono tra loro nel codice è invece mostrato in Fig. 2.21, sempre attraverso un diagramma concettuale che collega i blocchi dei vari elementi, chiamati così come lo sono sul codice.

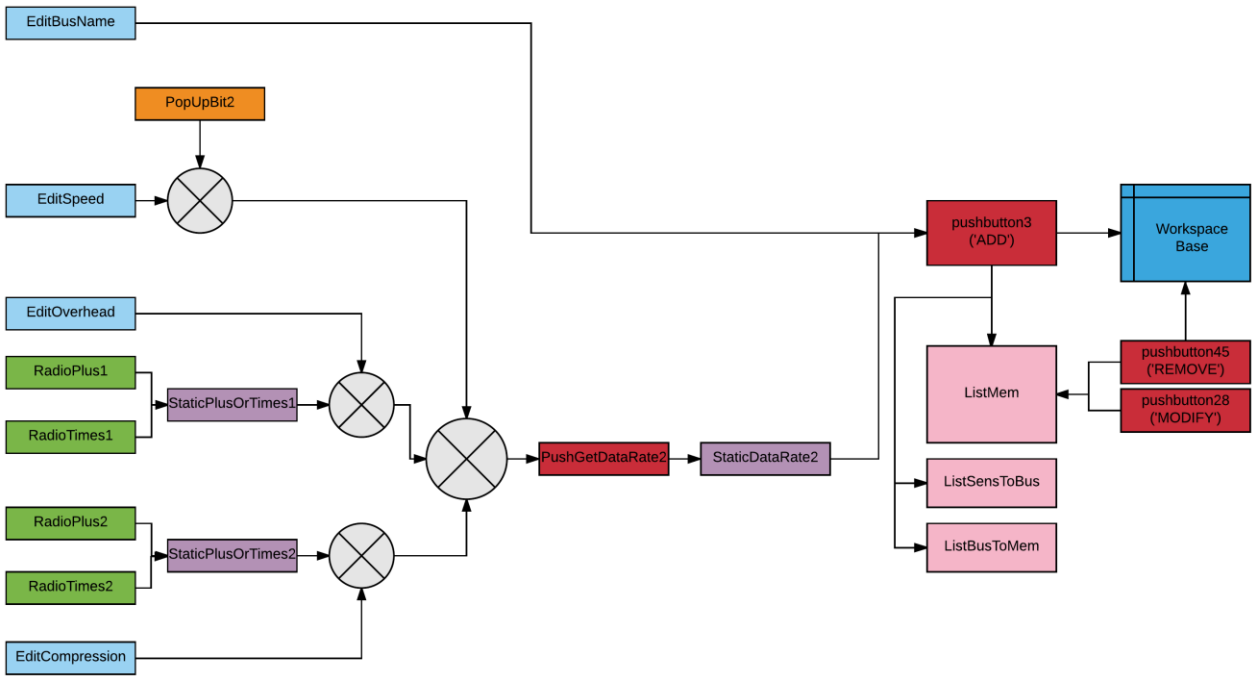

Fig. 2.21 - Il diagramma concettuale della sezione Bus

2.6.1 La compilazione di *Bus*

Come visto per i sensori, si può iniziare la configurazione inserendo il nome del bus all’interno della casella di *EditBusName*, dalla quale il codice estrapola la stringa da assegnare all’apposito campo nel workspace.

Per calcolare il *Data Rate* effettivo fornito dal bus, invece, l’interfaccia offre diversi strumenti. L’utente può cominciare compilando *EditSpeed*, la quale deve contenere il *Data Rate* disponibile, a meno di eventuali overhead o compressioni, che verranno valutati nel seguito. È possibile inoltre

servirsi della lista *PopUpBit2* per selezionare l'unità di misura di interesse (indicata in bps, kbps, ecc.).

2.6.1.1 Il criterio di implementazione di overhead e compressione

Per quanto riguarda la definizione di overhead e compressione (che di default valgono 0, come visibile in Fig. 2.20) è anzitutto necessario stabilire se essi vogliono essere valutati in modo additivo oppure con un fattore moltiplicativo. Nel primo caso, essi valgono come bit aggiunti (nel caso di overhead) o sottratti (tramite la compressione) al *Data Rate* greggio riportato in *EditSpeed*, tali che

$$DataRate_{effettivo} = DataRate_{greggio} + Overhead - Compressione$$

Altrimenti, questi possono essere visti come un fattore moltiplicativo, come spesso accade, così che si possa ricavare

$$\begin{cases} DataRate_{effettivo} = DataRate_{greggio} * Overhead * Compressione \\ Overhead > 1, Compressione < 1 \end{cases}$$

Ovviamente, è possibile una combinazione dei due casi, tale per cui si abbia, ad esempio, un overhead additivo ed una compressione moltiplicativa.

Si noti che, per come è stato scritto il codice, overhead e compressione andranno ad aggiungere bit (nel caso di overhead) o a sottrarne (nel caso di compressione) al *Data Rate* digitato in *EditSpeed*.

È possibile scegliere se valutare overhead e compressione in termini additivi o moltiplicativi attraverso la selezione di uno tra i due Radio Button accoppiati. Sulla GUI ci sono due bottoni dedicati all'overhead e due alla compressione e permettono di selezionare '+' per una valutazione additiva, 'x' per una moltiplicativa.

Una volta compilate le caselle di testo *EditOverhead* e *EditCompression*, il programma dispone di tutte le informazioni per calcolare il *Data Rate* finale fornito dal bus. Premendo su *PushGetDataRate2* il codice svolge le operazioni di conversione in bit (sia di *EditSpeed* che di *EditOverhead* ed *EditCompression*) e prende in considerazione i valori di compressione ed overhead nel computo del *Data Rate* finale, che viene mostrato in *StaticDataRate2*.

2.6.2 Aggiungere il nuovo bus

Analogamente a quanto fatto per la sezione *Sensors*, è possibile servirsi del tasto 'ADD' (*pushbutton3*) per salvare nel *workspace base* il bus configurato ed aggiornare *ListBus*. La pressione del pulsante, dopo il controllo di eventuali voci mancanti, aggiorna la lista *ListSensToBus* di cui si è

parlato nella sezione *Sensors*, e *ListBusToMem* che verrà utilizzata nella configurazione delle memorie. Se non vi sono errori nella compilazione, si dovrebbe ottenere un'interfaccia simile a quella in Fig. 2.22. mentre nel workspace base la struct dei bus viene salvata come in Fig. 2.23. Sono inoltre disponibili i tasti 'REMOVE' e 'MODIFY' che svolgono operazioni analoghe ai corrispondenti pulsanti presenti in *Sensors*.

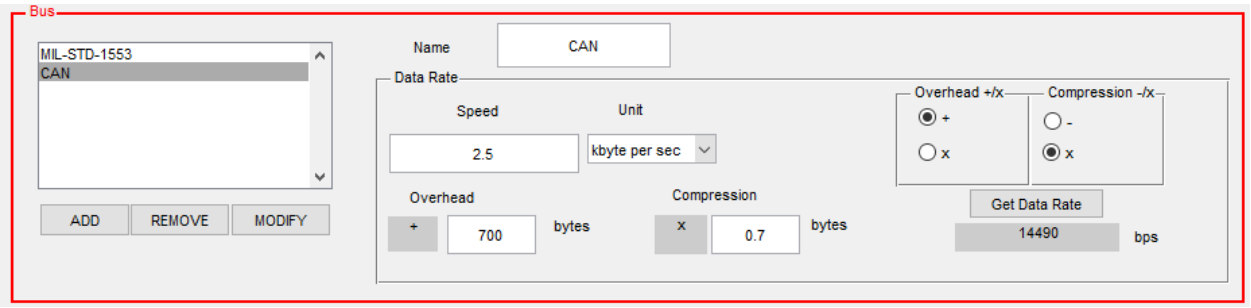

Fig. 2.22 - Un esempio di configurazione nella sezione Bus

```
>> b

b =

1x2 struct array with fields:

    name
    data
    EditSpeed
    PopUpBit2
    StaticPlusOrTimes1
    StaticPlusOrTimes2
    EditOverhead
    EditCompression
```

Fig. 2.23 - La struct b nel workspace base; in questo caso, essa contiene due bus.

2.7 La sezione Memories

La modellizzazione delle memorie è, tra le quattro sezioni di configurazione, la parte che richiede meno dettagli. Infatti, lo scopo della memoria è quello di immagazzinare i dati in arrivo dal bus, in attesa che questi possano essere scaricati verso terra. La memoria è quindi un contenitore ed è caratterizzata in questo programma soltanto dalla capacità, espressa in bit. La sezione *Memories* permette quindi all'utente di aggiungere al modello una memoria assegnandole nome, capacità e l'indicazione su quale tra i bus creati sia collegato a tale memoria.

I parametri che, nel memory budget, caratterizzano una memoria, sono quelli in Tab. 2.6.

Campo	Descrizione
m.name	Nome della memoria. È una stringa di caratteri
m.data	Capacità della memoria, espressa in bit
m.attached_to_upstream	Nome del bus che scarica i dati alla memoria. È una stringa di caratteri

Tab. 2.6 - I campi della struct dedicata alle memorie

Come al solito, l'utente può definire i campi suddetti attraverso una interfaccia grafica (Fig. 2.24).

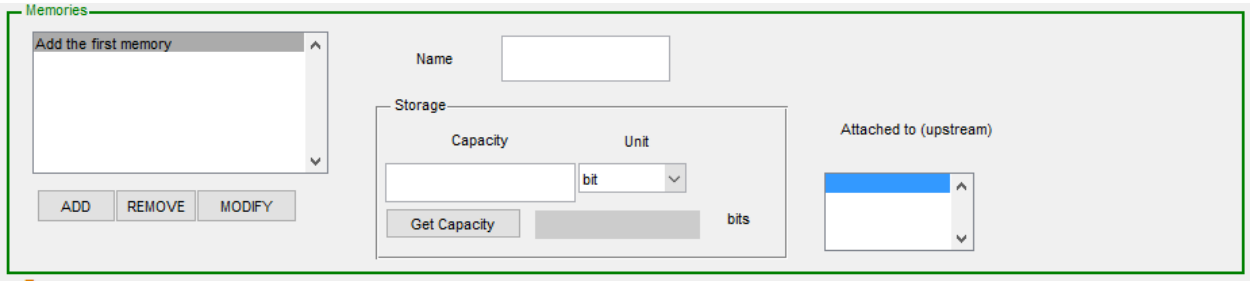

Fig. 2.24 - La sezione Memories, dedicata alla configurazione delle memorie

L'interfaccia, a sua volta, contiene gli elementi grafici riportati in Tab. 2.7.

Elemento della GUI	Nome nel codice
Edit Text riferita a 'Name'	EditMemName
Edit Text riferita a 'Capacity'	EditCapacity
Pop-up List riferita a 'Unit' (unità di misura della capacità)	PopUpBit3
Push Button 'Get Capacity'	PushGetCapacity
Static Text riferita a 'Get Capacity'	StaticCapacity
List Box contenente i bus (riferita ad 'Attached to')	ListBusToMem
Pushbutton 'ADD'	PushAddMem
Pushbutton 'REMOVE'	pushbutton6
Pushbutton 'MODIFY'	pushbutton29
Static Text segnalante un errore (invisibile di default)	StaticMissingError3

Tab. 2.7 - Gli elementi grafici della sezione Memories

Essi interagiscono tra loro per fornire i due campi necessari, secondo lo schema in Fig. 2.25.

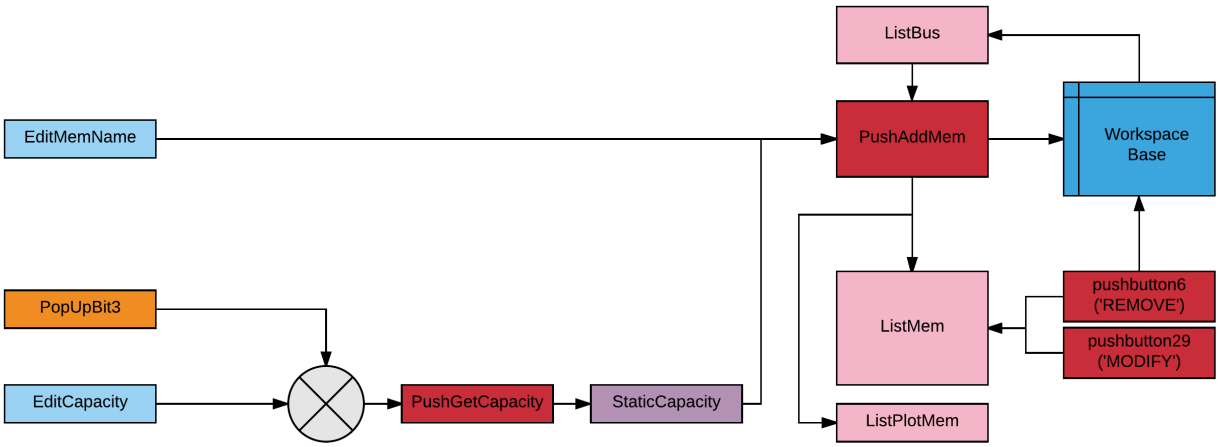

Fig. 2.25 – Il diagramma concettuale della sezione Memories

Il codice per definire il nome della memoria è analogo a quello usato per i nomi nelle altre sezioni (con la differenza che gli *handles* si riferiscono ad *EditMemName* e le funzioni *evalin* ed *assignin* alla struct *m*). Anche per assegnare la capacità si utilizza un algoritmo simile a quello utilizzato per definire i *Data Rate* in *Sensors* e *Bus*: in *EditCapacity* l'utente inserisce il valore della capacità, la cui unità di misura (bit, byte, Gbyte,…) può essere selezionata da *PopUpBit3*. Con la pressione di *PushGetCapacity* si ottiene poi la capacità riscritta in bit che viene riportata in *StaticCapacity*. Fatto ciò, con *PushAddMem* vengono salvati i valori nel *workspace base* e vengono aggiornate le liste contenenti le memorie (*ListMem* e *ListPlotMem*).

Nei campi della struct *m* (Fig. 2.26), per le ipotesi fatte nel paragrafo 2.1.6.2, viene incluso soltanto il bus che trasmette i dati dal sensore alla memoria e non quello che collega la memoria con il rispettivo trasmettitore. Infatti, il modello di questo programma si basa sull'ipotesi che la velocità di trasmissione dei dati (cioè il *Data Rate*) tra una memoria ed il rispettivo trasmettitore sia infinita. Pertanto, non appena i dati si trovano all'interno della memoria, essi sono immediatamente disponibili anche per il relativo trasmettitore.

Ancora una volta, l'indicazione sul bus connesso viene salvata sotto forma di stringa contenente il nome del bus, mentre gli altri campi sono costituiti da numeri *double*.

```
>> m

m =

                    name: 'HD'
                    data: 1.4400e+11
    attached_to_upstream: 'CAN'
            EditCapacity: 18
               PopUpBit3: 9
```

Fig. 2.26 - I campi della struct dedicata alle memorie

2.8 La sezione *Tx*

I trasmettitori sono i terminali dello scambio di dati all'interno del satellite. Infatti, mentre i sensori immettono bit all'interno del sistema, le antenne ed i sottocomponenti di un trasmettitore permettono di trasformare l'informazione da bit ad onda elettromagnetica, la quale viaggia nello spazio e raggiunge le antenne riceventi delle stazioni di terra. Nel momento in cui un'informazione viene trasmessa in downlink, è possibile liberare lo spazio che essa occupava in memoria, permettendo a nuovi dati di essere immagazzinati. Questo processo ciclico sta alla base del memory budget di un satellite, perché stabilisce se sia possibile o meno immagazzinare i dati in arrivo dai sensori, a seconda dell'occupazione della memoria.

Come visto nel paragrafo 1.7.4, in questo bilancio (specialmente per satelliti LEO) gioca un ruolo fondamentale la posizione reciproca tra veicolo e stazione di terra, che porta a definire le fasi in cui i due oggetti si trovano in linea di vista. Per questo motivo è fondamentale che, nel configurare un trasmettitore, l'utente inserisca le stazioni di terra verso cui esso può trasmettere. È inoltre opportuno specificare se il trasmettitore in oggetto ha la possibilità di effettuare il downlink anche di notte. Quest'ultimo aspetto dipende dalla logistica e dalla tecnologia della stazione di terra di riferimento: durante il passaggio del satellite sulla stazione, è spesso necessaria la presenza del personale specializzato, il quale potrebbe essere assente durante i passaggi notturni. Talvolta è possibile che il passaggio del satellite venga assistito dai sistemi automatici della stazione; in questo caso, il downlink può essere effettuato ogni volta che il satellite si trova in vista, a prescindere dall'ora locale.

Nell'effettuare il downlink, il trasmettitore può introdurre un overhead od una compressione, che vanno ad influenzare il *Data Rate* di trasmissione. Perciò, anche questi rientrano nelle specifiche tecniche che l'utente deve fornire al programma.

Alla luce di quanto detto, i campi da salvare nella struct MATLAB dedicata ai trasmettitori (detta *t*) sono quelli in Tab. 2.8. L'utente può inserirli attraverso l'interfaccia grafica (Fig. 2.27), che contiene gli elementi presentati in Tab. 2.9.

Campo	Descrizione
t.name	Nome del trasmettitore. È una stringa di caratteri
t.data	Data Rate di trasmissione, espresso in bps
t.attached_to	Nome della memoria da cui il trasmettitore preleva i dati. È una stringa di caratteri
t.night	Un logico (1 o 0) che segnala rispettivamente la possibilità o l'impossibilità di effettuare il downlink di notte
t.GS	Nome della/delle funzione/i *gscontact* di GMAT, contenute nel report file, le cui stazioni sono quelle che ricevono il downlink dal trasmettitore. È una stringa di caratteri

Tab. 2.8 - I campi della struct dedicata ai trasmettitori

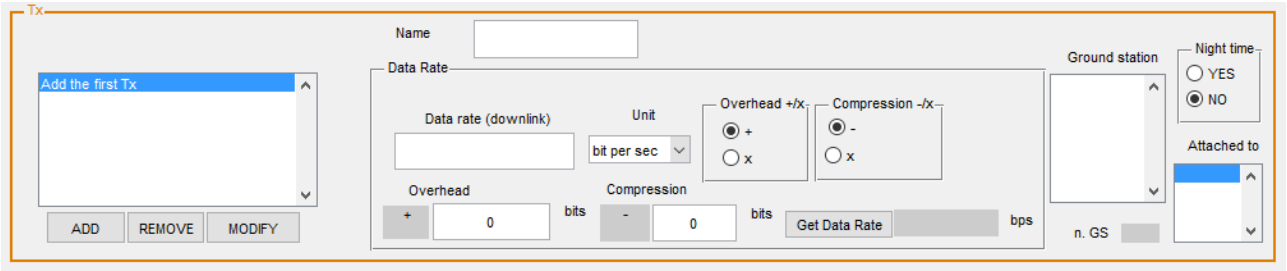

Fig. 2.27 - La sezione Tx, dedicata alla configurazione dei trasmettitori

Elemento della GUI	Nome nel codice
Edit Text riferita a 'Name'	EditTxName
Edit Text riferita a 'Data Rate (downlink)'	EditDataRateDownlink
Pop-up List riferita a 'Unit' (unità di misura del *Data Rate*)	PopUpBit4
Radio Button in alto, in 'Overhead +/x'	RadioPlus3
Radio Button in basso, in 'Overhead +/x'	RadioTimes3
Radio Button in alto, in 'Compression -/x'	RadioPlus4
Radio Button in basso, in 'Compression –/x'	RadioTimes4
Static Text riferita a 'Overhead', sulla sinistra	StaticPlusOrTimes3
Static Text riferita a 'Overhead', sulla destra	StaticBitOrByte3
Static Text riferita a 'Compression' sulla sinistra	StaticPlusOrTimes4
Static Text riferita a 'Compression', sulla destra	StaticBitOrByte4
Edit Text riferita a 'Overhead'	EditOverhead2
Edit Text riferita a 'Compression'	EditCompression2
Push Button 'Get Data Rate'	PushGetDataRate3
Static Text riferito a 'Get Data Rate'	StaticDataRate3

List Box contenente i trasmettitori	ListTx
Pushbutton ‘ADD’	PushAddTx
Pushbutton ‘REMOVE’	PushRemoveTx
Pushbutton ‘MODIFY’	Pushbutton30
Static Text segnalante un errore (invisibile di default)	StaticMissingError4
List Box contenente i campi del report file (riferita a ‘Ground Station’)	ListGS
Radio Button in alto, in ‘Night time’	RadioNightYES
Radio Button in basso, in ‘Night time’	RadioNightNO
List Box contenente le memorie, riferita ad ‘Attached to’	ListMemToTx
Static Text riferita a ‘n. GS’	StaticnGS

Tab. 2.9 - Gli elementi grafici della sezione Tx

Il vasto numero di caratteristiche e combinazioni possibili nella configurazione di un trasmettitore rende l’algoritmo piuttosto complesso, come mostra il diagramma in Fig. 2.28. Il codice di questa sezione è in parte simile a quello descritto nella sezione *Bus*, specialmente nel calcolo del *Data Rate* che include overhead e compressione. Per ulteriori dettagli, si può fare riferimento al paragrafo 2.6.

L’utente può selezionare le stazioni di terra con cui il trasmettitore si collega nel seguente modo: nel momento in cui vengono caricati, attraverso *General Simulation Setup*, i dati orbitali all’interno del programma, uno specifico comando all’interno del Push Button ‘IMPORT’ inserisce nella lista *ListGS* i nomi dei campi di *GMATout* (si veda la Fig. 2.29).

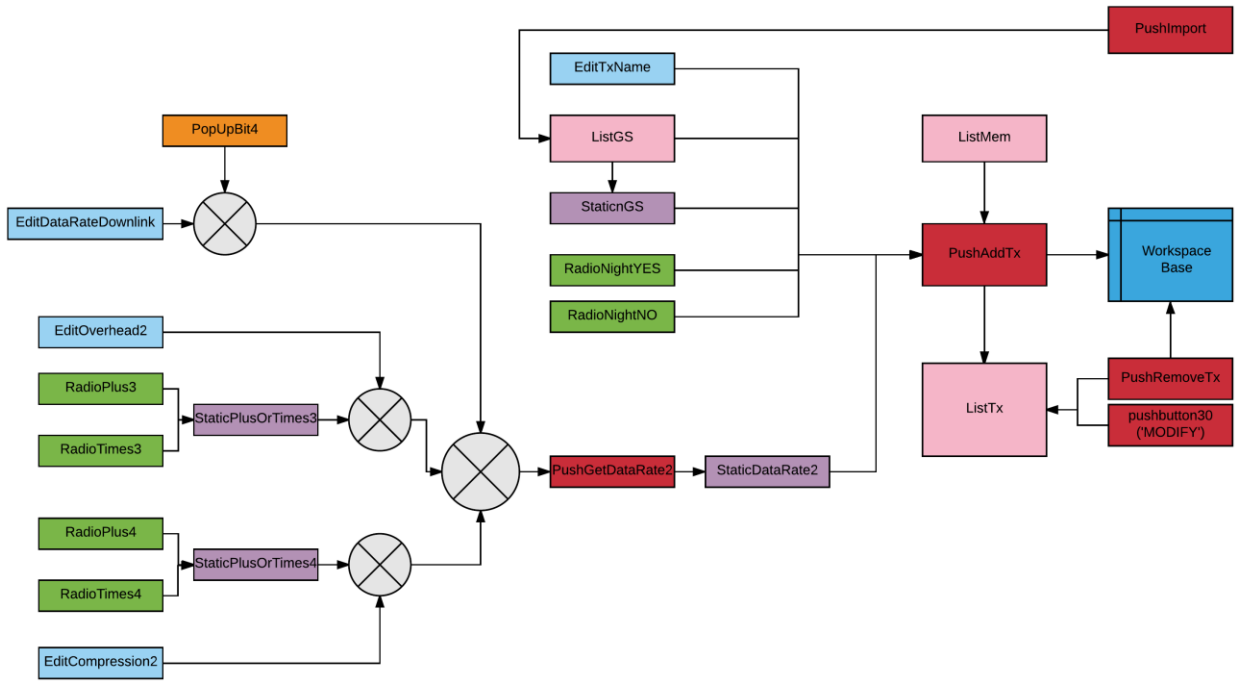

Fig. 2.28 - Diagramma concettuale della sezione Tx

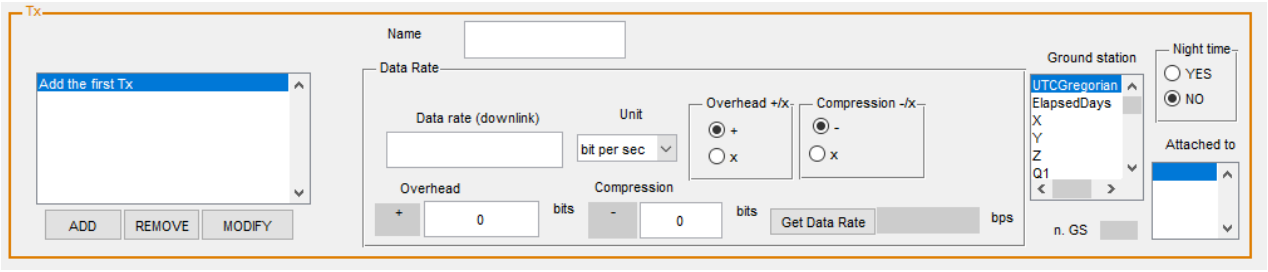

Fig. 2.29 - Dopo aver caricato GMATout, viene in automatico aggiornata la List Box sotto la voce 'Ground station'

Questo viene fatto perché, nella struct contenente i dati orbitali, sono incluse le variabili *gscontact*, le quali stabiliscono in quali istanti il satellite si trova in vista sulle stazioni di terra (si veda il Par. 1.7.4). Selezionando quindi dalla lista *ListGS* le funzioni *gscontact* corrispondenti alle stazioni con cui si vuole stabilire il contatto, sarà poi possibile individuare gli intervalli in cui avviene il downlink. Se l'utente seleziona dalla lista una voce diversa, il programma segnala un errore in fase di calcolo. È possibile selezionare una o più stazioni di terra: tutte quelle selezionate verranno tenute in considerazione nel bilancio delle memorie. Terminata questa operazione, nella casella di testo *ListnGS* viene salvato il numero di stazioni di terra selezionate, che viene poi memorizzato nel campo *t.num_GS_selected*. Questo parametro verrà utilizzato dall'algoritmo nell'esecuzione del programma *runtime*. Per finire l'utente deve scegliere, servendosi dei Radio Button appositi (*RadioNight*), la disponibilità del downlink notturno e la memoria da cui il trasmettitore preleva i dati (attraverso la lista *ListMemToTx*).

Prima di salvare la configurazione del componente, il codice passa in rassegna le caselle di testo per cercare eventuali campi mancanti (e nel caso in cui ci fossero, segnalarli con *StaticMissingError4*); dopodiché aggiunge il nuovo elemento alla struct *t*.

In Fig. 2.30 e Fig. 2.31 sono riportati esempi di due configurazioni complete e salvate nel *workspace base*: la prima è mostrata nella GUI, la seconda nella *command window*:

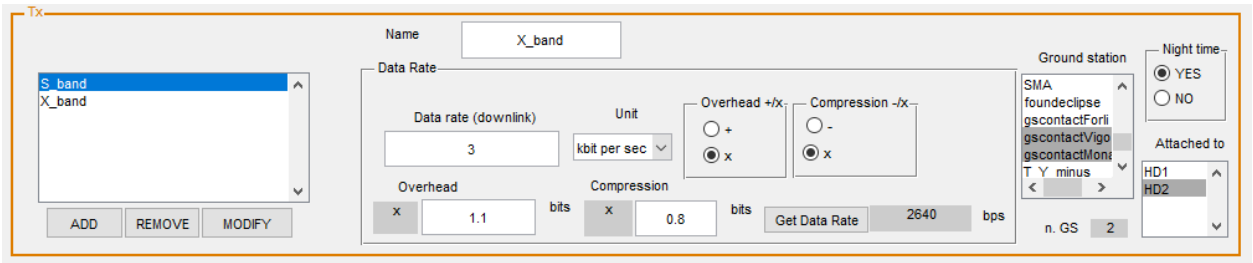

Fig. 2.30 - Un esempio di configurazione di Tx

```
>> t(2)

ans =

                    name: 'X_band'
                    data: 2640
             attached_to: 'HD1'
                   night: 1
                      GS: [2x15 char]
       index_GS_selected: [16 17]
         num_GS_selected: 2
    EditDataRateDownlink: 3
              PopUpBit4: 2
      StaticPlusOrTimes3: 'x'
      StaticPlusOrTimes4: 'x'
           EditOverhead2: 1.1000
        EditCompression2: 0.8000
```

Fig. 2.31 - I campi della struct dedicata ai trasmettitori; qui viene visualizzata solo la seconda componente

2.9 Il codice *Runtime*

Nei paragrafi precedenti è stata descritta la parte di codice che permette all'utente di configurare le varie sezioni del proprio modello, attraverso gli strumenti forniti dalla GUI. Le sezioni *General Simulation Setup*, *Sensors*, *Bus*, *Memories* e *Tx* hanno quindi il compito di raccogliere le condizioni al contorno per la simulazione, attraverso una interfaccia grafica che permette all'utente un approccio rapido ed intuitivo. Tuttavia, come visto nello schema di Fig. 2.2, è la funzione *runtime* a calcolare il memory budget ed a fornire le uscite richieste. Il codice che verrà approfondito in questa sezione rappresenta il centro di elaborazione vero e proprio del software (si veda in particolare il paragrafo 2.9.3).

2.9.1 La sezione *Select Plots*

Al termine della simulazione, il programma fornisce due diagrammi: il primo riferito all'occupazione dei buffer, il secondo all'occupazione delle memorie. Per non appesantire l'aspetto grafico, è possibile per l'utente selezionare i plot che interessa visualizzare. La sezione *Select Plots*, contenente le due liste *ListPlotSens* e *ListPlotMem*, permette una selezione multipla degli elementi da rappresentare sui diagrammi. Queste due liste contengono gli stessi elementi presenti, rispettivamente, in *ListSens* e *ListMem*, dato che vengono aggiornate in concomitanza con le ultime. Nella Fig. 2.32 è riportato un esempio di interfaccia grafica completata correttamente, al termine della configurazione. Si può vedere, sulla destra dell'immagine, la sezione *Select plots*, nella quale l'utente ha selezionato più elementi da plottare.

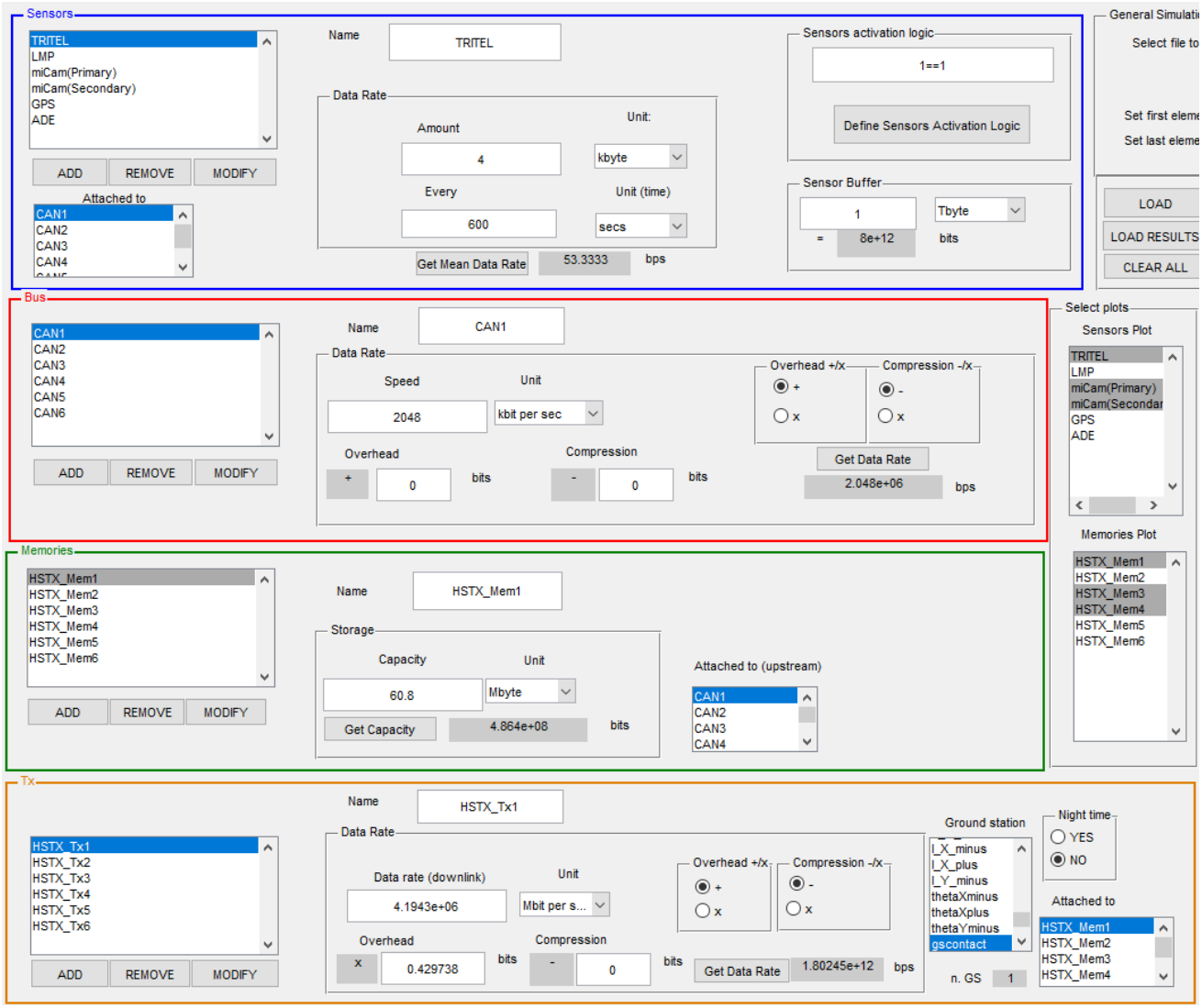

Fig. 2.32 - Un esempio di configurazione completa; in figura è visibile solo la parte sinistra della GUI

Arrivati a questo punto, il software dispone di tutte le informazioni necessarie ad avviare la simulazione.

2.9.2 Il pulsante *PushRUN*, inizializzazione della funzione *runtime*

Si è già accennato al fatto che la funzione *runtime* è programmata in modo tale da poter operare indipendentemente dall'interfaccia grafica. Ciò significa che, nel caso in cui in un generico script MATLAB fossero salvate sia le struct (*s*, *b*, m, *t*) contenenti i parametri della configurazione, sia la struct dei dati orbitali richiesti, sarebbe possibile svolgere il calcolo del memory budget richiamando la funzione *runtime* direttamente da tale script. Questo porta ad avere un programma suddiviso in due

parti principali: l'interfaccia grafica ed il programma per il calcolo, ciascuna con il proprio codice isolato ed indipendente dall'altro.

Nonostante i due moduli siano concettualmente indipendenti tra loro, all'interno del software i dati della configurazione vengono trasferiti dalla GUI al programma di calcolo. Infatti, la configurazione del satellite viene salvata nelle apposite *struct* del *workspace base*, le quali vengono poi recuperate dall' interfaccia grafica nel momento in cui si vuole avviare la simulazione. Questo avviene grazie al *callback* del pulsante 'RUN' (chiamato *PushRUN*), che viene cliccato dall'utente per avviare la simulazione.

La funzione eseguita dal tasto 'RUN' svolge una serie di operazioni necessarie ad inizializzare lo scenario: rende invisibili eventuali messaggi di errore connessi alla configurazione o relativi alla simulazione precedente (e quindi non più utili per l'utente); legge poi l'istante iniziale e finale della simulazione, scelti dall'utente in *General Simulation Setup* e importa gli altri ingressi per la funzione dedicata al calcolo: *GMATout*, *s*, *b*, *m* e *t*.

Nella successiva riga di codice di questo *callback* viene richiamata la funzione *runtime*, nominata *mem_bud_run*, al cui interno viene svolto il memory budget.

2.9.3 mem_bud_run: la funzione *runtime*

La funzione *runtime* è il nucleo del programma sviluppato. L'intero codice dell'interfaccia grafica e la struttura della GUI sono stati sviluppati per inizializzare il suo algoritmo, al termine del quale vengono fornite le uscite descritte nel paragrafo 2.1.3.

2.9.3.1 Gli ingressi e le uscite di mem_bud_run

Per svolgere il calcolo, *mem_bud_run* necessita in ingresso dei seguenti elementi:

- Le struct dedicate a sensori, bus, memorie e trasmettitori (chiamate, nel codice della funzione, *sensors*, *buses*, *memories*, *txs*) che contengono la configurazione completa del sistema di scambio dati del satellite.
- I parametri orbitali contenuti nel report file di GMAT, salvati precedentemente in una struct; questo ingresso viene qui chiamato *GMATfile*. Nella simulazione, in realtà, vengono richiesti soltanto i campi *ElapsedDays*, *foundeclipse* e *gscontact*. Sarebbe quindi possibile fornire in ingresso alla funzione soltanto questi ultimi tre campi.
- L'intervallo di simulazione, salvato con le componenti iniziale e finale del vettore *GMATfile.ElapsedDays*, che nella funzione vengono definite *t_0* e *t_f*.

- Gli elementi *handles*, descritti nel paragrafo 2.2.2.1, attraverso i quali è possibile manipolare gli elementi grafici della GUI.
- Il vettore contenente gli indici dei diagrammi da plottare, riferiti a sensori e memorie (*PlotSens* e *PlotMem*, rispettivamente)

Questi ingressi vengono importati dal pulsante *PushRUN*, come visto nel precedente paragrafo. Essendo *mem_bud_run* una funzione a se stante dotata quindi di un suo workspace interno, gli elementi non suo codice prendono nomi diversi da quelli della GUI. Tuttavia, quando la funzione viene richiamata nel *callback* del tasto 'RUN', gli argomenti in ingresso ed in uscita dalla funzione devono essere riportati con i nomi che essi assumono all'interno di quel workspace, cioè i nomi utilizzati nella GUI.

Le uscite della funzione sono quelle che caratterizzano il satellite modellizzato in termini di memory budget. Queste sono:

- *SENS_Buffer*: l'andamento nel tempo dei dati contenuti nei buffer. È una matrice di double *n*m*, dove *n* è il numero di sensori nel modello e *m* il numero dei time step del vettore tempo da simulare. Quindi, la generica componente *SENS_Buffer(j,i)* rappresenta l'occupazione del buffer *j*-esimo, all'istante *i*-esimo, espressa in bit.
- *MEM_occupation*: l'andamento nel tempo dell'occupazione delle memorie. Anch'essa è una matrice rettangolare, di dimensioni *l*m*, dove *l* è qui il numero di memorie. Dunque, *MEM_occupation(y,i)* è un double, espresso in bit, che indica l'occupazione della memoria *y*-esima quando si trova all'istante *i*-esimo del vettore *GMATfile.ElapsedDays*.
- *data_rate_mean*: è un vettore contenente, nella componente *n*-esima, il *Data Rate* medio di acquisizione del sensore *n*-esimo. L'unità di misura è il bit.
- *downlink_tot*: è un double (espresso in bit) che indica l'ammontare dei dati totali scaricati in downlink alla fine della missione simulata.
- *gen_tot*: simile al precedente, ma indica la quantità complessiva di dati generati a bordo del satellite.
- *error_flag*: un indice numerico che, al termine della simulazione, viene prelevato dalla GUI e permette di segnalare all'utente eventuali errori avvenuti durante l'esecuzione del *runtime*. Ogni indice corrisponde ad un diverso errore, che viene segnalato a video attraverso una stringa di testo.

Il codice di *mem_bud_run* è stato suddiviso in sezioni, ognuna delle quali svolge un ruolo ben definito.

2.9.3.2 Inizializzazione del calcolo

Innanzi tutto viene inizializzata la variabile *error_flag* al valore 0. Infatti, nel caso in cui non si presentano errori durante il calcolo, questa variabile mantiene il valore 0 e comunica al codice di *PushRUN* che la simulazione è stata svolta correttamente.

Poiché *mem_bud_run* sfrutta al suo interno diversi cicli *for*, un altro importante parametro da definire, prima della simulazione, è il numero di passi (o step) che caratterizzano la durata dei cicli. Essendo la propagazione in GMAT svolta in passi discreti, anche il *report file* da essa fornito avrà un certo numero finito di elementi. Il calcolo del memory budget deve ripercorrere, passo per passo, la propagazione della missione e avrà quindi lo stesso numero di step. Pertanto, si definisce una variabile *time* che contiene il numero di step eseguiti da GMAT.

Data la natura discreta del problema, oltre al numero di step da eseguire nella simulazione, è fondamentale conoscere la distanza che separa un passo e quello successivo. Questa viene definita *dt* e calcolata servendosi della built-in di MATLAB chiamata *diff*, che restituisce la differenza tra l'elemento *i* di un generico vettore e l'elemento *i-1*. Per come è definita la funzione *diff*, essa contiene in uscita un elemento in meno rispetto al vettore in ingresso; perciò, per far combaciare le dimensioni tra la lunghezza di un ciclo (che è *time*) ed il numero di elementi presenti in *dt*, si può aggiungere a quest'ultimo vettore uno zero come primo elemento. Questo accorgimento non compromette (per motivi che verranno illustrati in seguito) la validità dei risultati. Infine, per inizializzare tutti i parametri con le stesse unità di misura (bit per i dati, secondi per il tempo), il campo *GMATfile.ElapsedDays* viene moltiplicato convertito da frazioni di giorno a secondi.

2.9.3.3 Calcolo dei buffer dei sensori

La seconda sezione di *mem_bud_run* determina il valore di *SENS_Buffer* (andamento dei buffer), *gen_tot* e *data_rate_mean*. L'intero codice che svolge questa attività è contenuto dentro ad un ciclo *for*, dato che lo stesso algoritmo di calcolo va ripetuto per tutti i sensori presenti nel modello. Quindi, l'indice del *for* parte da 1 ed incrementa fino a raggiungere il numero di sensori.

Inizialmente l'algoritmo genera il vettore riga *SENS_activation* che indica, in ogni istante, se il sensore si trova attivato o disattivato. Questo vettore è composto da variabili booleane (numeri 0 oppure 1) tali che l'elemento *SENS_activation(j,t)* vale 0 se il *j*-esimo sensore all'istante *t*-esimo è disattivato; vale 1 se in tale circostanza il sensore è attivo. Per definire questo oggetto, è stata sfruttata la funzione *eval* riferita alla stringa contenuta in *sensors(j).activation_condition* (si veda Par. 2.5.1.1). Ad esempio, se la stringa afferma che '*ElapsedDays* è maggiore di 3', il sensore si attiva soltanto trascorsi i primi tre giorni di missione, prima dei quali *SENS_activation* vale sempre 0.

La funzione dispone così delle informazioni necessarie a calcolare l'occupazione dei buffer. Il passaggio successivo è quindi il ciclo *for* all'interno del quale viene svolto il calcolo dei vettori di *SENS_Buffer*, dall'istante iniziale all'istante finale della simulazione.
Per calcolare l'occupazione dei buffer è necessario definire, passo per passo, la quantità di dati acquisita dal sensore e la quantità drenata attraverso i bus, chiamate rispettivamente *SENS_DataVolume* e *BUS_DataVolume*. La prima voce è definita come:

$$SENS_{DataVolume}(j,i) = sensors(j).data * dt(i) * SENS\_activation(j,i)$$

Quindi, il *Data Volume* del sensore *j*-esimo, all'istante *i*-esimo della simulazione, è uguale al prodotto tra il proprio *Data Rate* medio, il passo temporale *i*-esimo (espresso in secondi) e la condizione di attivazione del sensore, al dato istante. Quando il sensore è disattivato, i primi due fattori vengono moltiplicati per *SENS_activation* che vale 0, quindi il *Data Volume* vale 0.

Il *BUS_DataVolume* viene invece definito attraverso con l'espressione:

$$BUS_{DataVolume}(j,i) = min\begin{cases} SENS_{Buffer}(j,i-1) \\ buses(bus_{index}).data * dt(i) * (SENS_{Buffer}(j,i-1) > 0)) \end{cases}$$

Secondo questa equazione il *DataVolume* del bus *j*-esimo, all'istante *i*-esimo, vale quanto il minimo tra la quantità di dati presenti nel buffer all'istante precedente (*i-1*) e la quantità di dati che il bus è in grado di trasportare per unità di tempo. Definire il *Data Volume* in questo modo è necessario, dato che il trasferimento dei dati attraverso i bus comincia soltanto dopo che questi sono stati acquisiti.
Si noti che *BUS_DataVolume* vale quanto il massimo *Data Volume* fornito dal bus, nel caso in cui quest'ultimo sia minore della quantità di bit presenti nel buffer; vale invece quanto il contenuto del buffer se tale contenuto è minore del massimo *Data Volume* che il bus può erogare. Questa condizione serve ad evitare che il modello sottragga dal buffer più bit di quanti ve ne siano effettivamente all'interno.
Per finire, si noti che il *Data Volume* del bus è essere diverso da zero soltanto se il contenuto del buffer a monte è anch'esso diverso da zero. Infatti, se la memoria del sensore fosse vuota ed il bus continuasse ad erogare dati, il modello porterebbe ad un valore di occupazione negativo. Per evitare ciò, è stato necessario moltiplicare il secondo argomento della funzione *min* per un logico che vale 0 se la memoria, all'istante precedente, è vuota e vale 1 altrimenti.
Per chiarire il concetto, si vedano i tre differenti scenari proposti in Tab. 2.10:

***SENS_Buffer(j,i-1)* [bits]**	***buses(bus_index).data* [bits]**	***BUS_DataVolume (j,i)*[bits]**
100	45	45
45	100	45
0	100	0

Tab. 2.10 - Tre differenti scenari ed il rispettivo Data Volume del bus

Nel primo caso, il buffer contiene 100 bit, ma la quantità massima di dati che il bus può scaricare, nell'intervallo di tempo *dt*, è 45 bit. Pertanto, il *Data Volume* fornito dal bus vale 45 bit.
Nella riga successiva, invece, ci sono 45 bit residui nel buffer, il quale è collegato ad un bus con una velocità massima di 100 bit scaricabili in un tempo *dt*. Ciò significa che il bus è in grado di liberare l'intera memoria in un solo step. La mole di dati scaricati, e quindi *BUS_DataVolume*, vale in questo caso 45. L'ultimo caso prevede il buffer del sensore vuoto ed una capacità in download di 100 bit. Il codice assegna il valore 0 a *Bus_DataVolume*, così che il bus risulti inattivo.
Tutte le considerazioni appena fatte, implementate nell'algoritmo, portano ad avere nel vettore *BUS_DataVolume(j)* l'effettiva mole di dati che, ad ogni passo della simulazione, viene drenata dal bus *j*-esimo.

Una volta definiti i *Data Volume* di sensori e bus, è possibile calcolare l'evoluzione nel tempo dell'occupazione dei buffer. Concettualmente, il calcolo di *SENS_Buffer* è semplice, come mostra la Fig. 2.33: i dati nel buffer aumentano a seconda dell'entità del *Data Volume* in arrivo dal sensore (ad esempio, una videocamera) e diminuiscono tanto più è alto il *Data Volume* drenato dal bus.

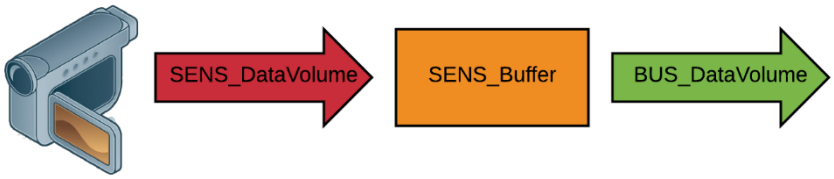

Fig. 2.33 - Lo schema concettuale del flusso di dati attraverso il buffer

Questo bilancio di dati è stato tradotto, nel codice, con l'espressione:

$$SENS_{Buffer}(j,i) = \max\{0, SENS_{Buffer}(j,i-1) + SENS_{DataVolume}(j,i) - BUS_{DataVolume}(j,i)/num\_active\_sens\}$$

Si può quindi dire che l'occupazione del buffer *j*-esimo, all'istante *i*-esimo, coincide con l'occupazione del buffer all'istante *i-1* più la differenza tra il *Data Volume* del sensore e quello del bus all'istante *i*. Si noti che il *Data Volume* scaricato dal bus non è quello nominale, ma viene diviso in tante parti quanti sono i sensori collegati a tale bus, in modo che ogni sensore abbia a disposizione la stessa banda degli altri.
Così come è stato fatto per la definizione di *BUS_DataVolume*, anche in questo caso sono necessarie delle accortezze nel codice per far sì che l'occupazione non assuma valori negativi: è stata utilizzata la funzione MATLAB *max*, che calcola, istante per istante, il massimo tra i due argomenti. Dato che il primo argomento è zero, se il secondo è un numero negativo il risultato vale zero.
All'interno dello stesso ciclo viene anche calcolato il valore di *gen_tot*. Esso è definito come

$$gen_{tot}(j,i) = gen_{tot}(j,i-1) + SENS\_DataVolume(j,i)$$

ed esprime la quantità totale di dati acquisiti fino all'istante *i*.
L'ultima operazione che il ciclo svolge è verificare, ad ogni step, se il buffer dei sensori supera la memoria allocata dall'utente in fase di configurazione (cioè la voce *sensors.buffer*). Nel caso in cui il buffer superi tale valore, viene posta la variabile *error_flag* uguale ad 1. Questo permetterà, al termine della funzione *runtime*, di mostrare a video l'errore che i dati hanno portato a saturazione il buffer.
Al termine del ciclo, è necessario fornire all'utente un riferimento grafico dei calcoli eseguiti, attraverso la funzione *plot*. Nell'interfaccia grafica è stato riservato uno spazio per la visualizzazione dei diagrammi forniti dal memory budget: in alto quello dedicato all'andamento della memoria nei buffer (*axes1*), in basso le memorie (*axes3*[24]). Entrambi i plot hanno sulle ascisse il vettore *ElapsedDays*, cioè i giorni trascorsi dall'inizio della missione; sulle ordinate ci sono i bit occupati.

Prima di passare all'occupazione delle memorie, viene definito un ultimo vettore, *data_rate_mean*, che contiene il *Data Rate* medio con cui i sensori hanno svolto le acquisizioni. Questo elemento ha tante componenti quanti sono i sensori nel modello; la componente *j*-esima di questo vettore equivale alla media nel tempo del *Data Rate* del sensore *j*-esimo. A tal proposito, è stato definito:

$$SENS_{DataRate}(j,i) = sensors(j).data * SENS\_activation(j,i)$$

[24] Si veda il Par. 2.2.1.

ovvero

$$SENS_{DataRate}(j,i) = \frac{SENS_{DataVolume}(j,i)}{dt(i)}$$

del quale è possibile fare la media degli elementi.

2.9.3.4 Calcolo dell'occupazione delle memorie

Il processo di calcolo per le memorie è concettualmente simile a quello già visto per i buffer dei sensori, ma in questo caso va aggiunta una importante considerazione: mentre i bus in uscita dai buffer scaricano dati ogni volta che l'occupazione del buffer è maggiore di zero, le memorie possono essere svuotate solo nel momento in cui il satellite si trova in vista sulla stazione di terra, ed è quindi possibile attivare il trasmettitore per effettuare il downlink. Per questo motivo, a seconda dell'orbita e della missione, è importante dimensionare correttamente la capacità delle memorie, così che queste possano trattenere le informazioni raccolte fino a quando non si presenta la finestra di downlink (che in LEO può arrivare anche dopo diverse ore). In questa sezione della funzione *mem_bud_run*, quindi, sono state innanzi tutto implementate una serie di operazioni che scandiscono gli istanti in cui è possibile scaricare i dati verso terra.

Così come nel codice dedicato ai buffer, l'intero algoritmo è contenuto in un ciclo *for* che passa in rassegna tutte le memorie (fino all'indice *num_mem*).

Per prima cosa, vengono definite due variabili booleane: *TX_activation_night* e *gs_flag_vect*. La prima tiene conto della possibilità (o impossibilità) che il satellite effettui il downlink di notte; la seconda segnala invece gli istanti temporali in cui avviene il contatto con le ground station.

gs_flag_vect è un vettore di lunghezza *time*; le sue componenti valgono 1 se, all'istante *i*-esimo, ci sono stazioni di terra in vista, valgono 0 altrimenti. In questo modo, esso definisce gli istanti in cui i dati vengono scaricati. A *gs_flag_vect* è stato assegnato il vettore *gscontact* (contenuto nel report file). Dato che nel *report file* ci possono essere più vettori *gscontact*, dedicati ognuno ad una diversa stazione di terra, è necessario che il programma recuperi la stringa della stazione che l'utente ha selezionato durante la configurazione (che è stata salvata in *t.GS*).

Nel caso in cui il trasmettitore in oggetto scarichi i dati a più di una stazione di terra, i *gs_flag_vect* riferiti ad ognuna di queste stazioni vengono uniti tra loro (con la funzione *or* di MATLAB) per formare un nuovo vettore (si veda l'esempio in Fig. 2.34 con tre stazioni di terra).

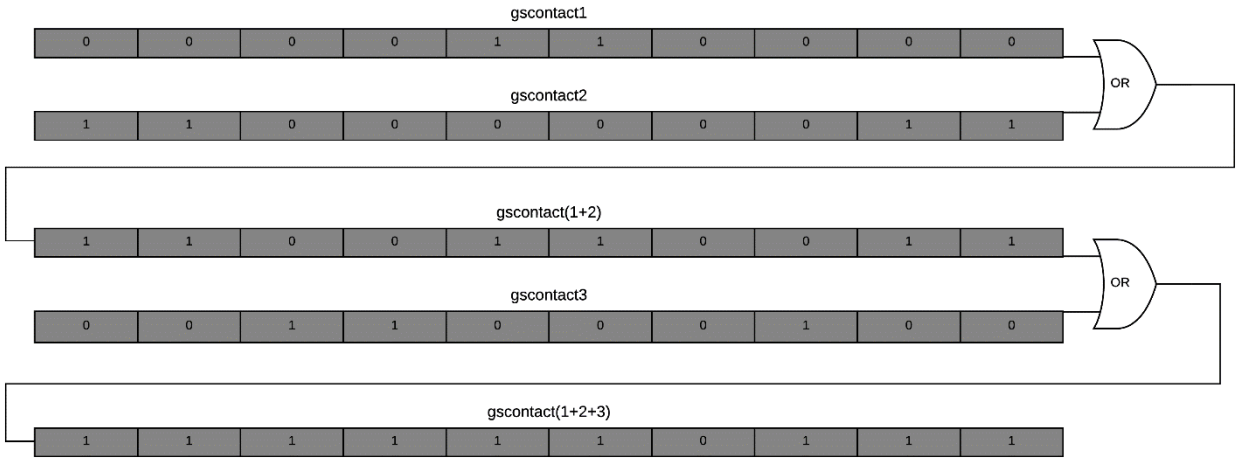

Fig. 2.34 - Esempio di funzionamento del codice per la determinazione di gs_flag_vect con stazioni di terra multiple

Lo scaricamento dei dati può avvenire solo se *gs_flag_vect* vale 1.

Una volta definiti gli intervalli in cui il satellite può effettuare il downlink, può essere avviata la simulazione vera e propria, questa volta contenuta in un ciclo *for* che parte da *t_0+1* ed arriva a *t_f*. Nelle prime righe viene definito il parametro *TX_activation_night*, sulla base della variabile *can_night (*che vale 1 se l'utente ha stabilito che il trasmettitore funziona di notte, 0 altrimenti*)* e del vettore *is_eclipse (*che coincide *con* il campo *foundeclipse* di *GMATfile)*. I valori di queste due variabili vengono valutati contemporaneamente da una serie di cicli *if*, tali che, all'istante *i*-esimo:

- Se il trasmettitore può operare di notte ed è notte, *TX_activation_night* vale 1;
- Se invece il trasmettitore può operare di notte ed è giorno, *TX_activation_night* vale 1;
- Se invece il trasmettitore non può operare di notte ed è notte, *TX_activation_night* vale 0;
- Infine, se il trasmettitore non può operare di notte ed è giorno, *TX_activation_night* vale 1.

Quando *TX_activation_night* vale 0, le condizioni non permettono di effettuare il downlink.
È da notare che, per come è stato scritto il codice, la condizione di eclissi del satellite coincide con la notte nel punto sulla Terra sotto il satellite. Per satelliti in orbita bassa, l'approssimazione può essere mantenuta senza commettere grandi errori.
Alla luce delle considerazioni appena fatte, è evidente che il downlink del satellite avviene soltanto se sia *TX_activatrion_night* sia *gs_flag_vect* sono uguali a 1.
Per stimare l'occupazione della memoria *y*-esima, quindi, si segue la logica illustrata nello schema di Fig. 2.35.

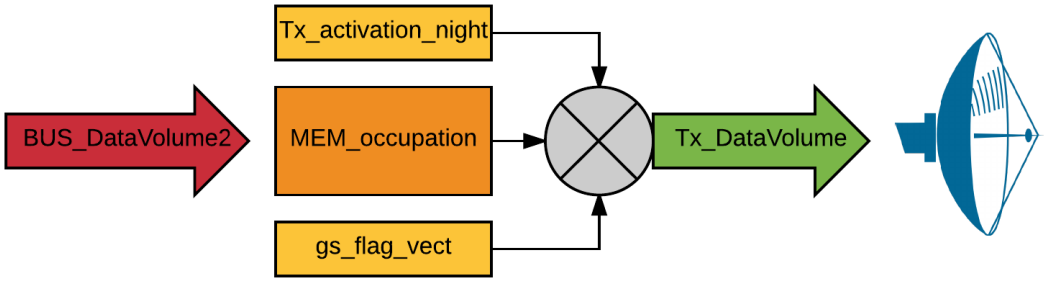

Fig. 2.35 - Lo schema concettuale del flusso di dati attraverso la memoria

Al generico istante *i*-esimo, i dati in ingresso sono quelli di *BUS_DataVolume2*, mentre quelli in uscita sono di *TX_DataVolume.BUS_DataVolume2* è la somma dei *Data Rate* di tutti i sensori collegati al bus in oggetto. Per ogni memoria, quindi, viene definito un vettore *sens_index* che contiene gli indici di tutti sensori che scaricano dati verso tale memoria. Per sommare i *Data Volume* di ogni sensore, innanzi tutto si inserisce in un ciclo *for*, la relazione:

$$BUS_{DataVolume}(r,i) = min\begin{cases} SENS_{Buffer(sens\_index(r),i-1)} \\ buses(bus_{index}).data * dt(i) * (SENS_{Buffer(y,i-1)} > 0) \end{cases}$$

in cui *r* è un indice che spazia tra 1 ed il numero di sensori che scaricano verso la memoria *y*. Quindi, per ogni *r*, il *Data Volume* calcolato corrisponde a quello scaricato dal sensore *r*-esimo. A questo punto, il *Data Volume* complessivo che entra nella memoria è la somma di quelli calcolati nel ciclo. Quindi:

$$BUS_{DataVolume2}(y) = \sum_{k=1}^{r} BUS_{DataVolume}(k)$$

Il vettore *TX_DataVolume* è invece la quantità di dati trasmessi, nell'intervallo *dt*, da parte del trasmettitore *y*-esimo. Pertanto si definisce come:

$$TX_{DataVolume}(y,i) = txs(y).data * dt(i)$$

dove *txs(y).data* è il campo contenente il *Data Rate* stabilito dall'utente per il trasmettitore *y*-esimo.

Con questi ultimi due elementi è possibile stimare l'occupazione della memoria, istante per istante. Essa vale:

$$MEM_{occupation}(y,i) = \max \begin{cases} 0 \\ MEM_{occupation(y,i-1)} + BUS_{DataVolume}(y,i) + \\ - TX_{DataVolume}(y,i) * TX_{activation_{night(1,i)}} * gs\_flag\_vect(txs(y).num_{GS_{selected}}, i) \end{cases}$$

Anche in questo caso, così come per *SENS_Buffer*, la funzione *max* garantisce che l'occupazione della memoria non scenda mai sotto lo zero, che è il valore minimo raggiungibile, a prescindere dal *Data Rate* fornito dal trasmettitore. Ancora una volta, l'occupazione della memoria all'istante *i* vale quanto l'occupazione all'istante *i-1*, più il *Data Volume* in ingresso dal bus, meno l'eventuale *Data Volume* drenato dal trasmettitore. Quest'ultimo termine viene però moltiplicato sia per *TX_activation_night* che per *gs_flag_vect*, entrambe variabili booleane.

Dopo aver quantificato l'occupazione della memoria, il programma controlla che questo valore non superi la capacità massima predefinita dall'utente (cioè il valore salvato nel campo *m.data*). Se questo si verifica, viene assegnato il valore 2 ad *error_flag*. Questo valore viene in seguito segnalato all'interfaccia grafica che, attraverso una stringa di testo, comunica all'utente il superamento della memoria massima consentita.

Nel momento in cui *i* coincide con *t_f*, si interrompe il ciclo *for*. Il programma è ora in grado di calcolare anche la quantità totale di dati scaricati. Per quantificare questa grandezza, che viene salvata nell'uscita *downlink_tot*, l'algoritmo somma la variazione di *MEM_occupation* negli istanti in cui la memoria decrementa con il *Data Volume* del bus quando l'occupazione è costante o aumenta e si sta effettuando il downlink.

Nelle ultime righe, il codice plotta l'occupazione delle memorie.

2.9.4 Il pulsante *PushRUN*, salvataggio dei risultati e controllo errori

Con il plot delle memorie si conclude il codice *runtime* del programma. Dopo aver richiamato la funzione *mem_bud_run*¸l'algoritmo di *PushRUN* esegue una serie di operazioni che sfruttano i risultati ottenuti.

Innanzi tutto, le variabili in uscita dalla simulazione (*SENS_Buffer*, *MEM_occupation*, *error_flag*, *downlink_tot*, *gen_tot*) vengono salvate nel workspace base con il comando *assignin*, in modo tale che siano tutte accessibili dall'utente. Vengono poi create, per ogni sensore, le variabili *gen_sens_max*, a cui viene assegnata la componente più alta del vettore *gen_tot*. Queste ultime vengono a loro volta sommate, per calcolare il valore totale di dati generati, salvati nella variabile *gen_tot_max*. Analogamente viene calcolato anche *downlink_tot_max*.
Questi ultimi due valori vengono mostrati a video, riportati nelle due caselle di testo statiche chiamate *StaticTotGen* e *StaticTotDown*, con le giuste unità di misura. I numeri nelle due caselle di testo sono quindi a loro volta scalati in base all'unità di misura corrispondente (si veda la Fig. 2.36).

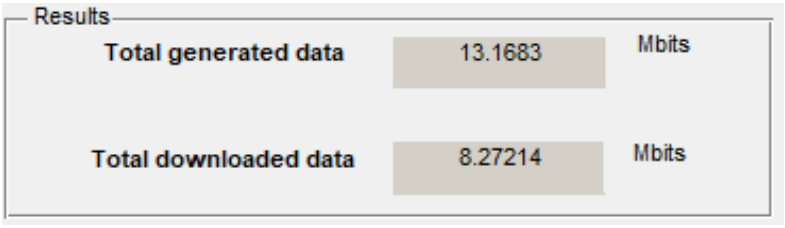

Fig. 2.36 - Un esempio della sezione Results, al termine della simulazione

Per finire, questo callback recupera la variabile *error_flag* fornita da *mem_bud_run*. Se *error_flag* vale 0, così come era stata inizializzata, significa che il codice *runtime* non ha riscontrato errori e che il memory budget ha dato esito positivo: i dati non hanno superato il limite fisico imposto dalla capacità dei buffer e/o delle memorie. Se *error_flag* a fine simulazione vale 1, significa che uno dei buffer è stato saturato dai bit in ingresso. Questo viene segnalato all'utente tramite la casella di testo *StaticRunError*, che viene resa visibile (settando a 'on' la sua proprietà *Visible*) sull'interfaccia (Fig. 2.37). Se *error_flag* vale 2, significa che il problema della saturazione riguarda una o più memorie. Il codice procede come visto poc'anzi: mostra a video la stringa di testo che segnala l'errore, questa volta con riferimento alla memoria (*A memory capacity exceeds the defined value*).
Con questo si conclude l'algoritmo che il programma svolge durante l'esecuzione lanciata con 'RUN'. Tutti i risultati della simulazione, a questo punto, sono disponibili sotto varie forme: alcuni sotto forma di plot, altri come valore numerico mostrato a video; tutti quanti sono stati inoltre salvati nel workspace base di MATLAB.
Per facilitare la raccolta di eventuali errori che riguardano l'utilizzo della funzione *runtime*, l'intero codice di *PushRUN* è stato racchiuso in un ciclo *try-catch*.

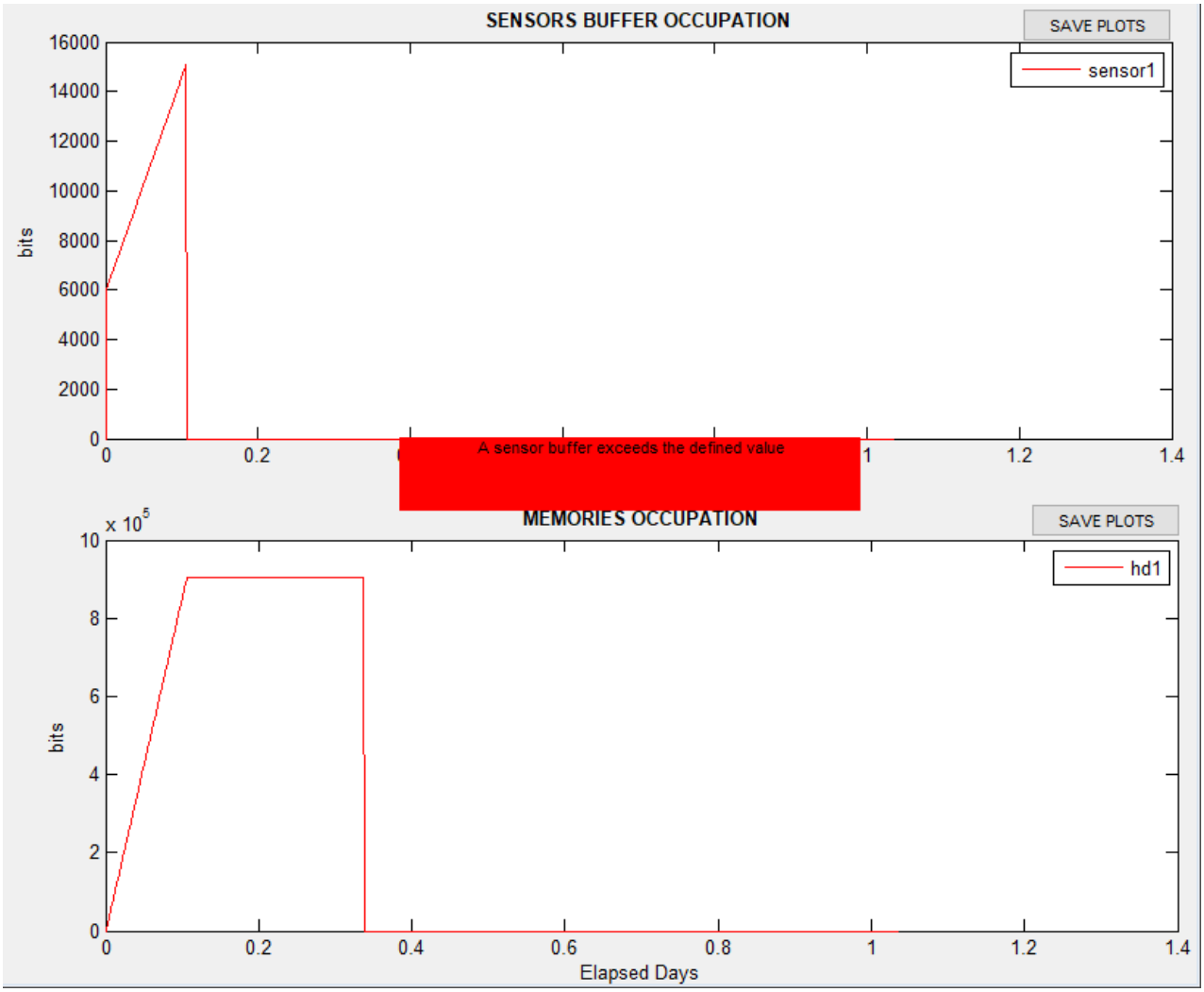

Fig. 2.37 - Un esempio di simulazione in cui l'occupazione del buffer supera la capacità predefinita

2.10 Salvataggio e caricamento dei risultati

Una volta terminata la simulazione, l'utente può salvare i risultati ottenuti e/o la configurazione del modello. Nel programma sono stati implementati gli strumenti adatti a svolgere queste operazioni, ai quali è possibile accedere attraverso i Push Button dedicati.

2.10.1 Il pulsante *PushSAVE*

Attraverso il pulsante nominato 'SAVE' sulla GUI, è possibile effettuare il salvataggio dell'intera configurazione: tutti i campi compilati dall'utente nelle sezioni *Sensors*, *Bus*, *Memories*, e *Tx* vengono salvati in un file in formato *.mat*, che può essere in seguito riutilizzato.

Per questa procedura, il callback valuta dal workspace base (tramite *evalin*) le quattro struct principali, *s*, *b*, *m* e *t*. Dopodiché, sfruttando la corretta sintassi e la funzione *uiputfile*, è possibile definire il percorso file in cui salvare il documento ed il nome da assegnare a quest'ultimo. Infine, la

funzione *save* procede al salvataggio. Si noti che questo pulsante salva soltanto le struct dedicate alla configurazione e non i risultati ottenuti.

2.10.2 Il pulsante *PushSaveResults*

Questo elemento è dedicata al salvataggio della configurazione ma anche dei risultati ottenuti. È possibile, infatti, utilizzare il programma anche per visualizzare nella GUI plot e i risultati precedentemente prodotti.

Per fare ciò, in questo workspace vengono valutate dal workspace base non solo le quattro struct dedicate al setup del satellite, ma anche le uscite della funzione *runtime*: *SENS_Buffer*, *MEM_occupation*, *data_rate_mean*, *downlink_tot*, *gen_tot* ed *error_flag*. Per il resto, la procedura di salvataggio è analoga a quella descritta nel paragrafo 2.10.1.

2.10.3 Il pulsante *PushLOAD*

Come è possibile salvare i dati ed i risultati prodotti nel programma, essi possono anche essere ricaricati. Per ricaricare soltanto la configurazione del satellite (e quindi le quattro struct principali) l'utente può sfruttare questo pulsante.

A differenza dei callback dedicati al salvataggio, il codice per il caricamento dei dati è piuttosto lungo. Questo contiene infatti una lunga serie di funzioni che servono a riempire le caselle di testo ed a compilare le liste di tutta l'interfaccia grafica. Innanzi tutto viene caricato, attraverso *uigetfile* ed *evalin*, il file *.mat* contenente i dati della configurazione. Poi, utilizzando le informazioni del file, la funzione *set* aggiorna gli elementi grafici.

2.10.4 Il pulsante PushLoadResults

I dati ottenuti con il pulsante 'LOAD' forniscono soltanto le informazioni a monte della simulazione. Talvolta, per l'utente può essere comodo visualizzare i plot ed i risultati ricavati dal calcolo. In questo caso, si può utilizzare *PushLoadResults*: questo valuta, con gli stessi metodi descritti in precedenza, non solo le struct (compresa *GMATout*) ma anche le uscite di *mem_bud_run*. Infine, questa funzione compila le liste *ListPlotSens* e *ListPlotMem*, così che per l'utente sia possibile, semplicemente cliccando sulla voce desiderata, plottare i risultati ottenuti in precedenza.

2.10.5 I pulsanti *PushSavePlots*

Per salvare soltanto i plot ricavati, senza ulteriori informazioni sulla simulazione, l'utente può servirsi di questi bottoni, posti sopra a ciascun elemento *axes*. Essi, sfruttando ancora una volta *uiputfile*, creano un file *.fig* che viene salvato nel percorso scelto dall'utente.

2.10.6 Il pulsante *PushClearAll*

Infine, grazie al tasto 'CLEAR ALL' è possibile ripulire sia la GUI che il workspace base da tutti gli elementi presenti. Dopo l'esecuzione del pulsante, l'interfaccia appare come se fosse appena stata avviata, senza elementi nel workspace base.

2.10.7 La funzione *OpeningFcn*

Come già visto, nel momento in cui una GUI in MATLAB viene aperta, viene eseguito anche il suo codice, che è un file *.m*. Oltre alle sezioni di codice relative ai callback degli elementi grafici, sono presenti altre funzioni che ricoprono un certo ruolo nel funzionamento dell'interfaccia grafica. Una di queste è la funzione *OpeningFcn*, che viene eseguita immediatamente prima che la GUI venga mostrata all'utente. Essa è contenuta infatti nelle primissime righe di codice ed è quindi tra le prime operazioni svolte dal *.m*.

Alcune operazioni all'interno della funzione sono scritte di default da GUIDE, pertanto non si entrerà nel merito. Per quanto riguarda il contenuto originale, invece, è stato implementato l'aggiornamento automatico dell'interfaccia grafica partendo delle variabili presenti nel workspace base: se questo contiene, all'apertura della GUI, delle variabili pertinenti al programma, queste ultime vengono passate automaticamente agli elementi grafici. In questo modo, i campi relativi alle variabili presenti nel workspace vengono visualizzati dall'utente nell'interfaccia grafica.

3. CAPITOLO 3
VALIDAZIONE DEL SOFTWARE

Al termine dello sviluppo del software e di tutte le sue funzionalità, è necessario un processo di validazione che consenta di dimostrare la veridicità dei risultati forniti. Alcuni dei calcoli eseguiti dal programma sono facilmente riproponibili su altre piattaforme; invece, alcuni di essi sfruttano informazioni complesse, come ad esempio i dati orbitali della missione e pertanto sono difficili da verificare 'a mano'. Tuttavia, passando attraverso alcuni casi di studio semplici, si può effettuare una serie di test che permetta di provare la validità dei risultati forniti dalle principali subroutine. La semplicità dei test e dei risultati forniti permette il confronto dei risultati forniti dal programma con i calcoli eseguiti dallo sviluppatore (attraverso strumenti di calcolo esterni al software).

3.1 Criteri di passaggio dei test

Per poter affermare che il test sia avvenuto con successo, è fondamentale stabilire a priori dei requisiti secondo i quali il test si considera passato con successo. Nella maggior parte dei casi, il valore di riferimento con cui effettuare il paragone è la quantità di dati totali generati (chiamata *gen_tot* nel codice). Oltre a ciò, è possibile osservare, nei grafici prodotti dal programma, l'andamento dei *buffer* e delle memorie, traendone ulteriori considerazioni qualitative. Il metodo con cui, per ogni test, viene stimato il valore di *gen_tot* (e talvolta anche di *downlink_tot*, che sono i dati totali scaricati) verrà specificato in ogni paragrafo. Per le simulazioni eseguite nel seguito è stato utilizzato un *report file* di GMAT contenente 1494 passi temporali, distanziati da un *dt* uguale a 60 secondi. Questo significa che la sensibilità della simulazione non può scendere sotto tale valore.

3.2 Test 1

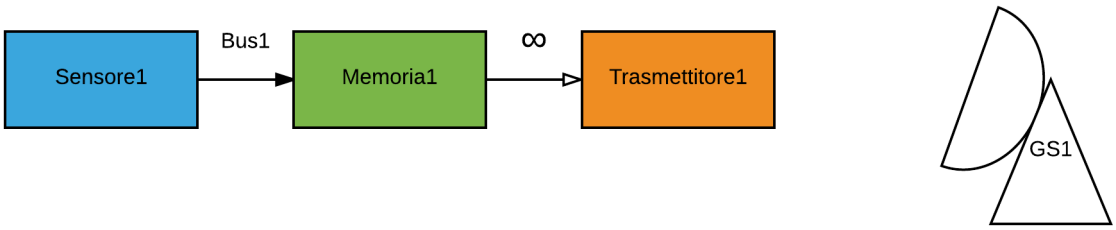

Fig. 3.1 – Il flusso di dati nel Test 1

In questo primo test, si calcola il memory budget di una piattaforma composta da un solo sensore, una memoria ed un trasmettitore. Sensore e memoria sono collegati tra loro da un *bus* e il satellite in questione effettua il downlink solo con una stazione di terra. In questo test si studia in particolare il comportamento del *buffer* del sensore, per avere una prima prova di validità del modello sviluppato. Il *Data Rate* del bus sensore-memoria (*Bus1*) ed il *Data Rate* in downlink sono stati scelti con valori molto alti, di molti ordini di grandezza maggiori rispetto a quelli generati dal sensore. Infatti, data la diversità nelle grandezze (tra il *Data Volume* in entrata nel sensore e il *Data Volume* fornito dal *bus* e scaricato dai trasmettitori), in caso di errore nei risultati è più facile localizzarne la sorgente ed eventualmente effettuare una correzione nei modelli. Inoltre, per evitare di superare la memoria massima del *buffer*, questo è stato inizializzato con una capacità molto alta. Infine, il simbolo 'infinito' sopra la freccia che collega memoria e trasmettitore indica che, per le ipotesi fatte nel paragrafo 2.1.6.2, memoria e trasmettitore si scambiano dati con un *Data Rate* infinito.
Alla luce di quanto detto, il test è stato eseguito seguendo le informazioni contenute in Tab. 3.1:

Sensore1	**Bus1**	**Memoria1**	**Trasmettitore1**
Data rate = 100 bps	Data rate = 1 TBps	Capacità = 1 TB	Data Rate in downlink = 1 TBps (no overhead e compressione)
Buffer = 1 TB	Overhead = 0x		Funzionamento di notte: Sì
Logica di attivazione: (1==1) (sempre attivo)	Compressione = 0x		Stazione di Terra: Forlì

Tab. 3.1 – Test 1, condizioni al contorno

Il criterio con cui è stato calcolato *gen_tot*, con strumenti terzi e per verificare che il valore coincida con quello fornito dal programma, è il seguente:

$$gen_{tot} = sensors.data * dt * \ simulation\_steps$$

dove *sensors.data* è il campo della *struct* dedicata ai sensori che indica il *Data Rate*, *dt* è il passo temporale con cui viene eseguita la simulazione e *simulation_steps* è il numero di passi totali eseguiti. In questo caso, inserendo i dati della Tab. 3.1 e considerando che nel primo passo temporale della simulazione non vengono raccolti dati:

$$gen_{tot} = 100 * 60 * 1493 = \ 8.958 * \ 10^6 \ bit = 8.958 \ Mbit$$

Quest'ultimo valore coincide con l'uscita *gen_tot* fornita dal software (Fig. 3.2); il primo criterio di passaggio del test è stato quindi soddisfatto.

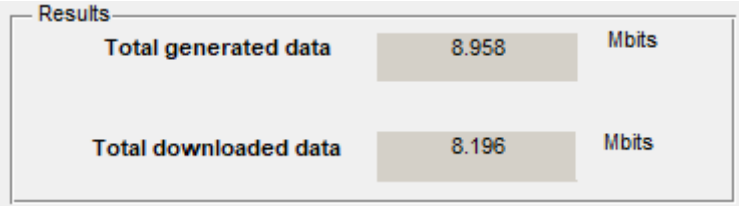

Fig. 3.2 - I dati totali generati e scaricati ottenuti in Test 1, mostrati della GUI del programma.

Si noti in Fig. 3.3 e in Fig. 3.4 rispettivamente l'occupazione del buffer e della memoria, ottenuti al termine della simulazione. Sulle ascisse sono indicati i giorni di missione trascorsi, sulle ordinate la quantità di dati residui indicata in bit. Con il primo diagramma il programma fornisce i risultati attesi: dato che il sensore in oggetto acquisisce informazioni con un *Data Rate* di 100 bps, esso avrà, per ogni time-step, un *Data Volume* di

$$SENS_{DataVolume} = SENS_{DataRate} * dt = 100 \ bps * 60 \ s = 6000 \ bit$$

Inoltre, dato che il bus collegato al sensore comincia a drenare dati dal secondo istante di simulazione e non dal primo (come spiegato in 2.9.3.3), è presente nel buffer un picco di 6000 bit iniziale, che resta costante negli istanti successivi perché il bus è entrato in funzione.

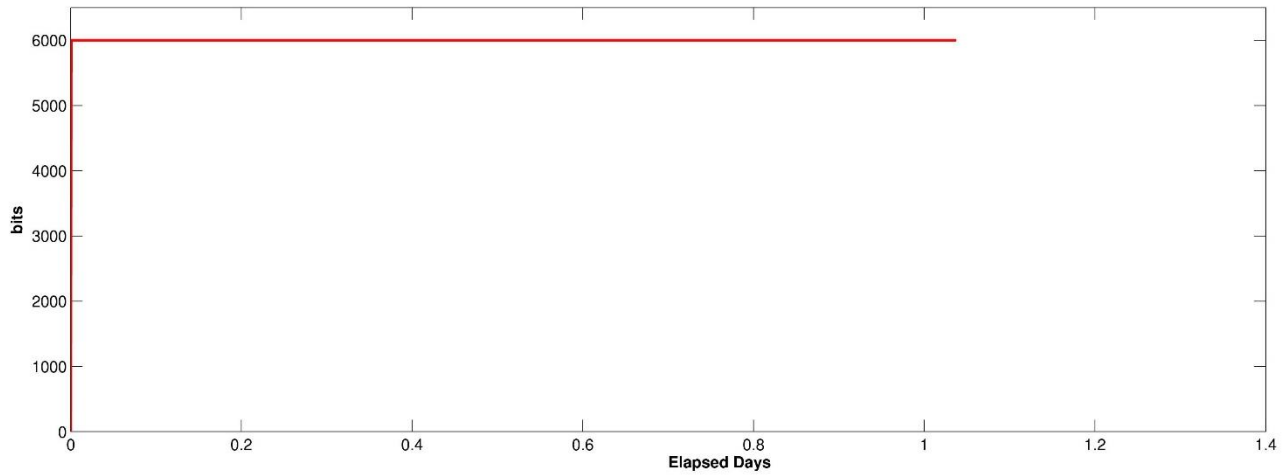

Fig. 3.3 - Test 1, 2 e 4; occupazione del buffer

Dalla Fig. 3.4 è invece possibile notare che il numero complessivo di dati scaricati (il valore *downlink_tot* nel codice) è difficile da stimare tramite formule rapide come quelle utilizzate per verificare *gen_tot*. La veridicità dei risultati forniti da *downlink_tot* verrà quindi verificata in un test ad hoc che permette, con i dovuti accorgimenti nelle condizioni al contorno, di stimare in modo semplice la mole di dati scaricati (si veda a tal proposito il paragrafo 3.4).

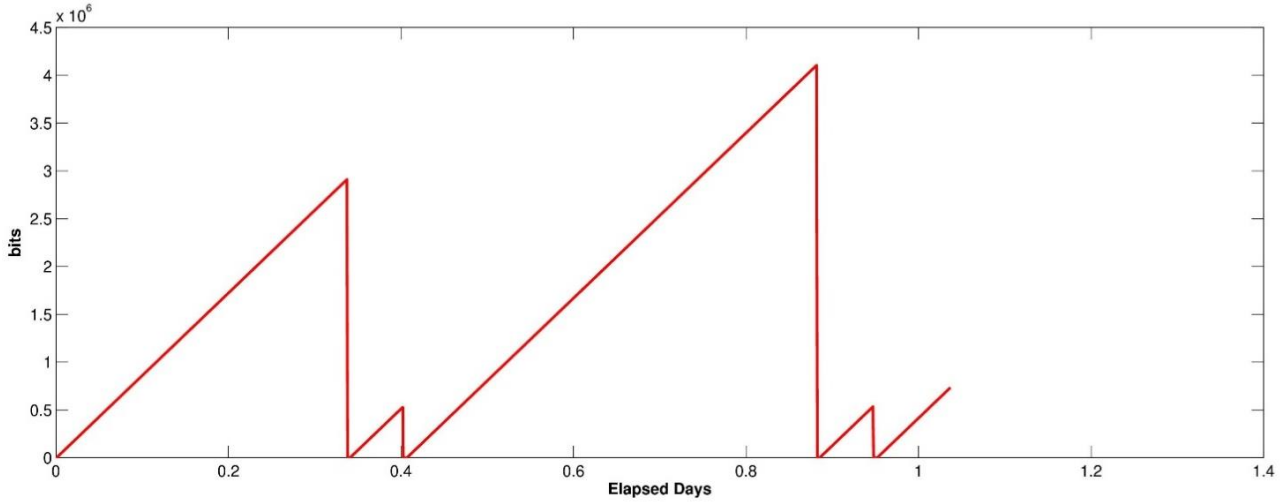

Fig. 3.4 - Test 1, occupazione della memoria

Tuttavia, è possibile effettuare una verifica grafica riferita ai massimi della curva. Si consideri, ad esempio, il primo picco della curva: esso è posizionato sull'ascissa *ElapsedDays* uguale a *0.3375*. Sapendo che la memoria varia nel tempo secondo la legge descritta nel Par. 2.9.3.4 e poiché fino a tale punto non è ancora stato effettuato il downlink, ci si aspetta che:

$$MEM_{occupation}(0.3375) = BUS_{DataVolume} * (simulation_{steps} - 1)$$

dove $BUS_{DataVolume}$, in questo caso, è costante al valore 6000 bit, mentre i passi di simulazione eseguiti possono essere stimati con la proporzione:

$$simulation_{steps} : 0.3375 = total_{simulation,\,steps} : 1.037$$

dove *total_simulation_steps* è il numero complessivo di passi di simulazione (1494) e *1.037* il punto finale della simulazione sulle ascisse. Quindi:

$$simulation_{steps} \cong 486$$

che porta a:

$$MEM_{occupation}(0.3375) = 2.91 * 10^6 bit$$

Lo stesso valore viene mostrato sul diagramma nella GUI, in corrispondenza del primo picco (Fig. 3.5), quindi il test si considera passato.

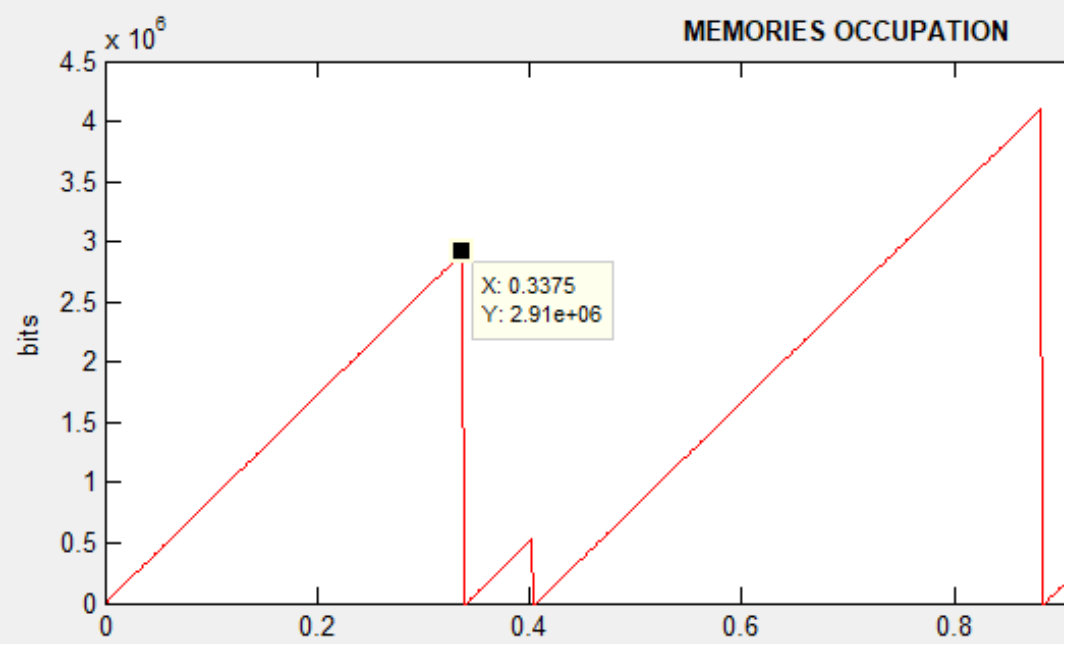

Fig. 3.5 - Verifica dell'occupazione di memoria attraverso il grafico fornito dalla GUI; Test 1

Si noti che, per stimare l'occupazione di memoria in tale punto, è stata sottratta un'unità dal valore *simulation_steps*, dato che nel primo passo del ciclo il bus non scarica dati verso la memoria.

3.3 Test 2

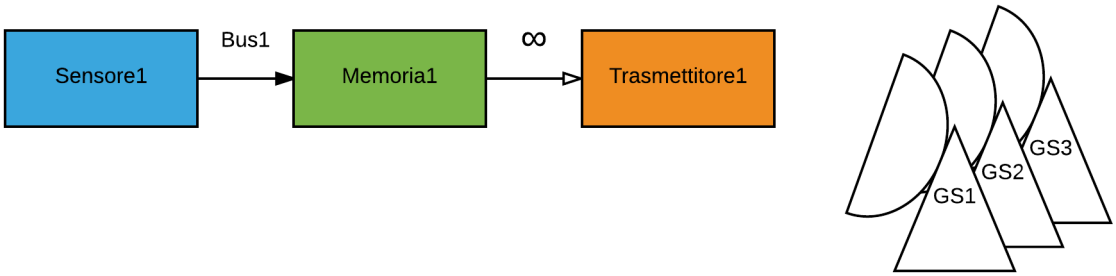

Fig. 3.6 – Il flusso di dati in Test 2

Il secondo test effettuato è del tutto identico al primo, tranne che per il numero di stazioni di terra verso cui il trasmettitore scarica i dati. Questo test serve infatti per verificare che il programma sia in grado di riconoscere il numero e la posizione delle stazioni di terra assegnate ad ogni trasmettitore. Sono stati utilizzati i dati in Tab. 3.2.

Sensore1	**Bus1**	**Memoria1**	**Trasmettitore1**
Data rate = 100 bps	Data rate = 1 TBps	Capacità = 1 TB	Data Rate in downlink = 1 TBps (no overhead e compressione)
Buffer = 1 TB	Overhead = 0x		Funzionamento di notte: No
Logica di attivazione: (1==1)	Compressione = 0x		Stazione di Terra: Forlì, Singapore, Honolulu

Tab. 3.2 – Test 2, condizioni al contorno

Per verificare anche graficamente una corretta distribuzione, nel corso del tempo, dei contatti con le stazioni di terra, si è pensato di associare a *Trasmettitore1* tre stazioni geograficamente molto distanti

tra loro: è stata ricavata una nuova propagazione con GMAT, nella quale sono state inserite le coordinate delle stazioni di Forlì (Italia), Singapore e Honolulu (Hawaii).
I dati generati (e quindi *gen_tot*) in questo secondo caso coincidono con quelli del precedente, poiché si mantiene lo stesso sensore con le stesse caratteristiche:

$$gen_{tot} = 100 * 60 * 1493 = \ 8.958 * \ 10^6 \ bit = 8.958 \ Mbit$$

I risultati ottenuti sono mostrati nella precedente Fig. 3.3 e in Fig. 3.7:

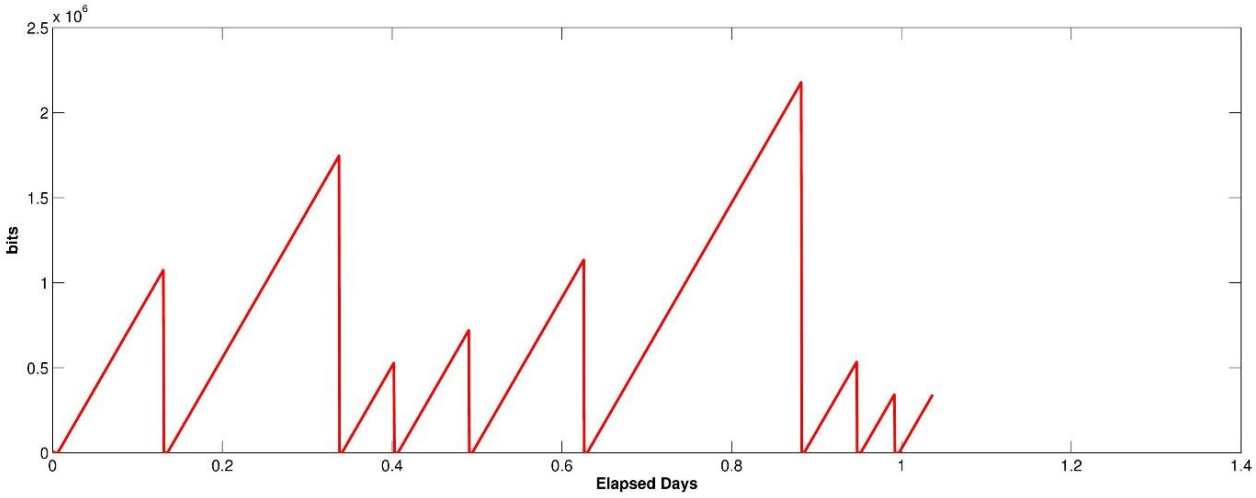

Fig. 3.7 - Test 2, occupazione della memoria

Come ci si aspettava, il buffer occupato risulta lo stesso del Test 1, dato che le configurazioni dei satelliti coincidono. Quello che cambia è l'occupazione della memoria, che in questo secondo test viene svuotata più spesso, dato che le ground station sono sparse sulla Terra. Infatti, si può notare che i picchi massimi di occupazione sono minori di quelli ottenuti nel Test 1. Anche in questo caso è possibile effettuare un'analisi grafica, attraverso considerazioni analoghe a quelle del paragrafo precedente. In questo caso si ha, prendendo come esempio il secondo massimo della curva:

$$\Delta simulation_{steps} \cong 179$$

con *ElapsedDays* uguale a *0.1306* in corrispondenza del massimo considerato. In questo caso i passi di simulazione considerati sono la differenza tra il numero di passi effettuati quando ci si trova nel massimo (*ElapsedDays = 0.1306*) e quelli effettuati prima che i dati ricominciassero ad aumentare (*ElapsedDays = 0.00625*). Si ottiene infine:

$$MEM_{occupation}(0.1306) = 1.074 * 10^6 bit$$

che è lo stesso valore mostrato dal plot in Fig. 3.8, quindi anche in questo caso il test è stato passato.

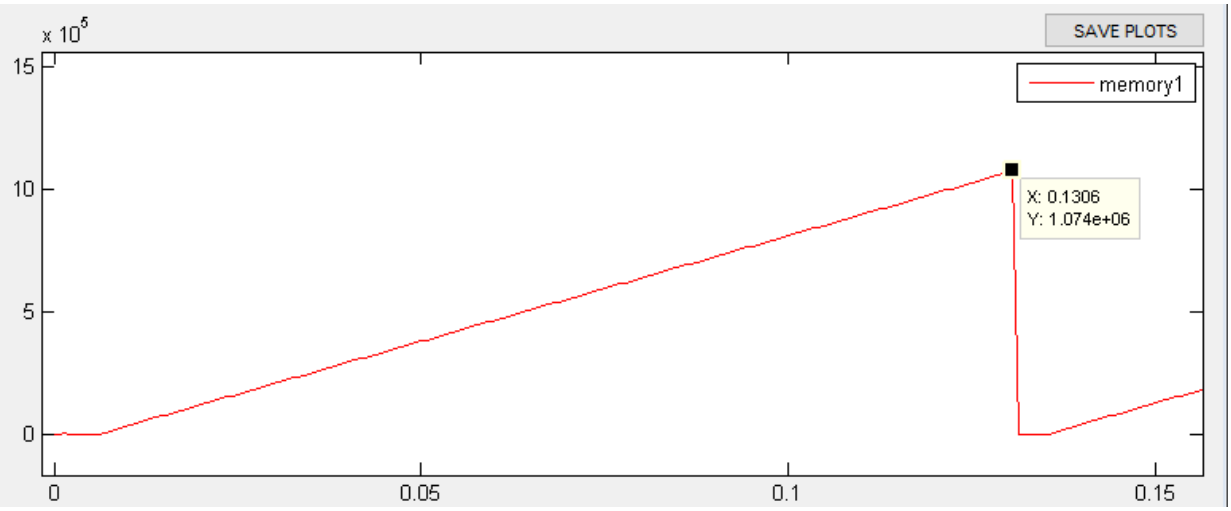

Fig. 3.8 - Verifica dell'occupazione di memoria attraverso il grafico fornito dalla GUI; Test 2. Si noti che il primo picco di occupazione è prossimo all'inizio della simulazione ed assume quindi un valore molto basso.

3.4 Test 3

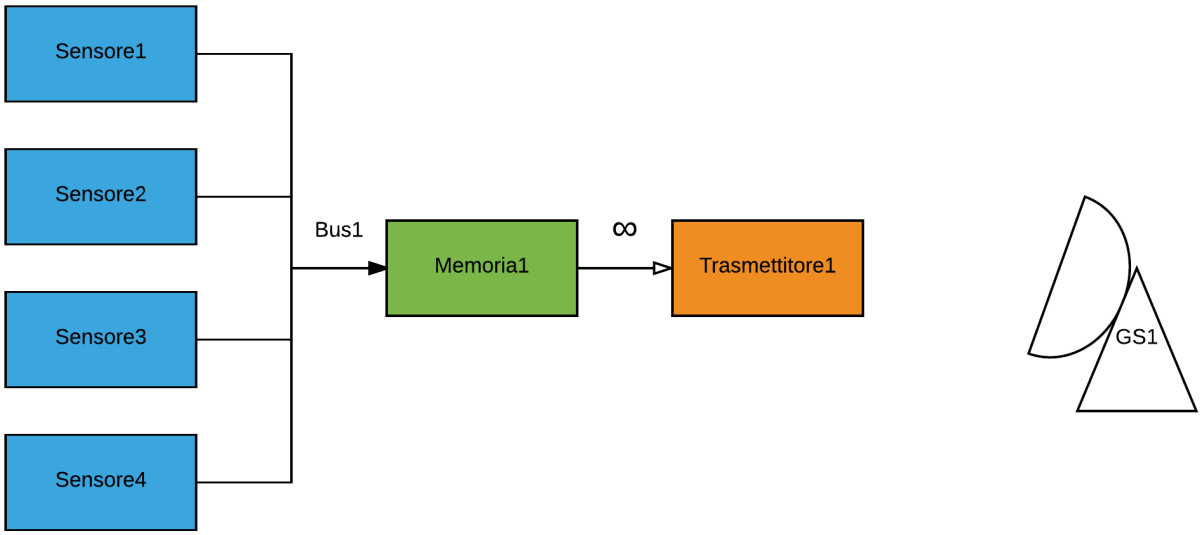

Fig. 3.9 – Il flusso di dati in Test 3

In questo terzo caso si riprende la struttura dei due precedenti, ma con quattro sensori in ingresso alla memoria (sempre tramite *Bus1*) invece che uno. In questo modo, è possibile verificare la bontà del modello utilizzato per i *bus*: esso deve infatti poter raccogliere dati da più sensori contemporaneamente, portando alla memoria che si trova a valle la somma dei *Data Volume* di ogni sensore. Si avrà:

Sensore1	**Sensore2**	**Sensore3**	**Sensore4**	**Bus1**	**Memoria 1**	**Trasmettitore 1**
Data Rate = 100 bps	Data Rate = 10 bps	Data Rate = 15 bps	Data Rate = 17 bps	Data Rate = 1 TBps	Capacità = 1 TB	Data Rate in downlink = 1 TBps (no overhead e compressione)
Buffer = 1 TB	Buffer = 1 TB	Buffer = 1 TB	Buffer = 1 TB	Overhead = 0x		Funzionamento di notte: No
Logica di attivazione: (1==1)	Logica di attivazione: (1==1)	Logica di attivazione: (1==1)	Logica di attivazione: (1==1)	Compressione = 0x		Stazione di Terra: Forlì

Tab. 3.3 – Test 3, condizioni al contorno

Avendo il satellite in questione quattro sensori, l'equazione per calcolare *gen_tot* diventa:

$$gen_{tot} = \sum_{i=1}^{4} sensors(i).data * dt * \ simulation\_steps$$

che porta a

$$gen_{tot} = (100 + 10 + 15 + 17) * 60 * 1493 = 1.27204 * 10^7 \ bit = 12.7204 \ Mbit$$

il cui valore coincide con quello mostrato dalla GUI (Fig. 3.10):

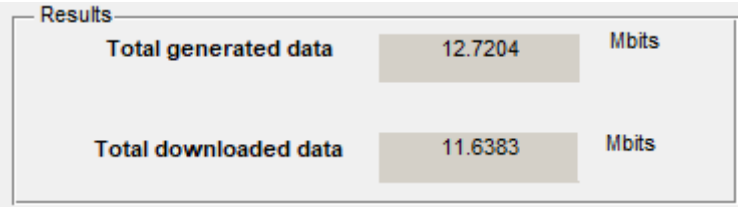

Fig. 3.10 - I dati totali generati e scaricati ottenuti in Test 3, mostrati della GUI del programma.

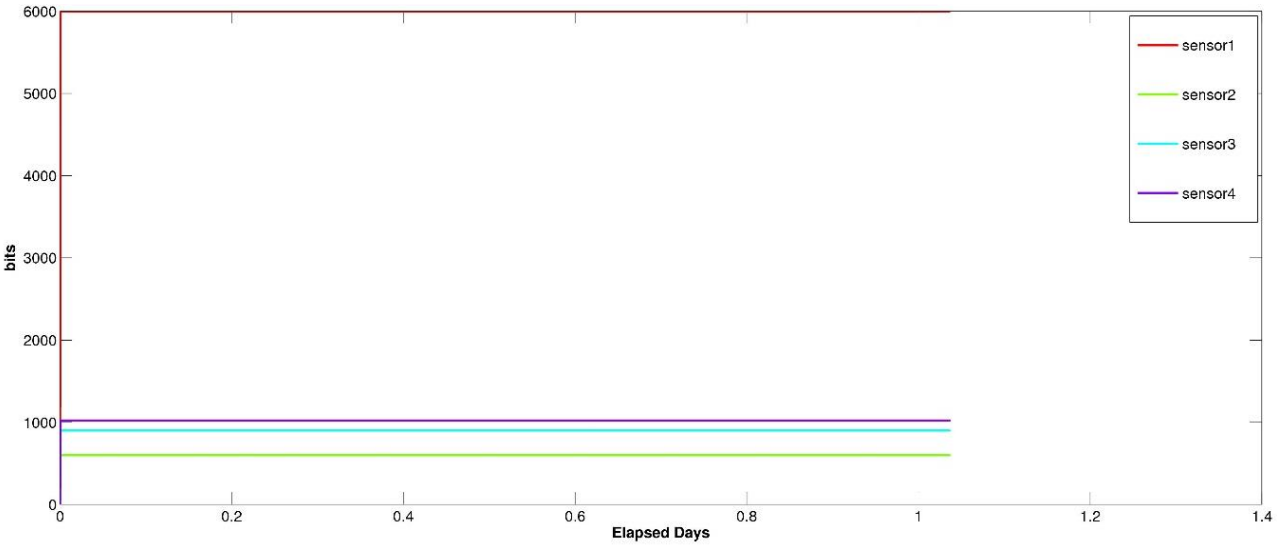

Fig. 3.11 - Test 3, occupazione dei buffer

In Fig. 3.11 sono riportate le rette orizzontali che indicano l'occupazione costante dei quattro buffer definiti. Grazie alla matrice *hsv*, il programma genera in modo automatico un colore diverso per ogni plot. Secondo i criteri illustrati nel Par. 3.2, i quattro andamenti dei buffer risultano corretti in relazione alle condizioni iniziali imposte.

L'andamento della memoria è quello in Fig. 3.12:

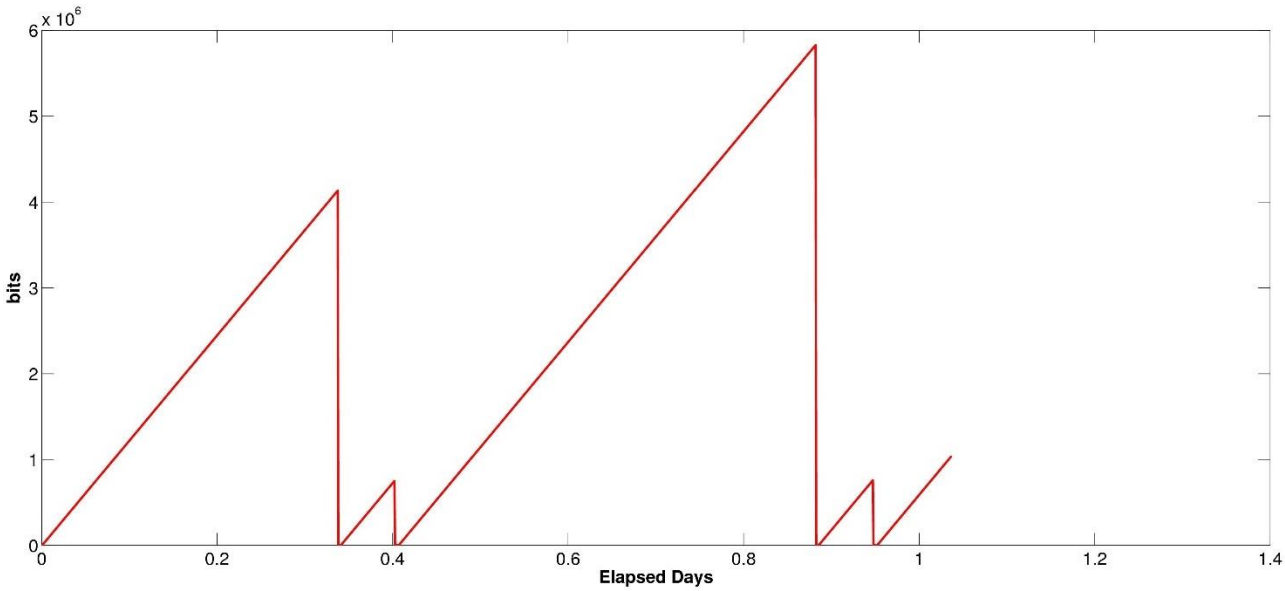

Fig. 3.12 - Test 3, occupazione della memoria

In questo caso, il valore del massimo nella curva deve essere maggiore di quello calcolato nei test precedenti. Infatti, dato che i quattro bus scaricano tutti verso lo stesso bus:

$$BUS_{DataVolume} = SENS_{Buffer1} + SENS_{Buffer2} + SENS_{Buffer3} + SENS_{Buffer4}$$
$$= 6000\ bit + 600\ bit + 900\ bit + 1020\ bit = 8520\ bit$$

per ogni istante della simulazione, tranne il primo.

In questo caso, così come nel Par. 3.2, si ha il picco per *ElapsedDays = 0.3375*, per cui:

$$MEM_{Occupation}(0.3375) = 4.132 * 10^6 bit$$

come mostrato in Fig. 3.13. Anche il terzo test è stato quindi superato.

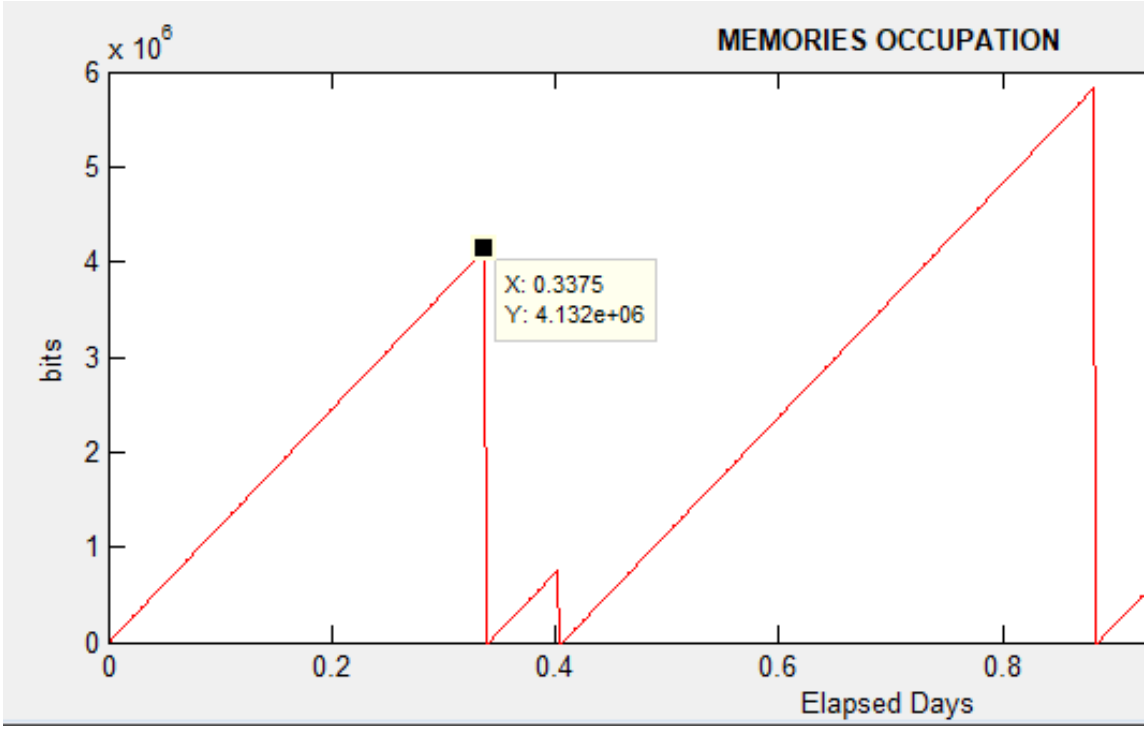

Fig. 3.13 - Verifica dell'occupazione di memoria attraverso il grafico fornito dalla GUI; Test 3.

3.5 Test 4

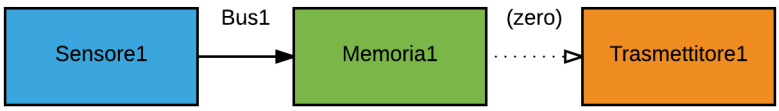

Fig. 3.14 – Il flusso di dati in Test 4

Dopo aver constatato la veridicità dei risultati forniti dal codice nel calcolo dei *buffer* e dei *Data Rate* dei *bus*, in questo quarto test viene spostata l'attenzione sul modello che descrive l'occupazione della memoria. A tal proposito, viene utilizzato lo stesso schema proposto nel paragrafo 3.2, attribuendo però il valore zero al *Data Rate* in *downlink*. Così facendo, nessun dato può essere scaricato a terra dal trasmettitore; tutti i dati raccolti dal sensore vengono quindi immagazzinati nella memoria. Questo test si considera passato qualora i dati presenti nella memoria siano uguali ai dati immagazzinati (*gen_tot*), a meno del numero di bit ancora presenti nel *buffer*, nell'ultimo passo temporale della simulazione.

I dati utilizzati sono (Tab. 3.4):

Sensore1	Bus1	Memoria1	Trasmettitore1
Data rate = 100 bps	Data rate = 1 TBps	Capacità = 1 TB	Data Rate in downlink = 0 TBps
Buffer = 1 TB	Overhead = 0x		Funzionamento di notte: Il trasmettitore non è mai attivo
Logica di attivazione: (1==1)	Compressione = 0x		Stazione di Terra: Forlì

Tab. 3.4 – Test 4, condizioni al contorno

Seguendo i criteri sopra descritti, si può scrivere che:

$$MEM_{occupation}(end) = gen_{tot} - SENS_{Buffer}(end)$$

dove *MEM_occupation(end)* e *SENS_Buffer(end)* sono le ultime componenti dei vettori dedicati all'occupazione di memoria e buffer. La seconda voce viene calcolata sapendo che, ad ogni passo, il sensore acquisisce un *Data Volume* di

$$SENS_{DataVolume} = sensors.data * dt = 100 * 60 = 6000\ bits$$

Nel primo passo della simulazione il sensore acquisisce 6000 bit, ma il *bus* ad esso collegato inizia a scaricare i dati dal secondo passo in poi. Questo significa che il *buffer* del sensore resterà costante, per tutta la durata della simulazione, a 6000 bit. Quindi

$$SENS_{Buffer}(end) = 6000\ bit$$

da cui si ottiene:

$$MEM_{Occupation}(end) = 8.958 * 10^6 - 6000 = 8.952 * 10^6\ bit = 8.952\ Mbit$$

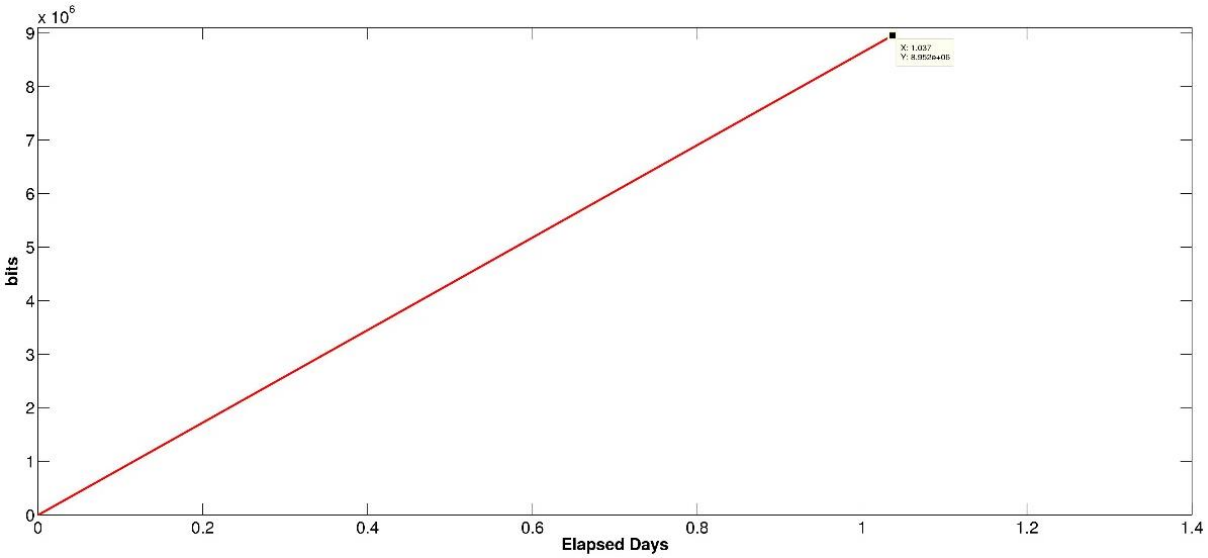

Fig. 3.15 - Test 4, occupazione della memoria

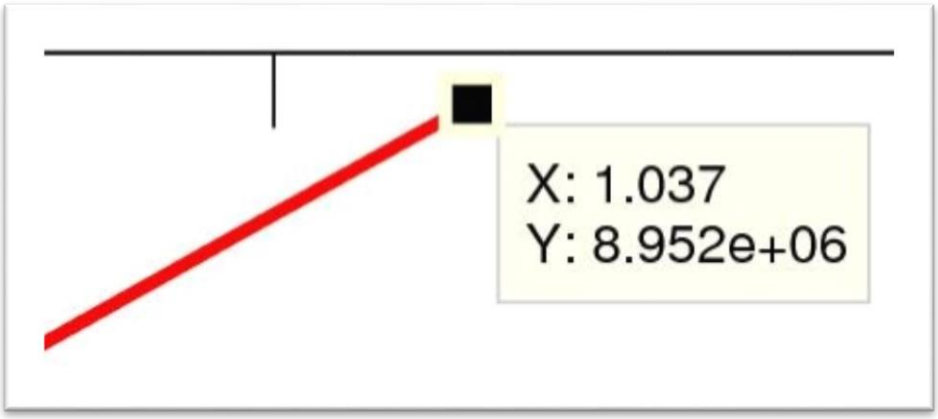

Fig. 3.16 - Test 4, occupazione della memoria; ingrandimento sul punto finale

Il plot relativo al buffer è ancora identico a quello di Fig. 3.3. La Fig. 3.15 mostra invece che l'occupazione della memoria, come atteso, aumenta sempre e raggiunge il valore atteso per *MEM_occupation* appena calcolato. Infatti, essendo il *Data Rate* del trasmettitore sempre uguale a zero, la memoria non viene mai svuotata. Il Test 4 si ritiene perciò superato.

3.6 Test 5

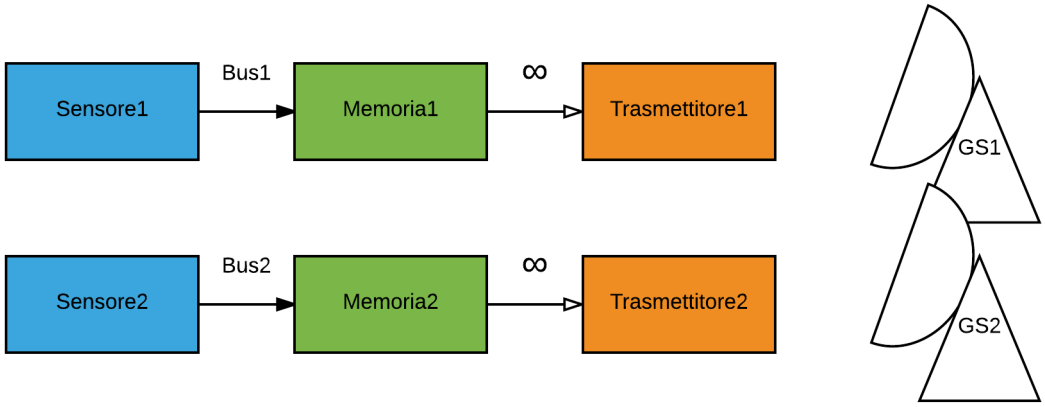

Fig. 3.17 – Il flusso di dati in Test 5

Il quinto test mette alla prova un'architettura con bus multipli per lo scambio dei dati. Infatti, come mostrato in Fig. 3.17, si usa uno schema con due percorsi in parallelo, ognuno dei quali ricalca quello del paragrafo 3.2: il satellite possiede due sensori; ogni sensore trasmette dati alla propria memoria, che viene liberata grazie al proprio trasmettitore. Ognuno dei due trasmettitori effettua il *downlink* con una diversa stazione di terra. I dati utilizzati sono i seguenti (Tab. 3.5):

Sens 1	**Sens 2**	**Bus1**	**Bus2**	**Mem1**	**Mem2**	**Tx1**	**Tx2**
Data rate = 100 bps	Data rate = 47 bps	Data rate = 1 TBps	Data rate = 1 TBps	Capacità = 1 TB	Capacità = 1 TB	Data Rate in downlink = 1 TBps	Data Rate in downlink = 1 TBps
Buffer = 1 TB	Buffer = 1 TB	Overhead = 0x	Overhead = 0x			Funzionamento di notte: No	Funzionamento di notte: No
Logica di attivazione: (1==1)	Logica di attivazione: (1==1)	Compressione = 0x	Compressione = 0x			Stazione di Terra: Forlì	Stazione di Terra: Vigo

Tab. 3.5 - Test 5, condizioni al contorno

In questo caso, il primo sensore segue l'equazione:

$$gen_{tot}(1) = sensors(1).data * dt * \ simulation\_steps$$
$$gen_{tot}(1) = 100 * 60 * \ 1493 = 8.958 * 10^6 \ bit = 8.958 \ Mbit$$

Il secondo sensore, invece, viene descritto dall'espressione:

$$gen_{tot}(2) = sensors(2).data * dt * \ simulation\_steps$$
$$gen_{tot}(1) = 47 * 60 * \ 1493 = 4.210 * 10^6 \ bit = 4.210 \ Mbit$$

Quindi, la quantità totale di dati generati risulta:

$$gen_{tot} = gen_{tot}(1) + gen_{tot}(2) = 13.168 \ Mbit$$

che è la stessa proposta dal programma (Fig. 3.18):

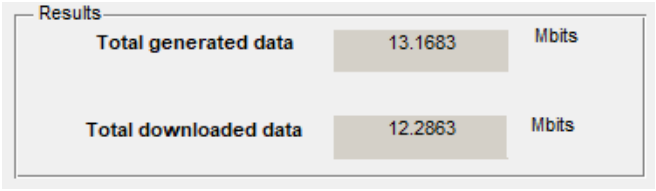

Fig. 3.18 - I dati totali generati e scaricati ottenuti in Test 5, mostrati della GUI del programma

.

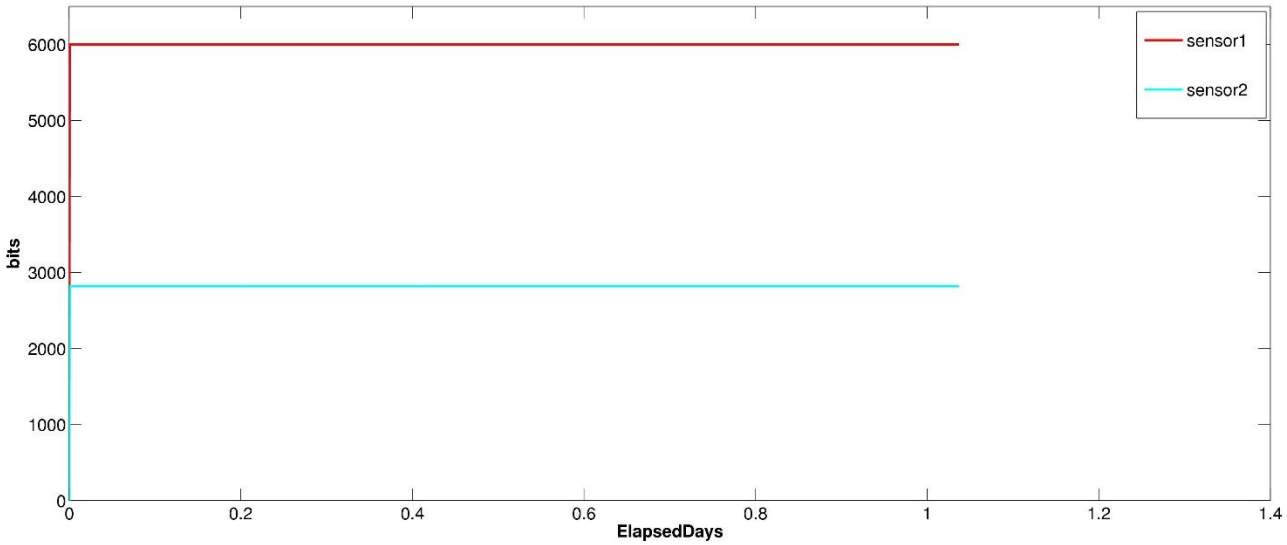

Fig. 3.19 – Test 5, occupazione dei buffer

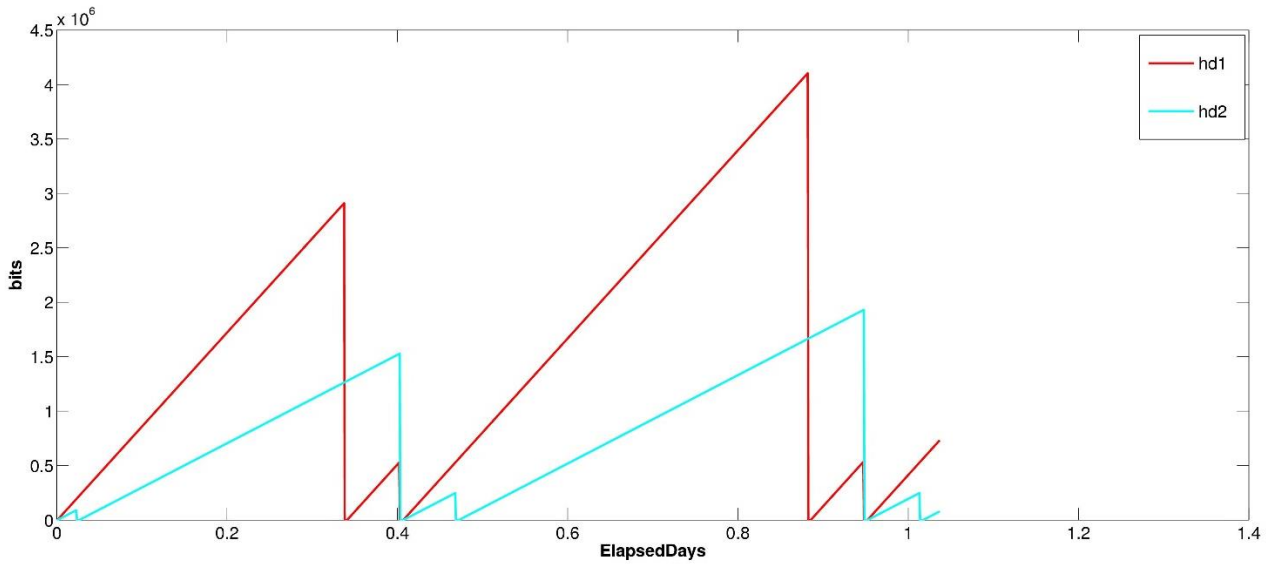

Fig. 3.20 - Test 5, occupazione delle memorie

Entrambi i buffer mantengono un'occupazione sempre uguale al valore iniziale, calcolato come visto in precedenza, dato che:

$$SENS_{DataVolume1} = 100\ bps * 60\ s = 6000\ bit$$
$$SENS_{DataVolume2} = 47\ bps * 60\ s = 2820\ bit$$

In Fig. 3.19 si vede che il sensore con il maggior *Data Rate* di acquisizione dati è anche quello con maggiore occupazione della memoria interna. Inoltre, la Fig. 3.20 mette in luce che i due trasmettitori (collegati uno ad *hd1*, l'altro ad *hd2*) effettuano il downlink in momenti diversi, come era da attendersi data la diversità tra le due stazioni di terra di riferimento. Il plot relativo alla prima memoria risulta identico a quello trattato nel 3.2, perché i *Data Rate* di bus e sensore coincidono. Per quanto riguarda il secondo sensore, è possibile stimare l'occupazione teorica della memoria ottenuta nel primo picco della curva (che si trova nella posizione *ElapsedDays = 0.02292*). La stima avviene con le stesse modalità descritte in precedenza, e coincide (anche in Fig. 3.21) con

$$MEM_{occupation}(0.02292) = 9.024 * 10^4 bit$$

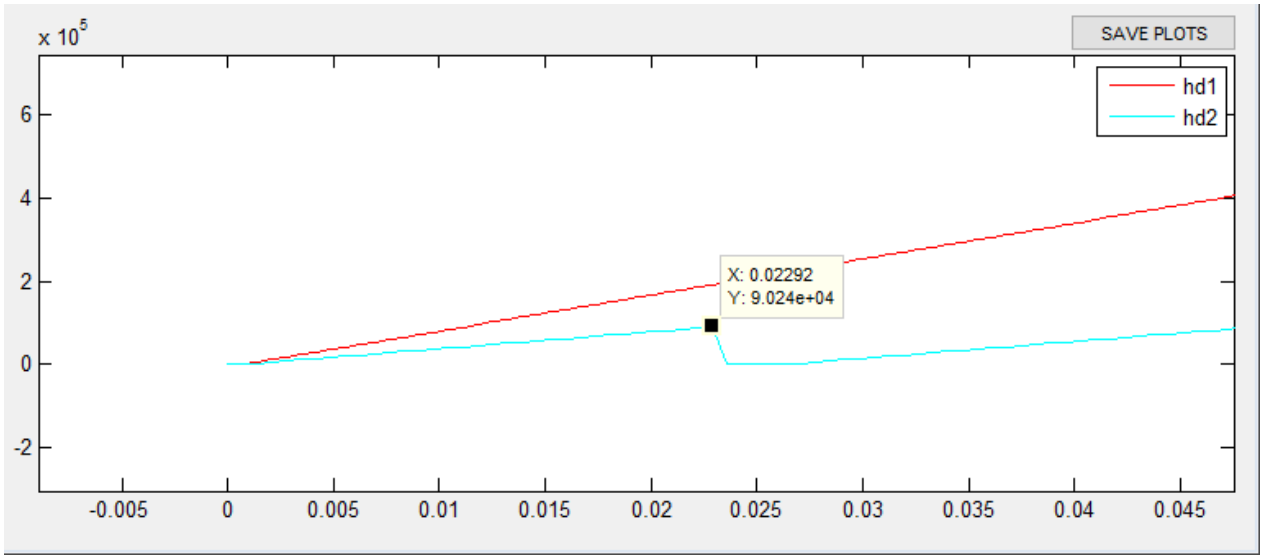

Fig. 3.21 - Verifica dell'occupazione di memoria attraverso il grafico fornito dalla GUI; Test 5;

Data la corrispondenza tra i valori forniti dal software e quelli teorici, il Test 5 si considera passato

3.7 Test 6

Nella realtà, non sempre i *Data Rate* di *bus* e trasmettitore sono di molto maggiori rispetto alle altre grandezze di scambio dati. Se il sensore acquisisse dati più velocemente di quanto il *bus* riesce a scaricare verso la memoria, si noterebbe un incremento nel tempo del *buffer* del sensore, il quale non viene mai svuotato completamente ma anzi accumula, passo dopo passo, una certa quantità di dati. Per simulare uno scenario del genere, si possono utilizzare le seguenti specifiche:

Sensore1	**Bus1**	**Memoria1**	**Trasmettitore1**
Data rate = 100 bps	Data rate = 60 bps	Capacità = 1 TB	Data Rate in downlink = 0 TBps (no overhead e compressione)
Buffer = 1 TB	Overhead = 0x		Funzionamento di notte: Il trasmettitore non è mai attivo
Logica di attivazione: (1==1) (sempre attivo)	Compressione = 0x		Stazione di Terra: Forlì

Tab. 3.6 - Test 6, condizioni al contorno

Si noti che il trasmettitore è lasciato inattivo così da poter verificare la conservazione di tutti i dati all'interno del satellite.

Ad ogni step, il *Data Volume* acquisito dal sensore è maggiore di quello scaricato dal bus. Cioè:

$$\begin{cases} DataVolume_{Sensore1} = 100\ bps * 60\ s = 6000\ bit \\ DataVolume_{Bus1} = 60\ bps * 60\ s = 3600\ bit \end{cases}$$

da cui si ricava che il buffer deve aumentare, ad ogni passo, di una quantità:

$$Incremento = SENS_{DataVolume} - BUS_{DataVolume} = 2400\ bit$$

Al termine della simulazione il valore di bit residuo nel buffer dovrà ammontare a

$$SENS_{Buffer}(end) = simulation\_steps * Incremento = 3.58\ Mbit$$

cosa che si verifica infatti in Fig. 3.22, il che permette di dichiarare il test superato.
Nella memoria esterna vengono invece raccolti

$$MEM_{occupation}(end) = simulation\_steps * BUS_{DataVolume} = 5.37\ Mbit$$

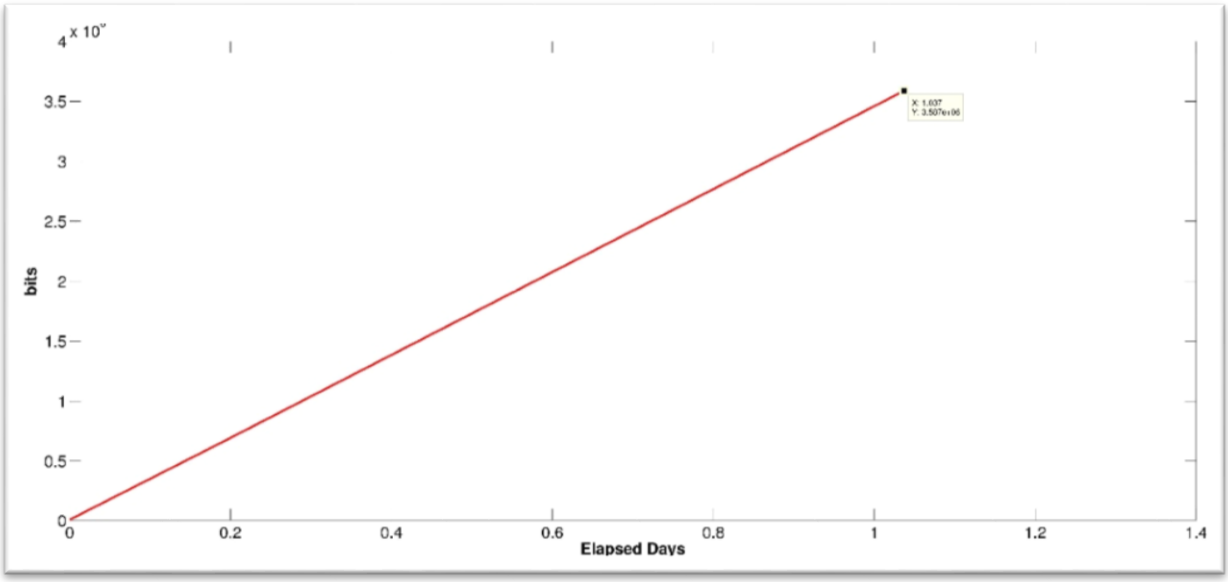

Fig. 3.22 - Test 6, occupazione del buffer

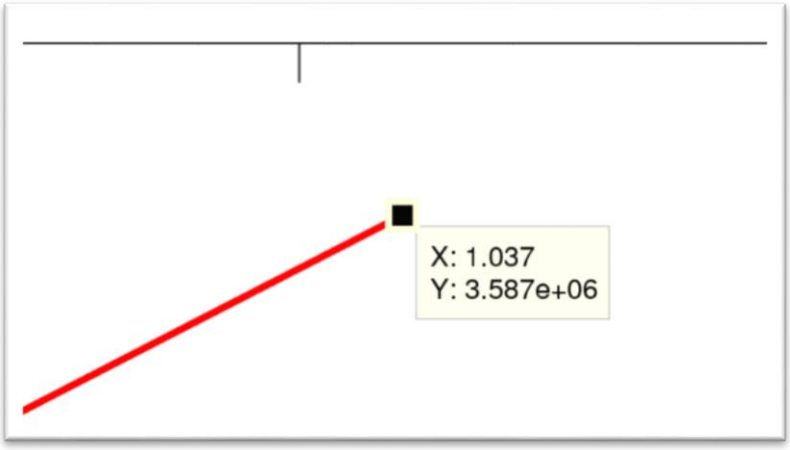

Fig. 3.23 - Test 6, occupazione del buffer; ingrandomento sul punto finale del diagramma.

3.8 Test 7

L'ultimo test effettuato permette di verificare il corretto funzionamento dello script *ACT_LOGIC*, usato per determinare gli intervalli di attivazione dei sensori, e la sua implementazione all'interno del resto del programma. Ancora una volta, si ricorre allo schema usato nel paragrafo 3.2, andando questa volta a modificare, nella configurazione, la logica di attivazione. Per verificare graficamente l'accensione e lo spegnimento dei sensori, si può utilizzare la stringa di esempio:

$$(GMATfile.ElapsedDays(t) >= 0.2)\&(GMATfile.ElapsedDays(t) <= 0.4)$$

inserita nella relativa voce in Tab. 3.7:

Sensore1	**Bus1**	**Memoria1**	**Trasmettitore1**
Data rate = 100 bps	Data rate = 60 bps	Capacità = 1 TB	Data Rate in downlink = 0 TBps (no overhead e compressione)
Buffer = 1 TB	Overhead = 0x		Funzionamento di notte: No
Logica di attivazione: $(\boldsymbol{GMATfile.ElapsedDays(t) >= 0.2)\&(GMATfile.ElapsedDays(t) <= 0.4)}$	Compressione = 0x		Stazione di Terra: Forlì

Tab. 3.7 - Test 7, condizioni al contorno

Come atteso, il sensore entra in funzione quando la variabile *ElapsedDays* è maggiore o uguale a 0.2 (4.8 ore dopo l'inizio della missione) e si spegne quando vale 0.4 (9.6 ore dopo l'istante iniziale). Il buffer e la memoria vengono riempiti di conseguenza, come mostrato in Fig. 3.24 e Fig. 3.25.

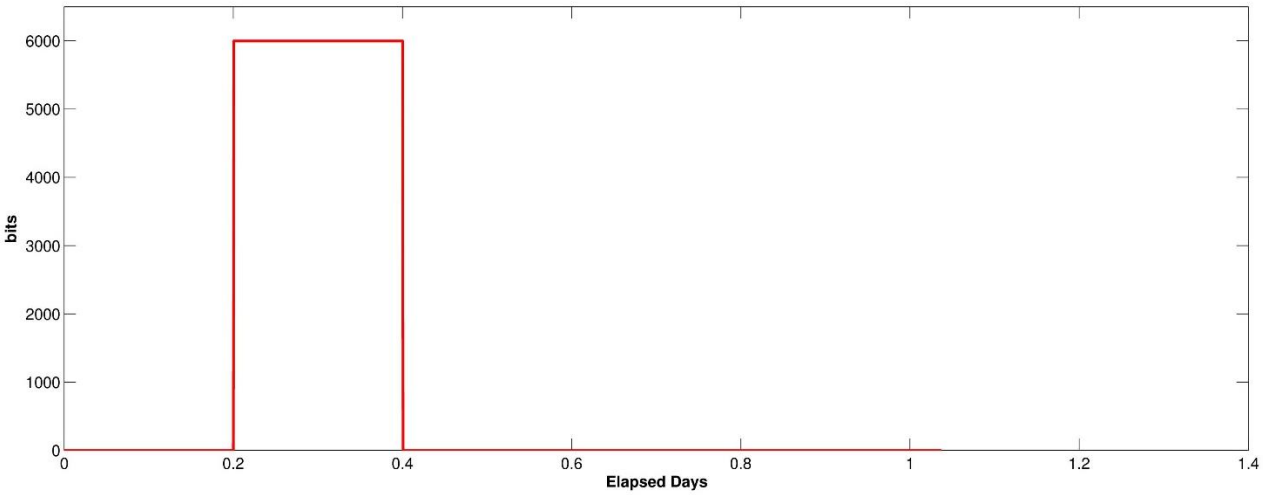

Fig. 3.24 - Test 7, occupazione del buffer

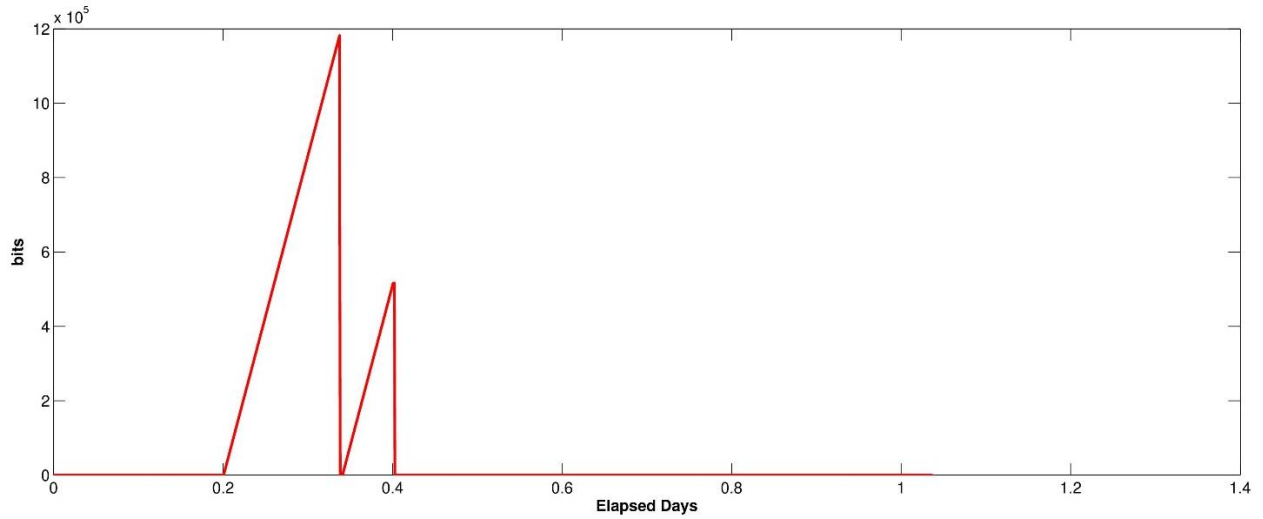

Fig. 3.25 - Test 7, occupazione della memoria

Lo svuotamento della memoria al tempo 0.4 in Fig. 3.25 non è dovuto al codice ma avviene dato che in quell'istante si è verificato anche un passaggio sulla stazione di terra.

3.9 Conclusioni sulla validazione del software

La serie di test effettuata è stata fondamentale per assicurarsi che non fossero presenti bug nelle principali subroutine e che gli output fossero corretti. La scelta di riprodurre scenari semplici ha permesso di verificare accuratamente i risultati attraverso i semplici calcoli di cui sopra.

Come anticipato, una parte non banale da verificare è l'andamento dell'occupazione di memoria (il vettore *MEM_occupation*), dato che per la sua determinazione subentrano i parametri orbitali forniti da GMAT.

Perciò, oltre a verificare che il modello della memoria è corretto (Par. 3.5), è stato simulato uno stesso scenario utilizzando diversi report file (ad es. con diverso *dt* e/o lunghezza di propagazione). I risultati ottenuti in quest'ultimo caso hanno dimostrato che le simulazioni provenienti da diversi report file sono praticamente sovrapponibili[25] e porta alla conclusione che l'intero modello sviluppato per le memorie, comprese le operazioni di downlink, lavora nel modo desiderato.

[25] Questo è valido fintanto che non si eccede con il valore di *dt.* Infatti, un time-step troppo elevato, porta alla perdita di risoluzione nella simulazione che, in questo caso, può essere la causa di un mancato downlink e/o di un numero di dati scaricati in downlink molto diverso da quello reale.

4. CAPITOLO 4
IL MEMORY BUDGET DI ESEO

I tre precedenti capitolo hanno illustrato, partendo dal livello concettuale e passando per la descrizione di dettaglio, il processo di sviluppo di un programma (e la relativa interfaccia) con il quale l'utente può configurare e simulare il memory budget di una generica piattaforma satellitare. La naturale conclusione del lavoro di tesi è stata l'impiego del programma per simulare una missione reale. In particolare, l'obiettivo finale del lavoro si identifica con la stima del memory budget per la missione ESEO, introdotta nel Par. 1.4.

4.1 I requisiti

La missione ESEO ha una durata nominale di sei mesi, durante i quali viene effettuata una serie di operazioni utilizzando i payload a bordo. Come visto nel Par. 1.5.1, lo scopo di un memory budget è quello di evitare perdite di dati causate da una memoria insufficiente e/o un downlink inefficace. La simulazione della missione in oggetto non fa eccezione, pertanto la procedura per il memory budget seguita in questo caso rispecchia concettualmente quella schematizzata in Fig. 1.4. In particolare, si vuole verificare che ogni payload del satellite sia in grado di trasmettere a terra i dati elaborati, senza perdite e senza saturare la memoria fornita. In secondo luogo, si vuole determinare il massimo picco di occupazione della memoria da parte di ognuno dei payload. Infatti, dato che è possibile partizionare via software la memoria a bordo di ESEO, in modo tale che ogni payload abbia una sotto-memoria ad esso dedicata, è opportuno quantificare il massimo numero di bit occupabili per conto di ogni strumento.

Nel Par. 1.7.2 sono raccolti i passaggi salienti della configurazione e della simulazione effettuata in GMAT, che vengono ripetuti anche in questo caso. Nell'esempio mostrato in precedenza, la durata della propagazione era di circa 24 ore; i requisiti di missione, tuttavia, richiedono che il memory budget sia verificato per l'intera durata della missione, in questo caso 180 giorni. Pertanto, in questa sede è stata propagata l'orbita di ESEO per i sei mesi di missione nominale. È stato poi generato il relativo report file, le cui variabili utilizzate sono, per quanto detto nei capitoli precedenti: *ElapsedDays*, *gscontact* e *foundeclipse*. La simulazione viene effettuata dal primo all'ultimo tipe-step presente nel file, con un *dt* di 60 secondi (Fig. 4.1).

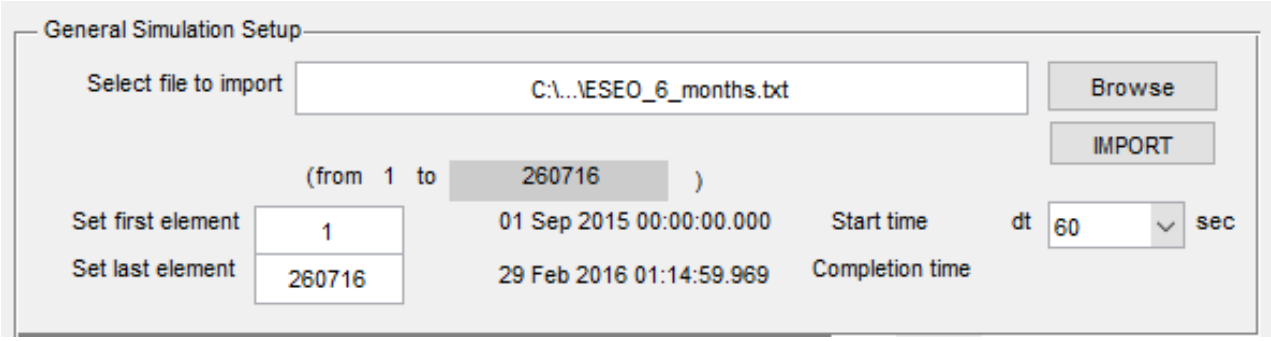

Fig. 4.1 - L'importazione del report file GMAT per ESEO.

4.2 La configurazione del satellite

Il sistema di scambio dati del satellite ESEO è stato realizzato con l'architettura mostrata in Fig. 4.2: sei payload collegati ad uno stesso CAN-bus, il quale scarica i dati verso la memoria interna del trasmettitore (*HSTX*, o *High Speed Transmitter*). L'HSTX effettua il downlink quando è in vista sulla stazione di terra di Forlì, le cui coordinate sono state inserite per inizializzare la propagazione.

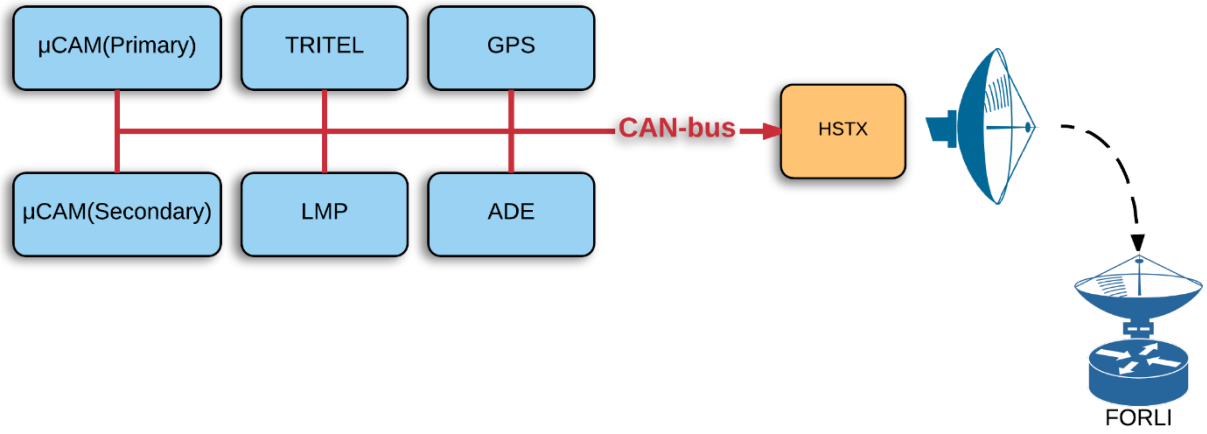

Fig. 4.2 - Il flusso di dati nella missione ESEO

4.2.1 La configurazione scelta per la simulazione

Dato che in questa simulazione l'attenzione verte sull'occupazione di memoria dell'HSTX da parte di ogni singolo sensore, lo scenario può essere modellizzato come se ogni payload comunicasse da solo, indipendentemente dagli altri, con la memoria esterna. Per realizzare un tale modello sono stati definiti sei diversi flussi di scambio dati, uno per ogni payload. Ciò significa che, nella simulazione, ogni strumento di bordo scarica i dati ad un proprio bus, che li trasmette ad un proprio trasmettitore (Fig. 4.3). In questo modo è facile estrapolare informazioni sull'andamento della

memoria a carico di un singolo sensore. Durante i sei mesi di propagazione i sei rami di scambio dati lavorano simultaneamente.

Un altro metodo possibile per verificare l'influenza di un singolo payload sulla memoria, potrebbe essere quello di accendere gli strumenti, durante i sei mesi di missione, in intervalli di tempo separati. In questo modo il memory budget si riferisce, una fase dopo l'altra, al solo payload acceso nel dato intervallo.

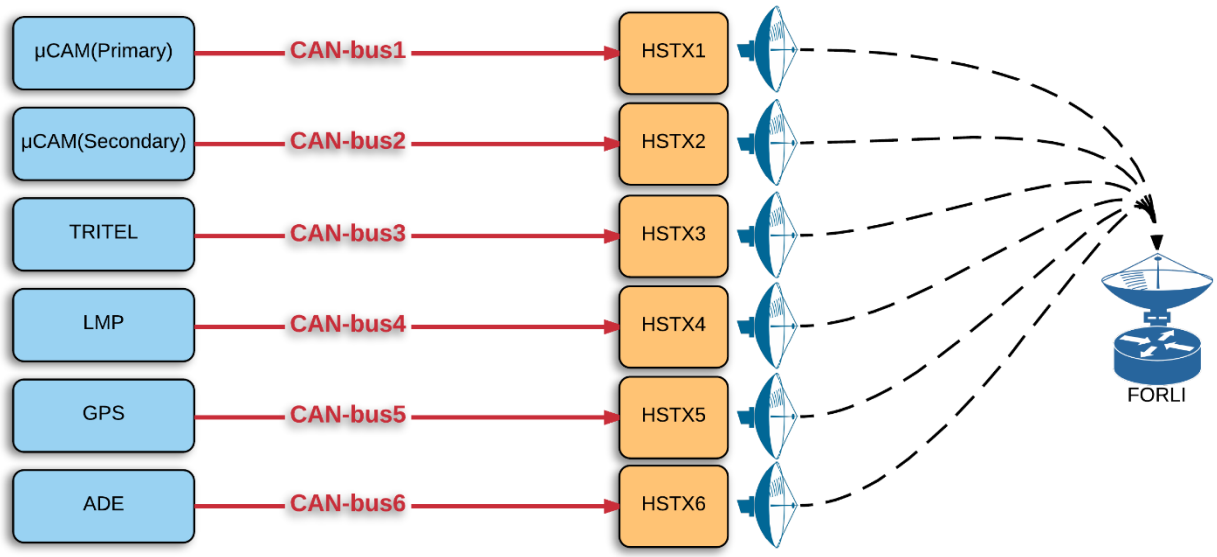

Fig. 4.3 - Il flusso di dati nel modello sviluppato

4.2.2 Il payload

Gli strumenti a bordo di ESEO svolgono ruoli molto diversi tra loro, infatti si distinguono sia in termini di *Data Rate* che di condizioni di attivazione, come visibile nella Tab. 4.1. La seconda colonna della tabella mostra che i payload non sono attivi in ogni istante della missione ed acquisiscono dati in modo discontinuo. Tuttavia, per un memory budget di primo tentativo, si può ipotizzare che questi operino con continuità. Quindi, Il *Data Rate* utilizzato è la media, in un certo intervallo di tempo, dei dati acquisiti 'a tratti' nel corso della missione. I valori riportati nella terza colonna sono stati calcolati dal codice sviluppato, che li riporta nell'elemento della GUI chiamato *StaticDataRate* (Fig. 4.4). Tutti i payload vengono mantenuti sempre attivi (logica di attivazione '1==1'), così da poter essere verificati singolarmente per l'intera durata della missione. Inoltre, per questa simulazione, il buffer di ogni payload viene inizializzato con un valore molto elevato (cioè 1 TB).

Payload	Data Rate[26]	Data Rate medio [bps]
µCAM (Primary)	(1.7 MiB + 400KiB) per orbita	3044.69
µCAM (Secondary)	59.4 MiB per giorno + 25.5 MiB per settimana	1177.57
TRITEL	4 kiB ogni 600 secondi	54.6133
LMP	400 B ogni 2 secondi	1600
GPS	392 B ogni 30 secondi	104.533
ADE	1099 B per minuto	146.533

Tab. 4.1 - Dati generati dai payload di ESEO

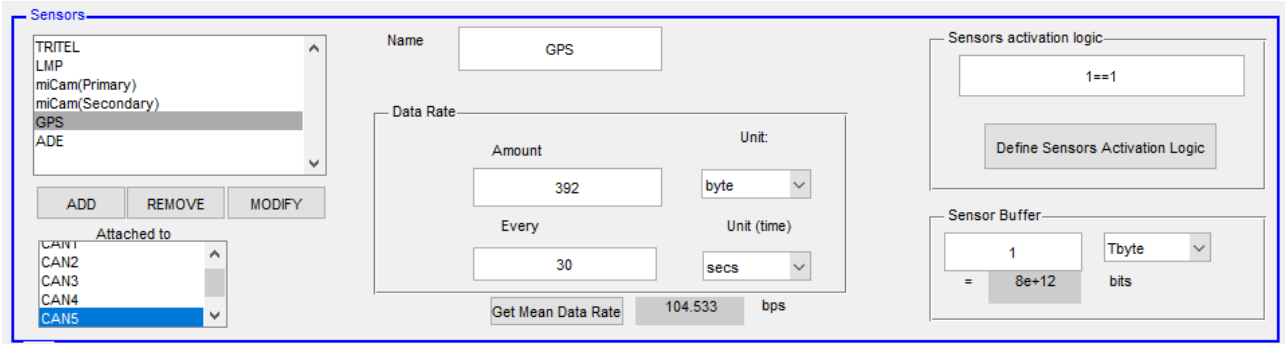

Fig. 4.4 - La configurazione di ESEO nella sezione Sensors della GUI

4.2.3 Il CAN-bus

Il bus utilizzato può raggiungere un *Data Rate* di 250 kbps, al quale va sottratta la frazione dedicata all'overhead. Tuttavia, nel nostro caso viene assunto il valore nominale di 2 kiBps, settando overhead e compressione a zero (come fatto in Fig. 4.5) dato che questo valore contiene già questa informazione. Ad ogni payload è assegnato il proprio bus, anche se questi ultimi sono uguali, in termini di *Data Rate*, per tutti gli strumenti.

[26] In questa colonna della tabella vengono talvolta utilizzate le grandezze kiB (*kibibyte*) e MiB (*mebibyte*). Queste unità di misura si distinguono da kB e MB perché sfruttano le potenze del 2 invece che del 10. Infatti, 1 kiB = $1*2^{10}$ byte, 1 MiB = $1*2^{20}$ byte.

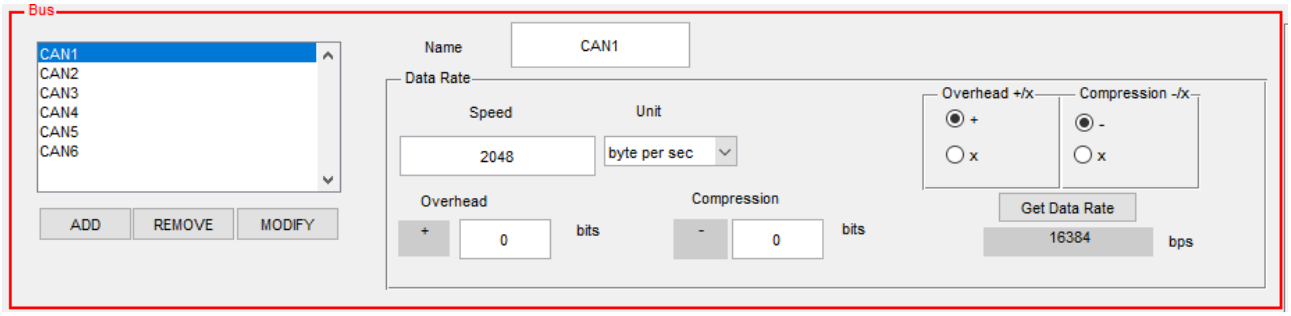

Fig. 4.5 - La configurazione di ESEO nella sezione Bus della GUI

4.2.4 Il trasmettitore HSTX

Il componente HSTX funge sia da memoria che da trasmettitore. Esso è dotato di una capacità di 64 MB, di cui il 5% è riservato. Nel modello viene quindi inserita una capacità di 60.8 MB (Fig. 4.6). Come anticipato, per la missione viene effettuata una partizione software che suddivide la capacità complessiva ed assegna ad ogni payload lo spazio di cui esso necessita. Infatti, un payload può salvare i dati acquisiti soltanto nello spazio in memoria ad esso riservato, al quale può accedere attraverso uno specifico ID.

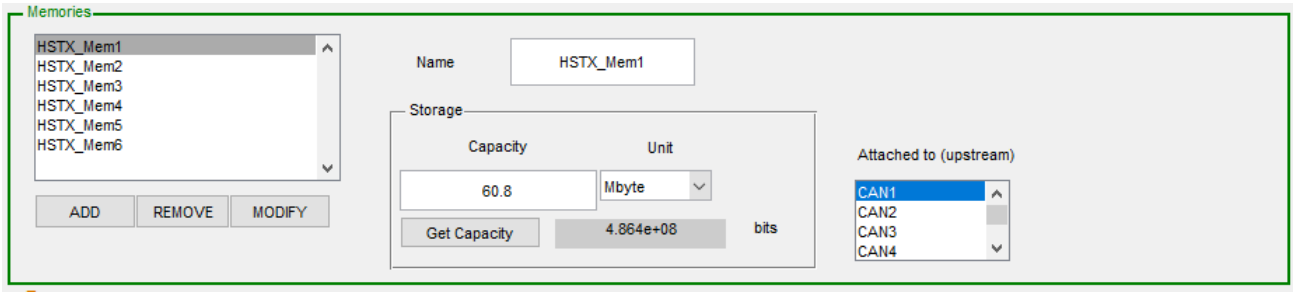

Fig. 4.6 - La configurazione di ESEO nella sezione Memories

Per modellizzare l'HSTX sia come memoria che come trasmettitore, sono state create due diverse classi di oggetti: *HSTX_Mem*, all'interno della sezione *Memories*, e *TX* nella sezione *Tx* (Fig. 4.7).

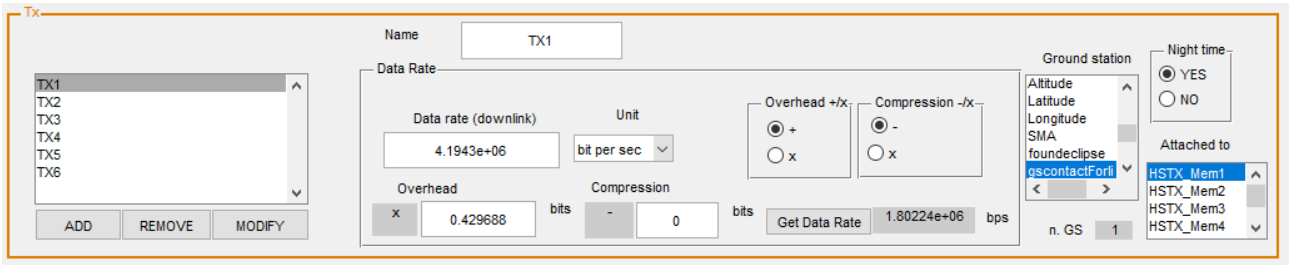

Fig. 4.7 - La configurazione di ESEO nella sezione Tx

Per quanto riguarda i parametri di trasmissione, per questa simulazione viene assunto un *Data Rate* in downlink di 4 Mibps, ulteriormente ridotto dall'overhead di un fattore 2.327. Questo significa che:

$$DataRate_{reale} = \frac{DataRate_{teorico}}{Overhead} = \frac{4.1943 * 10^6 bps}{2.327} = 1.8022 * 10^6 bps$$

I valori reali sono compresi tra 0.5 e 10 Mibps, a seconda dell'attenuazione atmosferica e della modulazione utilizzata. Come anticipato, la ground station di riferimento è il Tecnopolo di Forlì, verso la quale viene effettuato il downlink anche di notte. Infinte, su GMAT è stato settato l'angolo di maschera (paragrafo 1.7.4) a 10 gradi.
Con questo si conclude la configurazione del modello, completata la quale è possibile avviare la simulazione.

4.2.5 I risultati della simulazione

Una volta che il codice ha completato la simulazione, sono disponibili i plot dedicati ai buffer ed alla memoria, oltre alla quantità di dati complessivamente acquisita e scaricata.

4.2.5.1 L'occupazione dei buffer

La quantità di bit presente all'interno dei buffer è mostrata in Fig. 4.8: essendo, per tutti i sensori, il *Data Volume* fornito dal CAN sempre maggiore del *Data Volume* di acquisizione, l'occupazione dei buffer resta costante nel corso dei sei mesi. La memoria residua (molto diversa tra i vari casi) presente in ogni buffer è dovuta al fatto che, nel modello, il bus comincia a prelevare dati dal sensore all'istante *t_0+1*, mentre l'acquisizione da parte del payload comincia già all'istante *t_0*. Perciò la memoria occupata nei buffer coincide con il *Data Volume* acquisito dal sensore durante il primo time-step. In particolare, utilizzando la relazione

$$DataVolume = DataRate * dt$$

e sapendo che *dt* è costante, si ottiene

$$DataVolume_{TRITEL} = 54.6133 * 60 \cong 3.277 * 10^3\ bit$$
$$DataVolume_{LMP} = 1600 * 60 \cong 9.6 * 10^4\ bit$$
$$DataVolume_{miCAM(P)} = 3044.69 * 60 = 1.827 * 10^5\ bit$$

$$DataVolume_{miCAM(S)} = 1177.57 * 60 = 7.065 * 10^4\ bit$$

$$DataVolume_{GPS} = 104.533 * 60 \cong 6.272 * 10^3 bit$$

$$DataVolume_{ADE} = 146.533 * 60 \cong 8.792 * 10^3\ bit$$

così come mostrato dalla Fig. 4.8.

Si può notare nel plot riportato che l'occupazione massima dei buffer (prodotta dalla fotocamera principale) si aggira intorno ai 200 kbit, un valore ragionevole e non eccessivamente elevato rispetto alla ipotetica memoria interna del payload.

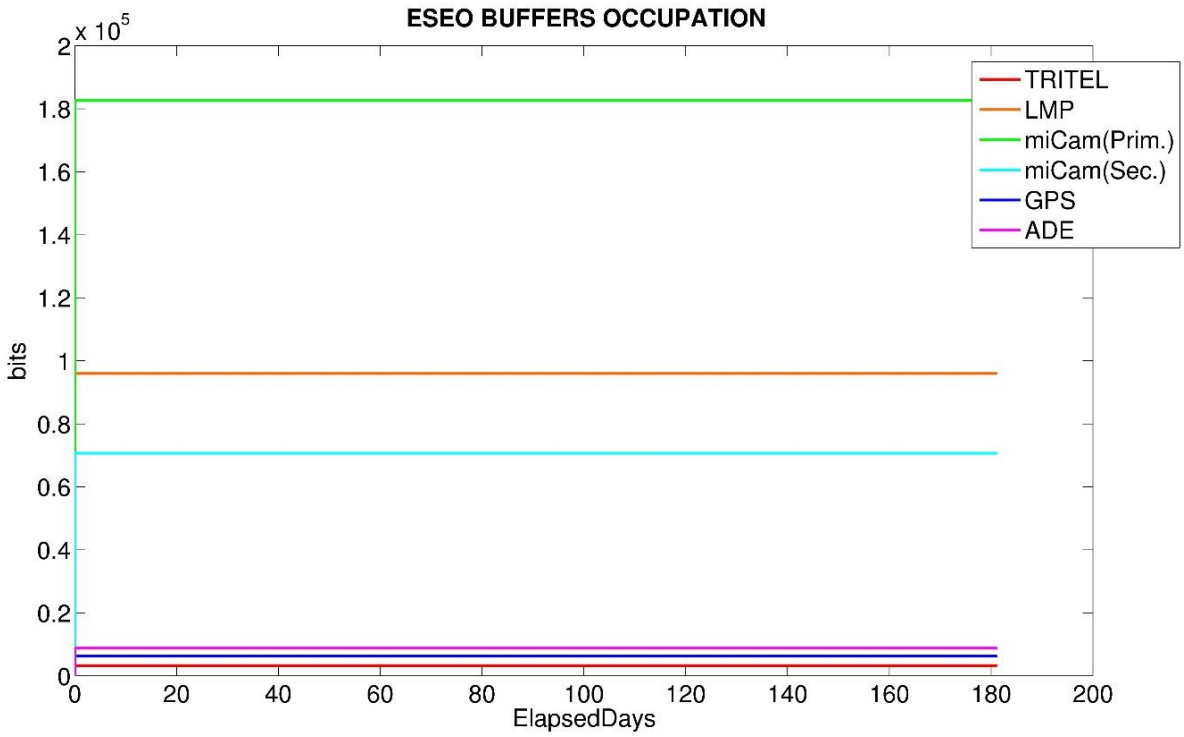

Fig. 4.8 - Occupazione dei buffer nella missione ESEO

4.2.5.2 L'occupazione della memoria

Date i differenti *Data Rate* di acquisizione, la memoria dell'HSTX viene occupata in modo assai diverso a seconda del payload. Dal diagramma riportato (in Fig. 4.9 e Fig. 4.10) è già possibile osservare che la quota di alcuni plot è di almeno un ordine di grandezza superiore rispetto ad altri.

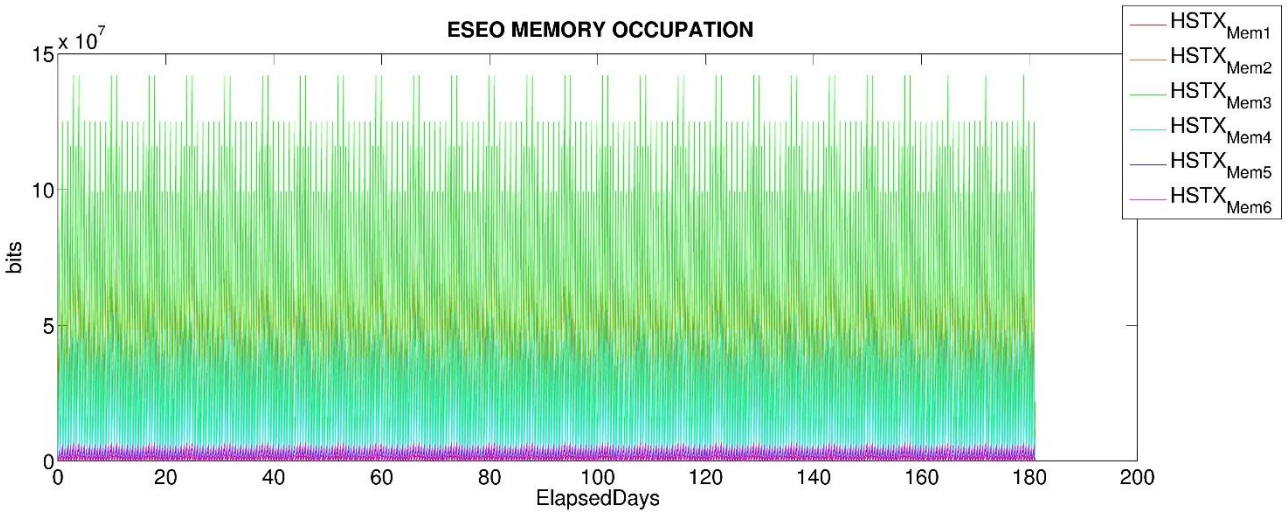

Fig. 4.9 - Occupazione della memoria da parte dei diversi payload di ESEO

Ciononostante, dato che la simulazione non ha segnalato l'errore, nessun payload ha portato la memoria a saturazione (cioè ad eccedere il valore predefinito di 60.8 MB).

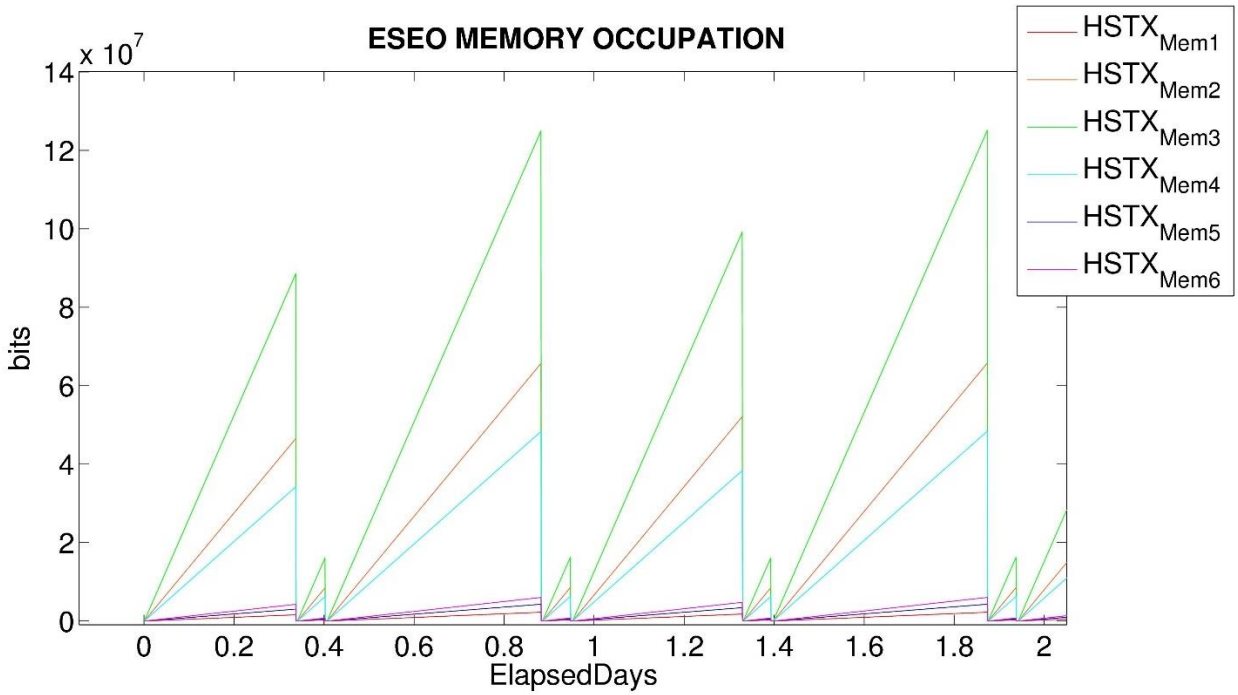

Fig. 4.10 - Occupazione della memoria da parte dei diversi payload di ESEO; ingrandimento sui primi due giorni di propagazione

La Fig. 4.10 mostra un ingrandimento del diagramma precedente sui primi due giorni di simulazione: come è possibile vedere, dato che i trasmettitori nel modello fanno tutti riferimento alla ground station di Forlì, la memoria viene svuotata sempre negli stessi istanti. Il downlink si manifesta nell'immagine con l'improvviso crollo dei dati nella memoria. Il decremento così rapido dell'occupazione della

memoria, durante la finestra di downlink, mette in evidenza che il *Data Volume* fornito dal trasmettitore è molto più elevato della quantità di dati in arrivo dai sensori.
La validità dei risultati è confermata dal fatto che i valori di occupazione della memoria sono compatibili con i valori teorici attesi definiti dalla relazione presentata nel Par.3.2. Infatti, prendendo come riferimento lo riempimento di memoria dovuto all'acquisizione del TRITEL, dato che il primo downlink avviene alla coordinata *ElapsedDays = 0.03375*, ci si aspetta:

$$MEM_{occupation}(0.03375) = 1.589 * 10^6 bit$$

Quest'ultimo valore calcolato coincide infatti con quello mostrato in Fig. 4.11, nel punto in cui la curva raggiunge il suo primo massimo. Alla luce delle considerazioni fatte, l'intera simulazione svolta per il memory budget di ESEO si considera valida.

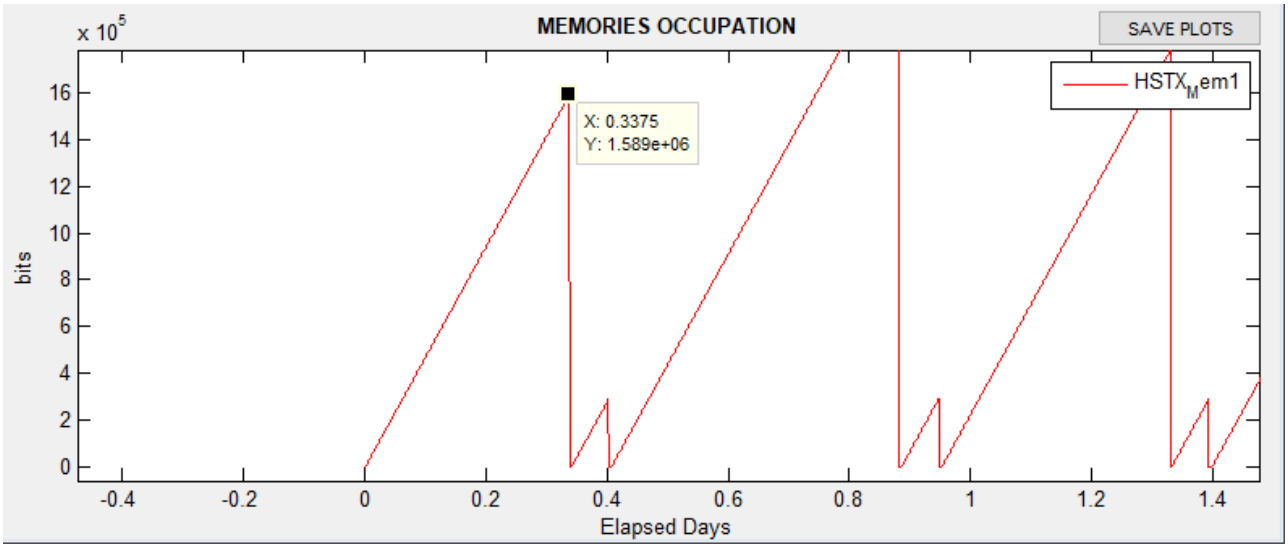

Fig. 4.11 - L'occupazione di memoria dovuta al TRITEL, al primo massimo, coincide con il valore teorico calcolato.

Analizzando l'andamento dell'occupazione di memoria nei sei casi in esame, questa simulazione ha permesso di determinare il massimo valore di dati immagazzinati nell'HSTX da ogni payload. I picchi raggiunti sono stati raccolti in Tab. 4.2 e nel diagramma di Fig. 4.12. I risultati ottenuti dal programma sono in seguito stati validati tramite confronto con analisi precedenti fornite da SITAEL, che hanno fatto da riferimento.

Payload	Occupazione massima dell'HSTX [MB]
µCAM (Primary)	17.77
µCAM (Secondary)	6.86
TRITEL	0.32
LMP	9.31
GPS	0.61
ADE	0.85

Tab. 4.2 - Occupazione massima della memoria da parte dei vari payload, nella missione ESEO

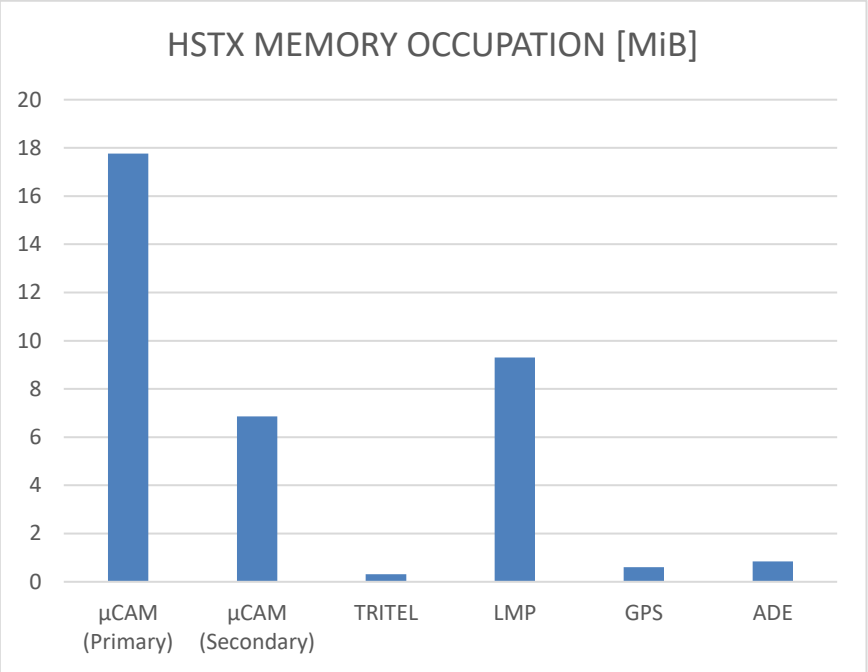

Fig. 4.12 - Diagramma a barre sull'occupazione dell'HSTX da parte dei payload di ESEO

5. CAPITOLO 5
CONCLUSIONI E SVILUPPI FUTURI

5.1 Conclusioni

L'obiettivo preposto per il lavoro di tesi era la realizzazione di un programma in grado di effettuare simulazioni per la definizione del memory budget di un satellite, le cui realizzazione e validazione sono riportate e descritte nel presente elaborato.

Il software, sviluppato in ambiente MATLAB tramite il pacchetto *GUIDE*, fornisce all'utente un'interfaccia grafica per la configurazione della piattaforma satellitare, permettendo di simulare un ampio spettro di architetture del sistema di scambio dati.

Il programma è in grado inoltre di importare i dati da un propagatore orbitale, svolgendo così il memory budget del modello all'interno di uno scenario realistico e di interesse pratico. Infine, esso fornisce le uscite relative all'occupazione delle memorie e dei buffer, ai dati acquisiti e scaricati. In questo modo, partendo dalle caratteristiche di sensori, memorie, bus e trasmettitori ed a seconda del criterio con cui questi vengono collegati, l'utente è in grado di sapere se il satellite in oggetto rispetta i requisiti operativi di missione.

Il programma realizzato è passato attraverso una serie di test di validazione, nei quali è stato previsto il soddisfacimento di determinati requisiti tecnici. Questi ultimi hanno permesso di provare la validità di tutte le uscite fornite dal software, anche attraverso un confronto con calcoli eseguiti da strumenti terzi. Ogni test è stato passato, dimostrando la veridicità dei risultati forniti da ogni subroutine in alcuni casi applicativi di riferimento.

Una volta provata la qualità del software sviluppato, questo è stato impiegato per la definizione del memory budget relativo alla missione ESEO. A tal proposito, partendo dalle specifiche di missione, è stata configurata ed eseguita la propagazione orbitale della missione in GMAT ed è stato modellizzato il sistema di data handling del satellite all'interno del software sviluppato. I risultati forniti hanno infine permesso di dimensionare le memorie interne del satellite ESEO.

Il lavoro di tesi ha spaziato in diversi ambiti riguardanti l'analisi di una missione spaziale, tra cui la meccanica orbitale di un satellite, l'utilizzo di metodi numerici per la simulazione di un'orbita, la stesura di budget di sistema per un satellite e lo sviluppo di codice in ambiente MATLAB.

5.2 Sviluppi futuri

Il modello su cui lavora il software realizzato si basa su alcune ipotesi semplificative, illustrate nel corso dell'elaborato. Queste ipotesi fanno sì che il programma sia ottimizzato per la simulazione di

un satellite LEO che sfrutta la memoria interna del trasmettitore (come ad esempio ESEO). È possibile, in futuro, migliorare il modello per renderlo completamente valido anche per le simulazioni di satelliti in orbita MEO o GEO. A tal proposito, è necessario rimuovere l'ipotesi relativa alla fase operativa notturna del satellite. Infatti, nel modello attuale, gli istanti in cui il satellite si trova in eclissi vengono tradotti come istanti in cui, nel punto sotto satellite, è notte. Tuttavia, maggiore è la quota, maggiore è l'errore introdotto da questa ipotesi.

Una seconda implementazione possibile è l'aggiunta di un modello per il bus che collega memoria e trasmettitore. Infatti, se nel caso attuale la velocità di trasmissione dati tra memoria e trasmettitore è sempre infinita, nei casi reali (in cui la memoria non è interna al trasmettitore) il bus introduce un limite fisico al *Data Volume* passante.

Inoltre, il programma allo stato attuale è in grado di modellizzare soltanto componenti che abbiano, in ingresso e/o in uscita, un solo bus. Sarebbe interessante in futuro migliorare l'algoritmo per permettere l'interfaccia tra un generico componente ed un numero di bus maggiore di uno.

Per finire, il codice è per ora in grado di importare i dati orbitali forniti da GMAT; per espanderne ulteriormente le potenzialità, è possibile implementare l'interfaccia del programma con altri propagatori orbitali (ad esempio *STK*).

APPENDICE A

LA FUNZIONE RUNTIME

```
function [SENS_Buffer,MEM_occupation,data_rate_mean,downlink_tot,gen_tot,error_flag] =
mem_bud_run(sensors,buses,memories,txs,GMATfile,t_0,t_f,handles,PlotSens,PlotMem)

%Computes and plots buffers and memories occupation. Estimates total
%generated and downloaded data.


%%INIZIALIZATION
axes1 = handles.axes1;
axes3 = handles.axes3;
error_flag = 0;


%-------------------------------------------------------
%DEFINE TIMING
%-------------------------------------------------------
time = length(GMATfile.ElapsedDays);
dt = [0 ; diff(GMATfile.ElapsedDays * 86400)];

%-------------------------------------------------------
%CALCULATE SENSORS BUFFER
%-------------------------------------------------------

%j stands for j-th sensor
num_sens = length(sensors);
for j = 1 : num_sens

    %-----definition of a logic 0/1 for 'SENS_activation'
    for t = 1 : time
        SENS_activation(j,t) = eval(sensors(j).activation_condition);
    end

    %-----------------------------------
    bus_index(j) =
find(strncmp(sensors(j).attached_to,{buses.name},length(sensors(j).attached_to)));
    SENS_Buffer(j,:) = zeros(1,time);
    SENS_DataVolume(j,:) = zeros(1,time);
    SENS_DataRate(j,:) = zeros (1,time);
    BUS_DataVolume(j,:) = zeros(1,time);
    gen_tot(j,:) = zeros(1,time);

    %---simulation starts:
    for i=(t_0+1):t_f
        SENS_DataVolume(j,i) = sensors(j).data * dt(i) * SENS_activation(j,i);
        %---defines data rate for calculating mean data rate
        SENS_DataRate(j,i) = sensors(j).data*SENS_activation(j,i);
        num_active_sens=sum(SENS_activation(:,i));
        BUS_DataVolume(j,i) = min(SENS_Buffer(j,i-1),buses(bus_index(j)).data/num_active_sens *
dt(i) * (SENS_Buffer(j,i-1) > 0));
        SENS_Buffer(j,i) = max(0,SENS_Buffer(j,i-1) + SENS_DataVolume(j,i) - BUS_DataVolume(j,i));
        %----calculates total generated data (from sensors)
```

```
        gen_tot(j,i) = gen_tot(j,i-1) + SENS_DataVolume(j,i);
        %---terminates the loop if buffer exceeds the one set from the user
        if SENS_Buffer(j,i) > sensors(j).buffer
            error_flag = 1;
            break
        end
    end

end

%---------
%SENSORS PLOT
%---------
col = hsv(num_sens);
hold(axes1,'off')
for j = 1 : length(PlotSens)

plot(axes1,GMATfile.ElapsedDays,SENS_Buffer(PlotSens(j),:),'color',col(j,:),'displayname',sensors(j
).name)
    hold(axes1,'on')
end
title(axes1,'SENSORS BUFFER OCCUPATION','fontweight','bold')
xlabel(axes1,'Elapsed Days');
ylabel(axes1,'bits');
legend(axes1,'location','ne');

%---calculates mean data rate
data_rate_mean = mean(SENS_DataRate');
```

```
%-----------------------------------------------------------
%CALCULATE MEMORIES OCCUPATION
%-----------------------------------------------------------
%---y stands for y-th memory
num_mem = length(memories);
for y = 1 : num_mem

    downlink_tot(y,:) = NaN(1,time);
    bus_index =
find(strncmp(memories(y).attached_to_upstream,{buses.name},length(memories(y).attached_to_upstream)
));
    sens_index(y,:) =
find(strncmp({sensors.attached_to},buses(bus_index).name,length(char({sensors.attached_to}))));

    tx_index = find(strncmp(txs(y).attached_to,{memories.name},length(txs(y).attached_to)));
    MEM_occupation(y,:) = zeros(1,time);
    diffMEM = [];
    TX_activation_night = zeros(1,time);

    %---definition of a logic 0/1 for 'TX_activation_night', pt.1

    can_night(y) = txs(y).night;
    is_eclipse = GMATfile.foundeclipse;
    %---it continues inside the following 'for' cycle...

    %---definition of a logic 0/1 for 'gs_contact'
    for g = 1 : txs(y).num_GS_selected
        gs_selected = char;
```

```
        gs_selected(g,:) = txs(y).GS(g,:);
        gs_flag(g,:) = {GMATfile.(strtrim(gs_selected(g,:)))};
        gs_flag_vect(g,:) = cell2mat(gs_flag(g,1));
    end
    if g > 1
        for g = 2 : txs.num_GS_selected
            gs_flag_vect(g,:) = or(gs_flag_vect(g,:),gs_flag_vect(g-1,:));
        end
    end

    %---simulation starts:
    for i=(t_0+1):t_f
        %---...definition of a logic 0/1 for 'TX_activation_night, pt.2
        if can_night(y) == 1 && is_eclipse(i) == 1
            TX_activation_night(i) = 1;
        elseif can_night(y) == 1 && is_eclipse(i) == 0
            TX_activation_night(i) = 1;
        elseif can_night(y) == 0 && is_eclipse(i) == 1
            TX_activation_night(i) = 0;
        elseif can_night(y) == 0 && is_eclipse(i) == 0
            TX_activation_night(i) = 1;
        end

        TX_DataVolume(y,i) = txs(y).data * dt(i);

        for r=1:length(sens_index(y,:))
            BUS_DataVolume_temp(r,i) = min(SENS_Buffer(sens_index(y,r),i-1),buses(bus_index).data *
dt(i) * (SENS_Buffer(y,i-1) > 0));
        end
        BUS_DataVolume2(y,i) = sum(BUS_DataVolume_temp(:,i));
        MEM_occupation(y,i) = max(0,MEM_occupation(y,i-1) + BUS_DataVolume2(y,i) -
TX_DataVolume(y,i)*TX_activation_night(1,i)*gs_flag_vect(txs(y).num_GS_selected,i));

        %---terminate the loop if buffer exceeds the one set from the user
        if MEM_occupation(y,i) > memories(y).data
            error_flag = 2;
            break
        end


    end


    %-----------------------------------------------------------
    %CALCULATE TOTAL DOWNLOADED DATA
    %-----------------------------------------------------------
    diffMEM(y,:)=diff(MEM_occupation(y,:));
    d(y) = abs(sum(diffMEM(diffMEM(y,:)<0)));
    d(y) = d(y) + sum(BUS_DataVolume2(y,diffMEM(y,:) == 0));
    d(y) = d(y) - sum(BUS_DataVolume2(y,
(TX_activation_night(1,i)*gs_flag_vect(txs(y).num_GS_selected,i))==1));
end

downlink_tot = sum(d);



%---MEMORIES PLOT
```

```
hold(axes3,'off')
col = hsv(num_mem);
for y = 1 : length(PlotMem)

plot(axes3,GMATfile.ElapsedDays,MEM_occupation(PlotMem(y),:),'color',col(y,:),'displayname',memorie
s(y).name)
    hold(axes3,'on')
end
title(axes3,'MEMORIES OCCUPATION','fontweight','bold')
xlabel(axes3,'Elapsed Days');
ylabel(axes3,'bits');
legend(axes3,'location','ne');
```

```
%---SAVE DATA
assignin('base','SENS_Buffer',SENS_Buffer);
assignin('base','MEM_occupation',MEM_occupation);
assignin('base','error_flag',error_flag);
assignin('base','downlink_tot',downlink_tot);
assignin('base','gen_tot',gen_tot);
assignin('base','data_rate_mean',data_rate_mean);
```

APPENDICE B

IL CODICE RELATIVO ALLA GUI

```
function varargout = Memory_Budget(varargin)
% MEMORY_BUDGET MATLAB code for Memory_Budget.fig
%      MEMORY_BUDGET, by itself, creates a new MEMORY_BUDGET or raises the existing
%      singleton*.
%
%      H = MEMORY_BUDGET returns the handle to a new MEMORY_BUDGET or the handle to
%      the existing singleton*.
%
%      MEMORY_BUDGET('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in MEMORY_BUDGET.M with the given input arguments.
%
%      MEMORY_BUDGET('Property','Value',...) creates a new MEMORY_BUDGET or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before Memory_Budget_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to Memory_Budget_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Memory_Budget

% Last Modified by GUIDE v2.5 23-Sep-2017 22:55:20

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
    'gui_Singleton',  gui_Singleton, ...
    'gui_OpeningFcn', @Memory_Budget_OpeningFcn, ...
    'gui_OutputFcn',  @Memory_Budget_OutputFcn, ...
    'gui_LayoutFcn',  [] , ...
    'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before Memory_Budget is made visible.
function Memory_Budget_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Memory_Budget (see VARARGIN)

% Choose default command line output for Memory_Budget
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);
WS_variables = evalin('base','who');


exist_s = ismember('s',WS_variables);
if exist_s == 1
s = evalin('base','s');
%SETTING THE FIELDS IN THE GUI

set(handles.ListSens,'String',{s.name});
set(handles.ListPlotSens,'String',{s.name});

set(handles.ListSens,'Value',1);
set(handles.EditSensName,'String',s(1).name);
set(handles.StaticDataRate,'String',s(1).data);
set(handles.pay_act_logic_edit,'String',s(1).activation_condition);
set(handles.StaticBuffer,'String',s(1).buffer);
set(handles.EditAmount,'String',s(1).EditAmount);
set(handles.EditEvery,'String',s(1).EditEvery);
set(handles.PopUpBit,'Value',s(1).PopUpBit);
set(handles.PopUpTime,'Value',s(1).PopUpTime);
set(handles.EditSensBuffer,'String',s(1).EditSensBuffer);
set(handles.PopUpBit5,'Value',s(1).PopUpBit5);

else
%     s = struct;
% assignin('base','s',s);
end

exist_b = ismember('b',WS_variables);
if exist_b == 1
b = evalin('base','b');
%SETTING THE FIELDS IN THE GUI

set(handles.ListBus,'String',{b.name});
set(handles.ListSensToBus,'String',{b.name});
set(handles.ListBusToMem,'String',{b.name});

set(handles.ListBus,'Value',1);
set(handles.EditBusName,'String',b(1).name);
set(handles.StaticDataRate2,'String',b(1).data);
set(handles.EditSpeed,'String',b(1).EditSpeed);
set(handles.PopUpBit2,'Value',b(1).PopUpBit2);
set(handles.StaticPlusOrTimes1,'String',b(1).StaticPlusOrTimes1);
set(handles.StaticPlusOrTimes2,'String',b(1).StaticPlusOrTimes2);
set(handles.EditOverhead,'String',b(1).EditOverhead);
set(handles.EditCompression,'String',b(1).EditCompression);

else
%     b = struct;
% assignin('base','b',b);
```

```
end

exist_m = ismember('m',WS_variables);
if exist_m == 1
m = evalin('base','m');
%SETTING THE FIELDS IN THE GUI

set(handles.ListMem,'String',{m.name});
set(handles.ListPlotMem,'String',{m.name});

set(handles.ListMem,'Value',1);
set(handles.EditMemName,'String',m(1).name);
set(handles.StaticCapacity,'String',m(1).data);
set(handles.EditCapacity,'String',m(1).EditCapacity);
set(handles.PopUpBit3,'Value',m(1).PopUpBit3);
set(handles.ListMemToTx,'String',{m.name});

else
%     m = struct;
% assignin('base','m',m);
end

exist_t = ismember('t',WS_variables);
if exist_t == 1
t = evalin('base','t');
%SETTING THE FIELDS IN THE GUI

set(handles.ListTx,'String',{t.name});
set(handles.StaticnGS,'String',{t.num_GS_selected});

set(handles.ListTx,'Value',1);
set(handles.EditTxName,'String',t(1).name);
set(handles.StaticDataRate3,'String',t(1).data);
set(handles.EditDataRateDownlink,'String',t(1).EditDataRateDownlink);
set(handles.PopUpBit4,'Value',t(1).PopUpBit4);
set(handles.StaticPlusOrTimes3,'String',t(1).StaticPlusOrTimes3);
set(handles.StaticPlusOrTimes4,'String',t(1).StaticPlusOrTimes4);
set(handles.EditOverhead2,'String',t(1).EditOverhead2);
set(handles.EditCompression2,'String',t(1).EditCompression2);

%MULTIPLE SELECTION FOR GS LIST

 for i = 1 : t(1).num_GS_selected
     set(handles.ListGS,'Value',t(1).index_GS_selected(i));
     hold on
 end

else
%     t = struct;
%     assignin('base','t',t);
end

exist_GMATout = exist ('GMATout');
if exist_GMATout == 1
GMATout = evalin('base','GMATout');
%SETTING THE FIELDS IN THE GUI

x = fieldnames(GMATout);
```

```
set(handles.ListGS,'String',x);
else
%     GMATout = struct;
% assignin('base','GMATout',GMATout);
end



set(gcf,'MenuBar','figure');
set(gcf,'ToolBar','Figure');


% UIWAIT makes Memory_Budget wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = Memory_Budget_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on selection change in ListTx.
function ListTx_Callback(hObject, eventdata, handles)
% hObject    handle to ListTx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
m = evalin('base','m');
b = evalin('base','b');
t = evalin('base','t');
index_selected = get(handles.ListTx,'Value');
struct_selected = t(index_selected);
set(handles.EditDataRateDownlink,'String',t(index_selected).EditDataRateDownlink);
set(handles.EditOverhead2,'String',t(index_selected).EditOverhead2);
set(handles.EditCompression2,'String',t(index_selected).EditCompression2);
set(handles.PopUpBit4,'Value',t(index_selected).PopUpBit4);
set(handles.StaticDataRate3,'String',struct_selected.data);
set(handles.EditTxName,'String',struct_selected.name);
%----update memories list
mem_index =
find(strncmp(t(index_selected).attached_to,{m.name},length(t(index_selected).attached_to)));
set(handles.ListMemToTx,'Value',mem_index);
%----update ground stations list
GSs = get(handles.ListGS,'String');
for gi=1:size(t(index_selected).GS,1)
    GS_index(gi) =
find(strncmp(strtrim(t(index_selected).GS(gi,:)),GSs,length(strtrim(t(index_selected).GS(gi,:)))));
end
set(handles.ListGS,'Value',GS_index);
set(handles.StaticPlusOrTimes3,'String',t(index_selected).StaticPlusOrTimes3);
set(handles.StaticPlusOrTimes4,'String',t(index_selected).StaticPlusOrTimes4);
```

```
% Hints: contents = cellstr(get(hObject,'String')) returns ListTx contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListTx


% --- Executes during object creation, after setting all properties.
function ListTx_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListTx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in ListMem.
function ListMem_Callback(hObject, eventdata, handles)
% hObject    handle to ListMem (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
s = evalin('base','s');
b = evalin('base','b');
m = evalin('base','m');
index_selected = get(handles.ListMem,'Value');
struct_selected = m(index_selected);
set(handles.EditCapacity,'String',(m(index_selected).EditCapacity));
set(handles.StaticCapacity,'String',(struct_selected.data));
set(handles.EditMemName,'String',(struct_selected.name));
set(handles.PopUpBit3,'Value',m(index_selected).PopUpBit3);
%----update bus list
bus_index =
find(strncmp(m(index_selected).attached_to_upstream,{b.name},length(m(index_selected).attached_to_u
pstream)));
set(handles.ListBusToMem,'Value',bus_index);


% Hints: contents = cellstr(get(hObject,'String')) returns ListMem contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListMem


% --- Executes during object creation, after setting all properties.
function ListMem_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListMem (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in ListSens.
```

```
function ListSens_Callback(hObject, eventdata, handles)
% hObject    handle to ListSens (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
s = evalin('base','s');
b = evalin('base','b');
index_selected = get(handles.ListSens,'Value');
struct_selected = s(index_selected);
set(handles.EditAmount,'String',s(index_selected).EditAmount);
set(handles.EditEvery,'String',s(index_selected).EditEvery);
set(handles.EditSensBuffer,'String',s(index_selected).EditSensBuffer);
set(handles.StaticDataRate,'String',struct_selected.data);
set(handles.StaticBuffer,'String',struct_selected.buffer);
set(handles.pay_act_logic_edit,'String',struct_selected.activation_condition);
set(handles.EditSensName,'String',{struct_selected.name});
set(handles.PopUpBit,'Value',s(index_selected).PopUpBit);
set(handles.PopUpTime,'Value',s(index_selected).PopUpTime);
set(handles.PopUpBit5,'Value',s(index_selected).PopUpBit5);
%----update bus list
bus_index =
find(strncmp(s(index_selected).attached_to,{b.name},length(s(index_selected).attached_to)));
set(handles.ListSensToBus,'Value',bus_index);

% --- Executes during object creation, after setting all properties.
function ListSens_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListSens (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditAmount_Callback(hObject, eventdata, handles)
% hObject    handle to EditAmount (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditAmount as text
%        str2double(get(hObject,'String')) returns contents of EditAmount as a double


% --- Executes during object creation, after setting all properties.
function EditAmount_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditAmount (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on selection change in listbox5.
function listbox5_Callback(hObject, eventdata, handles)
% hObject    handle to listbox5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns listbox5 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from listbox5


% --- Executes during object creation, after setting all properties.
function listbox5_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditEvery_Callback(hObject, eventdata, handles)
% hObject    handle to EditEvery (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditEvery as text
%        str2double(get(hObject,'String')) returns contents of EditEvery as a double


% --- Executes during object creation, after setting all properties.
function EditEvery_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditEvery (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in listbox6.
function listbox6_Callback(hObject, eventdata, handles)
% hObject    handle to listbox6 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns listbox6 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from listbox6


% --- Executes during object creation, after setting all properties.
```

```
function listbox6_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox6 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditActivationCond_Callback(hObject, eventdata, handles)
% hObject    handle to EditActivationCond (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditActivationCond as text
%        str2double(get(hObject,'String')) returns contents of EditActivationCond as a double


% --- Executes during object creation, after setting all properties.
function EditActivationCond_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditActivationCond (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in ListSensToBus.
function ListSensToBus_Callback(hObject, eventdata, handles)
% hObject    handle to ListSensToBus (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns ListSensToBus contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListSensToBus


% --- Executes during object creation, after setting all properties.
function ListSensToBus_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListSensToBus (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on selection change in ListBus.
function ListBus_Callback(hObject, eventdata, handles)
% hObject    handle to ListBus (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
%s = evalin('base','s');
b = evalin('base','b');
index_selected = get(handles.ListBus,'Value');
struct_selected = b(index_selected);
set(handles.EditSpeed,'String',b(index_selected).EditSpeed);
set(handles.EditOverhead,'String',b(index_selected).EditOverhead);
set(handles.EditCompression,'String',b(index_selected).EditCompression);
set(handles.StaticDataRate2,'String',struct_selected.data);
set(handles.EditBusName,'String',struct_selected.name);
set(handles.PopUpBit2,'Value',b(index_selected).PopUpBit2);
set(handles.StaticPlusOrTimes1,'String',b(index_selected).StaticPlusOrTimes1);
set(handles.StaticPlusOrTimes2,'String',b(index_selected).StaticPlusOrTimes2);

% Hints: contents = cellstr(get(hObject,'String')) returns ListBus contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListBus


% --- Executes during object creation, after setting all properties.
function ListBus_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListBus (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditSpeed_Callback(hObject, eventdata, handles)
% hObject    handle to EditSpeed (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditSpeed as text
%        str2double(get(hObject,'String')) returns contents of EditSpeed as a double


% --- Executes during object creation, after setting all properties.
function EditSpeed_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditSpeed (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on selection change in listbox8.
function listbox8_Callback(hObject, eventdata, handles)
% hObject    handle to listbox8 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns listbox8 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from listbox8


% --- Executes during object creation, after setting all properties.
function listbox8_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox8 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditDataRateDownlink_Callback(hObject, eventdata, handles)
% hObject    handle to EditDataRateDownlink (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditDataRateDownlink as text
%        str2double(get(hObject,'String')) returns contents of EditDataRateDownlink as a double


% --- Executes during object creation, after setting all properties.
function EditDataRateDownlink_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditDataRateDownlink (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditOverhead2_Callback(hObject, eventdata, handles)
% hObject    handle to EditOverhead2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditOverhead2 as text
%        str2double(get(hObject,'String')) returns contents of EditOverhead2 as a double


% --- Executes during object creation, after setting all properties.
function EditOverhead2_CreateFcn(hObject, eventdata, handles)
```

```
% hObject    handle to EditOverhead2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit11_Callback(hObject, eventdata, handles)
% hObject    handle to edit11 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit11 as text
%        str2double(get(hObject,'String')) returns contents of edit11 as a double


% --- Executes during object creation, after setting all properties.
function edit11_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit11 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in ListMemToTx.
function ListMemToTx_Callback(hObject, eventdata, handles)
% hObject    handle to ListMemToTx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns ListMemToTx contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListMemToTx


% --- Executes during object creation, after setting all properties.
function ListMemToTx_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListMemToTx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in ListBusToMem.
```

```
function ListBusToMem_Callback(hObject, eventdata, handles)
% hObject    handle to ListBusToMem (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns ListBusToMem contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListBusToMem


% --- Executes during object creation, after setting all properties.
function ListBusToMem_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListBusToMem (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditCapacity_Callback(hObject, eventdata, handles)
% hObject    handle to EditCapacity (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditCapacity as text
%        str2double(get(hObject,'String')) returns contents of EditCapacity as a double


% --- Executes during object creation, after setting all properties.
function EditCapacity_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditCapacity (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in listbox11.
function listbox11_Callback(hObject, eventdata, handles)
% hObject    handle to listbox11 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns listbox11 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from listbox11


% --- Executes during object creation, after setting all properties.
function listbox11_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox11 (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in listbox13.
function listbox13_Callback(hObject, eventdata, handles)
% hObject    handle to listbox13 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns listbox13 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from listbox13


% --- Executes during object creation, after setting all properties.
function listbox13_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox13 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditOverhead_Callback(hObject, eventdata, handles)
% hObject    handle to EditOverhead (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditOverhead as text
%        str2double(get(hObject,'String')) returns contents of EditOverhead as a double


% --- Executes during object creation, after setting all properties.
function EditOverhead_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditOverhead (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditCompression_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to EditCompression (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditCompression as text
%        str2double(get(hObject,'String')) returns contents of EditCompression as a double


% --- Executes during object creation, after setting all properties.
function EditCompression_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditCompression (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in PopUpBit2.
function PopUpBit_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpBit2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns PopUpBit2 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpBit2



% --- Executes during object creation, after setting all properties.
function PopUpBit_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PopUpBit2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in PopUpTime.
function PopUpTime_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpTime (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% Hints: contents = cellstr(get(hObject,'String')) returns PopUpTime contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpTime


% --- Executes during object creation, after setting all properties.
function PopUpTime_CreateFcn(hObject, eventdata, handles)
```

```
% hObject    handle to PopUpTime (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in PopUpBit3.
function PopUpBit3_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpBit3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns PopUpBit3 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpBit3


% --- Executes during object creation, after setting all properties.
function PopUpBit3_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PopUpBit3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in PopUpBit4.
function PopUpBit4_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpBit4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns PopUpBit4 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpBit4
x = get(handles.PopUpBit4,'Value');
if x <= 5
    set(handles.StaticBitOrByte3,'String','bits');
    set(handles.StaticBitOrByte4,'String','bits');
else
    set(handles.StaticBitOrByte3,'String','bytes');
    set(handles.StaticBitOrByte4,'String','bytes');
end

% --- Executes during object creation, after setting all properties.
function PopUpBit4_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PopUpBit4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
```

```
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in PopUpBit2.
function PopUpBit2_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpBit2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns PopUpBit2 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpBit2
x = get(handles.PopUpBit2,'Value');
if x <= 5
    set(handles.StaticBitOrByte1,'String','bits');
    set(handles.StaticBitOrByte2,'String','bits');
else
    set(handles.StaticBitOrByte1,'String','bytes');
    set(handles.StaticBitOrByte2,'String','bytes');
end


% --- Executes during object creation, after setting all properties.
function PopUpBit2_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PopUpBit2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in radiobutton2.
function radiobutton2_Callback(hObject, eventdata, handles)
% hObject    handle to radiobutton2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of radiobutton2


% --- Executes on button press in radiobutton1.
function radiobutton1_Callback(hObject, eventdata, handles)
% hObject    handle to radiobutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of radiobutton1


% --- Executes on button press in PushRemoveBus.
function PushRemoveBus_Callback(hObject, eventdata, handles)
% hObject    handle to PushRemoveBus (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
b = evalin('base','b');
deleted_index = get(handles.ListBus,'Value');
currentListBus = get(handles.ListBus,'String');
newListBus = currentListBus;
newListBus(deleted_index,:) = [];
set(handles.ListBus,'String',newListBus);
set(handles.ListSensToBus,'String',newListBus);
set(handles.ListBusToMem,'String',newListBus);
set(handles.ListBus,'Value',1);
set(handles.ListSensToBus,'Value',1);
set(handles.ListBusToMem,'Value',1);
assignin('base','b',b);


% --- Executes on button press in pushbutton3.
function pushbutton3_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%%%LOOK FOR MISSING FIELDS%%%
TF1 = isempty(get(handles.EditBusName,'String'));
TF2 = isempty(get(handles.StaticDataRate2,'String'));

if TF1 == 1
    set(handles.StaticMissingError2,'Visible','on');
    set(handles.StaticMissingError2,'String','Name is missing.');
elseif TF2 == 1

    set(handles.StaticMissingError2,'Visible','on');
    set(handles.StaticMissingError2,'String','Data rate is missing.');

else
    %%%CLEAR ERROR WINDOW%%%
    set(handles.StaticMissingError2,'Visible','off');

    %b = evalin('base','b');
    if strcmpi(get(handles.ListBus,'String'),'Add the first bus')
        b = struct;
        k = 1;
    else
        b = evalin('base','b');
        k = numel(b)+1;
    end
    b(k).name = char(get(handles.EditBusName,'String'));
    b(k).data = str2num(get(handles.StaticDataRate2,'String'));
    b(k).EditSpeed = str2num(get(handles.EditSpeed,'String'));
    b(k).PopUpBit2 = get(handles.PopUpBit2,'Value');
    b(k).StaticPlusOrTimes1 = get(handles.StaticPlusOrTimes1,'String');
    b(k).StaticPlusOrTimes2 = get(handles.StaticPlusOrTimes2,'String');
    b(k).EditOverhead = str2num(get(handles.EditOverhead,'String'));
    b(k).EditCompression = str2num(get(handles.EditCompression,'String'));
    assignin('base','b',b);
    set(handles.ListBus,'String',{b.name});
    set(handles.ListSensToBus,'String',{b.name});
    set(handles.ListBusToMem,'String',{b.name});
```

```
end


function EditBusName_Callback(hObject, eventdata, handles)
% hObject    handle to EditBusName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditBusName as text
%        str2double(get(hObject,'String')) returns contents of EditBusName as a double

% --- Executes during object creation, after setting all properties.
function EditBusName_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditBusName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditTxName_Callback(hObject, eventdata, handles)
% hObject    handle to EditTxName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditTxName as text
%        str2double(get(hObject,'String')) returns contents of EditTxName as a double


% --- Executes during object creation, after setting all properties.
function EditTxName_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditTxName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in PushAddMem.
function PushAddMem_Callback(hObject, eventdata, handles)
% hObject    handle to PushAddMem (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%%%LOOK FOR MISSING FIELDS%%%
TF1 = isempty(get(handles.EditMemName,'String'));
TF2 = isempty(get(handles.StaticCapacity,'String'));
TF3 = isempty(get(handles.ListBusToMem,'String'));
```

```
if TF1 == 1
    set(handles.StaticMissingError3,'Visible','on');
    set(handles.StaticMissingError3,'String','Name is missing.');
elseif TF2 == 1

    set(handles.StaticMissingError3,'Visible','on');
    set(handles.StaticMissingError3,'String','Capacity is missing.');
elseif TF3 == 1
    set(handles.StaticMissingError3,'Visible','on');
    set(handles.StaticMissingError3,'String','No upstream bus have been created');
else
    %%%CLEAR ERROR WINDOW%%%
    set(handles.StaticMissingError3,'Visible','off');
```

```
    if strcmpi(get(handles.ListMem,'String'),'Add the first memory')
        m = struct;
        k = 1;
    else
        m = evalin('base','m');
        k = numel(m)+1;
    end
    m(k).name = char(get(handles.EditMemName,'String'));
    m(k).data = str2num(get(handles.StaticCapacity,'String'));
    index_selected = get(handles.ListBusToMem,'Value');
    buses = get(handles.ListBusToMem,'String');
    bus_selected = buses(index_selected);
    m(k).attached_to_upstream = char(bus_selected);
    m(k).EditCapacity = str2num(get(handles.EditCapacity,'String'));
    m(k).PopUpBit3 = get(handles.PopUpBit3,'Value');
    bus_selected = buses(index_selected);
    assignin('base','m',m);
    set(handles.ListMem,'String',{m.name});
    set(handles.ListMemToTx,'String',{m.name});
    set(handles.ListPlotMem,'String',{m.name});
end


% --- Executes on button press in pushbutton6.
function pushbutton6_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton6 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
m = evalin('base','m');
deleted_index = get(handles.ListMem,'Value');
currentListMem = get(handles.ListMem,'String');
newListMem = currentListMem;
newListMem(deleted_index,:) = [];
m(deleted_index) = [];
set(handles.ListMem,'String',newListMem);
set(handles.ListMemToTx,'String',newListMem);
set(handles.ListPlotMem,'String',newListMem);
set(handles.ListMem,'Value',1);
set(handles.ListPlotMem,'Value',1);
set(handles.ListMemToTx,'Value',1);
assignin('base','m',m);
```

```
function EditMemName_Callback(hObject, eventdata, handles)
% hObject    handle to EditMemName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditMemName as text
%        str2double(get(hObject,'String')) returns contents of EditMemName as a double


% --- Executes during object creation, after setting all properties.
function EditMemName_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditMemName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in PushAddSens.
function PushAddSens_Callback(hObject, eventdata, handles)
% hObject    handle to PushAddSens (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%%%LOOK FOR MISSING FIELDS%%%
TF1 = isempty(get(handles.EditSensName,'String'));
TF2 = isempty(get(handles.StaticDataRate,'String'));
TF3 = isempty(get(handles.pay_act_logic_edit,'String'));
TF4 = isempty(get(handles.EditSensBuffer,'String'));
TF5 = isempty(get(handles.ListSensToBus,'String'));
if TF1 == 1
    set(handles.StaticMissingError1,'Visible','on');
    set(handles.StaticMissingError1,'String','Name is missing.');
elseif TF2 == 1

    set(handles.StaticMissingError1,'Visible','on');
    set(handles.StaticMissingError1,'String','Data rate is missing.');
elseif TF3 == 1

    set(handles.StaticMissingError1,'Visible','on');
    set(handles.StaticMissingError1,'String','Activation logic is missing.');
elseif TF4 == 1

    set(handles.StaticMissingError1,'Visible','on');
    set(handles.StaticMissingError1,'String','Sensor buffer is missing');
elseif TF5 == 1
    set(handles.StaticMissingError1,'Visible','on');
    set(handles.StaticMissingError1,'String','No bus have been created');
else
    %%%CLEAR ERROR WINDOW%%%
    set(handles.StaticMissingError1,'Visible','off');

    %-----------------------------------
```

```
    if strcmpi(get(handles.ListSens,'String'),'Add the first sensor')
        s = struct;
        k = 1;
    else
        s = evalin('base','s');
        k = numel(s)+1;
    end
    s(k).name = char(get(handles.EditSensName,'String'));
    s(k).data = str2num(get(handles.StaticDataRate,'String'));
    s(k).activation_condition = char(get(handles.pay_act_logic_edit,'String'));
    index_selected = get(handles.ListSensToBus,'Value');
    buses = get(handles.ListSensToBus,'String');
    bus_selected = buses(index_selected);
    s(k).attached_to = char(bus_selected);
    s(k).buffer = str2num(get(handles.StaticBuffer,'String'));
    s(k).EditAmount = str2num(get(handles.EditAmount,'String'));
    s(k).EditEvery = str2num(get(handles.EditEvery,'String'));
    s(k).PopUpBit = get(handles.PopUpBit,'Value');
    s(k).PopUpTime = get(handles.PopUpTime,'Value');
    s(k).EditSensBuffer = str2num(get(handles.EditSensBuffer,'String'));
    s(k).PopUpBit5 = get(handles.PopUpBit5,'Value');

    assignin('base','s',s);
    set(handles.ListSens,'String',{s.name});
    set(handles.ListPlotSens,'String',{s.name});

end



% --- Executes on button press in PushRemoveSens.
function PushRemoveSens_Callback(hObject, eventdata, handles)
% hObject    handle to PushRemoveSens (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
s = evalin('base','s');
deleted_index = get(handles.ListSens,'Value');
currentListSens = get(handles.ListSens,'String');
newListSens = currentListSens;
newListSens(deleted_index,:) = [];
s(deleted_index) = [];
set(handles.ListSens,'String',newListSens);
set(handles.ListPlotSens,'String',newListSens);
set(handles.ListSens,'Value',1);
set(handles.ListPlotSens,'Value',1);
assignin('base','s',s);




function EditSensName_Callback(hObject, eventdata, handles)

% hObject    handle to EditSensName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of EditSensName as text
%        str2double(get(hObject,'String')) returns contents of EditSensName as a double


% --- Executes during object creation, after setting all properties.
function EditSensName_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditSensName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in PushGetDataRate.
function PushGetDataRate_Callback(hObject, eventdata, handles)
% hObject    handle to PushGetDataRate (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%conversione della stringa nelle Edit Text in numero
str_amount = char(get(handles.EditAmount,'String'));
str_every = get(handles.EditEvery,'String');
amount = str2num(str_amount);
every = str2num(str_every);
%conversione in bit dei suoi multipli:
x = get(handles.PopUpBit,'Value');
if x == 1
    amount_new = amount * 1;
elseif x == 2
    amount_new = amount * 1e3;
elseif x == 3
    amount_new = amount * 1e6;
elseif x == 4
    amount_new = amount * 1e9;
elseif x == 5
    amount_new = amount * 1e12;
elseif x == 6
    amount_new = amount * 8;
elseif x == 7
    amount_new = amount * 8e3;
elseif x == 8
    amount_new = amount * 8e6;
elseif x == 9
    amount_new = amount * 8e9;
elseif x == 10
    amount_new = amount * 8e12;

end
%conversione in secondi dei suoi multipli:
y = get(handles.PopUpTime,'Value');
if y == 1
    every_new = every;
elseif y == 2
    every_new = every * 60;
```

```
elseif y == 3
    every_new = every * 3600;
elseif y == 4
    every_new = every * 86400;
end
%calcolo del Data Rate Medio%
data_rate = amount_new / every_new ;
set(handles.StaticDataRate,'String',data_rate);




% --- Executes during object creation, after setting all properties.
function uipanel11_CreateFcn(hObject, eventdata, handles)
% hObject    handle to uipanel11 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
x = get(handles.RadioBit,'Value');
PopUpBit_New_Str = {'byte';'kbyte';'Mbyte';'Gbyte';'Tbyte'};
PopUpBit_Old_Str = {'bit';'kbit';'Mbit';'Gbit';'Tbit'};
if x == 0
    set(handles.PopUpBit,'String',PopUpBit_New_Str);
else
    set(handles.PopUpBit,'String',PopUpBit_Old_Str);
end




% --- Executes on button press in PushOK1.
function PushOK1_Callback(hObject, eventdata, handles)
% hObject    handle to PushOK1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
x = get(handles.RadioBit,'Value');
PopUpBit_New_Str = {'byte';'kbyte';'Mbyte';'Gbyte';'Tbyte'};
PopUpBit_Old_Str = {'bit';'kbit';'Mbit';'Gbit';'Tbit'};
if x == 0
    set(handles.PopUpBit,'String',PopUpBit_New_Str);
else
    set(handles.PopUpBit,'String',PopUpBit_Old_Str);
end


% --- Executes on button press in PushOK2.
function PushOK2_Callback(hObject, eventdata, handles)
% hObject    handle to PushOK2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
x = get(handles.RadioBit2,'Value');
PopUpBit_New_Str = {'byte per sec';'kbyte per sec';'Mbyte per sec';'Gbyte per sec';'Tbyte per
sec'};
PopUpBit_Old_Str = {'bit per sec';'kbit per sec';'Mbit per sec';'Gbit per sec';'Tbit per sec'};
if x == 0
    set(handles.PopUpBit2,'String',PopUpBit_New_Str);
    set(handles.StaticBitOrByte1,'String','bytes');
```

```
    set(handles.StaticBitOrByte2,'String','bytes');
else
    set(handles.PopUpBit2,'String',PopUpBit_Old_Str);
    set(handles.StaticBitOrByte1,'String','bits');
    set(handles.StaticBitOrByte2,'String','bits');
end


% --- Executes on button press in PushOK3.
function PushOK3_Callback(hObject, eventdata, handles)
% hObject    handle to PushOK3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
x = get(handles.RadioPlus1,'Value');
Times = 'x';
Plus = '+';
if x == 0
    set(handles.StaticPlusOrTimes1,'String',Times);
else
    set(handles.StaticPlusOrTimes1,'String',Plus);
end



% --- Executes on button press in PushOK4.
function PushOK4_Callback(hObject, eventdata, handles)
% hObject    handle to PushOK4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%%%Choose between plus or times operation in overhead%%%
x = get(handles.RadioPlus1,'Value');
Times = 'x';
Plus = '+';
if x == 0
    set(handles.StaticPlusOrTimes1,'String',Times);
else
    set(handles.StaticPlusOrTimes1,'String',Plus);
end

%%%Choose between plus or times operation in compression%%%
x = get(handles.RadioPlus2,'Value');
Times = 'x';
Plus = '-';
if x == 0
    set(handles.StaticPlusOrTimes2,'String',Times);
else
    set(handles.StaticPlusOrTimes2,'String',Plus);
end


% --- Executes on button press in PushGetDataRate2.
function PushGetDataRate2_Callback(hObject, eventdata, handles)
% hObject    handle to PushGetDataRate2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
str_speed = get(handles.EditSpeed,'String');
speed = str2num(str_speed);
```

```
%conversione in bit dei suoi multipli:
x = get(handles.PopUpBit2,'Value');
if x == 1
    speed_new = speed * 1;
elseif x == 2
    speed_new = speed * 1e3;
elseif x == 3
    speed_new = speed * 1e6;
elseif x == 4
    speed_new = speed * 1e9;
elseif x == 5
    speed_new = speed * 1e12;
elseif x == 6
    speed_new = speed * 8;
elseif x == 7
    speed_new = speed * 8e3;
elseif x == 8
    speed_new = speed * 8e6;
elseif x == 9
    speed_new = speed * 8e9;
elseif x == 10
    speed_new = speed * 8e12;
end
%considering overhead
PlusOrTimes1 = get(handles.StaticPlusOrTimes1,'String');
EditOverhead_str = get(handles.EditOverhead,'String');
EditOverhead = str2num(EditOverhead_str);
if PlusOrTimes1 == 'x'
    speed_new = speed_new * EditOverhead;
else speed_new = speed_new + EditOverhead;
end
%considering compression
PlusOrTimes2 = get(handles.StaticPlusOrTimes2,'String');
EditCompression_str = get(handles.EditCompression,'String');
EditCompression = str2num(EditCompression_str);
if PlusOrTimes2 == 'x'
    speed_new = speed_new * EditCompression;
else speed_new = speed_new - EditCompression;
end
set(handles.StaticDataRate2,'String',speed_new);


% --- Executes on button press in PushOK5.
function PushOK5_Callback(hObject, eventdata, handles)
% hObject    handle to PushOK5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
x = get(handles.RadioBit3,'Value');
PopUpBit_New_Str = {'byte';'kbyte';'Mbyte';'Gbyte';'Tbyte'};
PopUpBit_Old_Str = {'bit';'kbit';'Mbit';'Gbit';'Tbit'};
if x == 0
    set(handles.PopUpBit3,'String',PopUpBit_New_Str);
else
    set(handles.PopUpBit3,'String',PopUpBit_Old_Str);
end
```

```
% --- Executes on button press in PushGetCapacity.
function PushGetCapacity_Callback(hObject, eventdata, handles)
% hObject    handle to PushGetCapacity (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
str_capacity = get(handles.EditCapacity,'String');
capacity = str2num(str_capacity);

%conversione in bit dei suoi multipli:
x = get(handles.PopUpBit3,'Value');
if x == 1
    capacity_new = capacity * 1;
elseif x == 2
    capacity_new = capacity * 1e3;
elseif x == 3
    capacity_new = capacity * 1e6;
elseif x == 4
    capacity_new = capacity * 1e9;
elseif x == 5
    capacity_new = capacity * 1e12;
elseif x == 6
    capacity_new = capacity * 8;
elseif x == 7
    capacity_new = capacity * 8e3;
elseif x == 8
    capacity_new = capacity * 8e6;
elseif x == 9
    capacity_new = capacity * 8e9;
elseif x == 10
    capacity_new = capacity * 8e12;
end
set(handles.StaticCapacity,'String',capacity_new);

% --- Executes on button press in PushAddTx.
function PushAddTx_Callback(hObject, eventdata, handles)
% hObject    handle to PushAddTx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

%%%LOOK FOR MISSING FIELDS%%%
TF1 = isempty(get(handles.EditTxName,'String'));
TF2 = isempty(get(handles.StaticDataRate3,'String'));
TF3 = isempty(get(handles.ListMemToTx,'String'));
TF4 = isempty(get(handles.ListGS,'String'));

if TF1 == 1
    set(handles.StaticMissingError4,'Visible','on');
    set(handles.StaticMissingError4,'String','Name is missing.');
elseif TF2 == 1

    set(handles.StaticMissingError4,'Visible','on');
    set(handles.StaticMissingError4,'String','Data rate is missing.');
elseif TF3 == 1
    set(handles.StaticMissingError4,'Visible','on');
    set(handles.StaticMissingError4,'String','No bus have been created.');
elseif TF4 == 1
    set(handles.StaticMissingError4,'Visible','on');
    set(handles.StaticMissingError4,'String','No ground station have been created');
```

```
else
    %%%CLEAR ERROR WINDOW%%%
    set(handles.StaticMissingError4,'Visible','off');

    if strcmpi(get(handles.ListTx,'String'),'Add the first Tx')
        t = struct;
        k = 1;
    else
        t = evalin('base','t');
        k = numel(t)+1;
    end
    t(k).name = char(get(handles.EditTxName,'String'));
    t(k).data = str2num(get(handles.StaticDataRate3,'String'));
    index_selected = get(handles.ListMem,'Value');
    memories = get(handles.ListMem,'String');
    memory_selected = memories(index_selected);
    t(k).attached_to = char(memory_selected);
    t(k).night = get(handles.RadioNightYES,'Value');
    index_GS_selected = get(handles.ListGS,'Value');
    GSs = get(handles.ListGS,'String');
    GS_selected = GSs(index_GS_selected);
    t(k).GS = char(GS_selected);
    num_GS_selected = length(index_GS_selected);
    t(k).index_GS_selected = index_GS_selected;
    t(k).num_GS_selected = num_GS_selected;
    set(handles.StaticnGS,'String',num_GS_selected);
    set(handles.ListTx,'String',{t.name});
    t(k).EditDataRateDownlink = str2num(get(handles.EditDataRateDownlink,'String'));
    t(k).PopUpBit4 = get(handles.PopUpBit4,'Value');
    t(k).StaticPlusOrTimes3 = get(handles.StaticPlusOrTimes3,'String');
    t(k).StaticPlusOrTimes4 = get(handles.StaticPlusOrTimes4,'String');
    t(k).EditOverhead2 = str2num(get(handles.EditOverhead2,'String'));
    t(k).EditCompression2 = str2num(get(handles.EditCompression2,'String'));
    assignin('base','t',t);
    assignin('base','num_GS_selected',num_GS_selected);

end


% --- Executes on button press in PushRemoveTx.
function PushRemoveTx_Callback(hObject, eventdata, handles)
% hObject    handle to PushRemoveTx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
t = evalin('base','t');
deleted_index = get(handles.ListTx,'Value');
currentListTx = get(handles.ListTx,'String');
newListTx = currentListTx;
newListTx(deleted_index,:) = [];
t(deleted_index) = [];
set(handles.ListTx,'String',newListTx);
set(handles.ListTx,'Value',1);
assignin('base','t',t)

% --- Executes on button press in PushOK7.
function PushOK7_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to PushOK7 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
%%%Choose between plus or times operation in overhead%%%
x = get(handles.RadioPlus3,'Value');
Times = 'x';
Plus = '+';
if x == 0
    set(handles.StaticPlusOrTimes3,'String',Times);
else
    set(handles.StaticPlusOrTimes3,'String',Plus);
end

%%%Choose between plus or times operation in compression%%%
x = get(handles.RadioPlus4,'Value');
Times = 'x';
Plus = '-';
if x == 0
    set(handles.StaticPlusOrTimes4,'String',Times);
else
    set(handles.StaticPlusOrTimes4,'String',Plus);
end


% --- Executes on button press in PushOK6.
function PushOK6_Callback(hObject, eventdata, handles)
% hObject    handle to PushOK6 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
x = get(handles.RadioBit4,'Value');
PopUpBit_New_Str = {'byte per sec';'kbyte per sec';'Mbyte per sec';'Gbyte per sec';'Tbyte per
sec'};
PopUpBit_Old_Str = {'bit per sec';'kbit per sec';'Mbit per sec';'Gbit per sec';'Tbit per sec'};
if x == 0
    set(handles.PopUpBit4,'String',PopUpBit_New_Str);
    set(handles.StaticBitOrByte3,'String','bytes');
else
    set(handles.PopUpBit4,'String',PopUpBit_Old_Str);
    set(handles.StaticBitOrByte3,'String','bits');
end


% --- Executes on button press in PushGetDataRate3.
function PushGetDataRate3_Callback(hObject, eventdata, handles)
% hObject    handle to PushGetDataRate3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
str_speed = get(handles.EditDataRateDownlink,'String');
speed = str2num(str_speed);

%conversione in bit dei suoi multipli:
x = get(handles.PopUpBit4,'Value');
if x == 1
    speed_new = speed * 1;
elseif x == 2
    speed_new = speed * 1e3;
elseif x == 3
```

```
    speed_new = speed * 1e6;
elseif x == 4
    speed_new = speed * 1e9;
elseif x == 5
    speed_new = speed * 1e12;
elseif x == 6
    speed_new = speed * 8;
elseif x == 7
    speed_new = speed * 8e3;
elseif x == 8
    speed_new = speed * 8e6;
elseif x == 9
    speed_new = speed * 8e9;
elseif x == 10
    speed_new = speed * 8e12;
end
if x <= 5
    set(handles.StaticBitOrByte3,'String','bits');
    set(handles.StaticBitOrByte4,'String','bits');
else
    set(handles.StaticBitOrByte3,'String','bytes');
    set(handles.StaticBitOrByte4,'String','bytes');
end
%considering overhead
PlusOrTimes = get(handles.StaticPlusOrTimes3,'String');
EditOverhead_str = get(handles.EditOverhead2,'String');
EditOverhead = str2num(EditOverhead_str);

if PlusOrTimes == 'x'
    speed_new = speed_new * EditOverhead;
else speed_new = speed_new + EditOverhead;
end

%considering compression
PlusOrTimes = get(handles.StaticPlusOrTimes4,'String');
EditCompression_str = get(handles.EditCompression2,'String');
EditCompression = str2num(EditCompression_str);

if PlusOrTimes == 'x'
    speed_new = speed_new * EditCompression;
else speed_new = speed_new - EditCompression;
end

set(handles.StaticDataRate3,'String',speed_new);



function EditFileName_Callback(hObject, eventdata, handles)
% hObject    handle to EditFileName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditFileName as text
%        str2double(get(hObject,'String')) returns contents of EditFileName as a double


% --- Executes during object creation, after setting all properties.
```

```
function EditFileName_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditFileName (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in PushBrowse.
function PushBrowse_Callback(hObject, eventdata, handles)
% hObject    handle to PushBrowse (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[filename pathname] = uigetfile({'*txt*'},'Select a file:');
fullpathname = strcat(pathname, filename);
set(handles.EditFileName, 'String', fullpathname);
handles.fullpathname = fullpathname;
guidata(hObject,handles);


% --- Executes on button press in PushImport.
function PushImport_Callback(hObject, eventdata, handles)
% hObject    handle to PushImport (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
fullpathname = handles.fullpathname;
%file = fileread(fullpathname);
%GMATout = evalin('base','GMATout');
GMATout = plot_GMAT_results(fullpathname,0);
x = fieldnames(GMATout);
L = length(GMATout.UTCGregorian);
set(handles.StaticTo,'String',L);
dt_vector = diff(GMATout.ElapsedDays);
dt_sec = dt_vector * 86400;
set(handles.Listdt,'String',unique(round(dt_sec*1e3)/1e3));
set(handles.ListGS,'String',x);
%x(strncmpi(x,'UTCGregorian',12))=[];
%set(handles.PopUpCondition,'String',x')
assignin('base','GMATout',GMATout);


% --- Executes on selection change in PopUpCondition.
function PopUpCondition_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpCondition (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns PopUpCondition contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpCondition


% --- Executes during object creation, after setting all properties.
function PopUpCondition_CreateFcn(hObject, eventdata, handles)
```

```
% hObject    handle to PopUpCondition (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditValue_Callback(hObject, eventdata, handles)
% hObject    handle to EditValue (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditValue as text
%        str2double(get(hObject,'String')) returns contents of EditValue as a double


% --- Executes during object creation, after setting all properties.
function EditValue_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditValue (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in PushArrow.
function PushArrow_Callback(hObject, eventdata, handles)
% hObject    handle to PushArrow (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
GMATout = evalin('base','GMATout');
index_selected = get(handles.PopUpCondition,'Value');
list = get(handles.PopUpCondition,'String');
list_selected = list(index_selected,:);
str = char(list_selected);
set(handles.PopUpValue,'String',GMATout.(str));



% --- Executes on selection change in PopUpValue.
function PopUpValue_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpValue (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns PopUpValue contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpValue
```

```
% --- Executes during object creation, after setting all properties.
function PopUpValue_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PopUpValue (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit20_Callback(hObject, eventdata, handles)
% hObject    handle to edit20 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit20 as text
%        str2double(get(hObject,'String')) returns contents of edit20 as a double


% --- Executes during object creation, after setting all properties.
function edit20_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit20 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit21_Callback(hObject, eventdata, handles)
% hObject    handle to edit21 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit21 as text
%        str2double(get(hObject,'String')) returns contents of edit21 as a double


% --- Executes during object creation, after setting all properties.
function edit21_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit21 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
```

```
end


% --- Executes on selection change in PopUpStartTime.
function PopUpStartTime_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpStartTime (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns PopUpStartTime contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpStartTime


% --- Executes during object creation, after setting all properties.
function PopUpStartTime_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PopUpStartTime (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in PopUpCompletionTime.
function PopUpCompletionTime_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpCompletionTime (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns PopUpCompletionTime contents as cell
array
%        contents{get(hObject,'Value')} returns selected item from PopUpCompletionTime


% --- Executes during object creation, after setting all properties.
function PopUpCompletionTime_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PopUpCompletionTime (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in PushOK8.
function PushOK8_Callback(hObject, eventdata, handles)
% hObject    handle to PushOK8 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
StartTime = str2num(get(handles.EditFrom,'String'));
CompletionTime = str2num(get(handles.EditTo,'String'));
```

```
GMATout = evalin('base','GMATout');
set(handles.StaticStartTime,'String',GMATout.UTCGregorian(StartTime));
set(handles.StaticCompletionTime,'String',GMATout.UTCGregorian(CompletionTime));

%GMATout = GMATout(StartTime:CompletionTime);
x = fieldnames(GMATout);
L = length(x);
for j=1:L
    FieldName(j) = {char(x(j))};
end
for i=1:L
    GMATout.(FieldName(i,:)) = GMATout.(FieldName(i,:))(StartTime:CompletionTime);
end

assignin('base','GMATout',GMATout);


% --- Executes on button press in PushRUN.
function PushRUN_Callback(hObject, eventdata, handles)
% hObject    handle to PushRUN (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
try
    hold off
    set(handles.StaticRunError,'Visible','off');
    set(handles.StaticSimulationCompleted,'Visible','off');
    set(handles.StaticBusy,'Visible','on');
    set(handles.error_text_edit,'Visible','off');
    set(handles.StaticMissingError1,'Visible','off');
    set(handles.StaticMissingError2,'Visible','off');
    set(handles.StaticMissingError3,'Visible','off');
    set(handles.StaticMissingError4,'Visible','off');
    t_start = str2num(get(handles.EditFrom,'String'));
    t_end = str2num(get(handles.EditTo,'String'));
    hold(handles.axes3,'off');
    s = evalin('base','s');
    b = evalin('base','b');
    m = evalin('base','m');
    t = evalin('base','t');
    GMATout = evalin('base','GMATout');
    num_GS_selected = t.num_GS_selected;
    PlotSens = get(handles.ListPlotSens,'Value');
    PlotMem = get(handles.ListPlotMem,'Value');
    [SENS_Buffer,memories,data_rate_mean,downlink_tot,gen_tot,error_flag] =
mem_bud_run(s,b,m,t,GMATout,t_start,t_end,handles,PlotSens,PlotMem);
    assignin('base','SENS_Buffer',SENS_Buffer);
    assignin('base','memories',memories);
    assignin('base','data_rate_mean',data_rate_mean);
    assignin('base','downlink_tot',downlink_tot);
    assignin('base','gen_tot',gen_tot);
    assignin('base','error_flag',error_flag);
    %---editing static texts for total generated data
    for k = 1 : size(gen_tot,1)
        gen_sens_max(k) = max(gen_tot(k,:));
    end

%     for g = 1 : size(downlink_tot,1)
```

```
%         downlink_tot_max(g) = max(downlink_tot(g,:));
%     end
    gen_tot_max = sum(gen_sens_max);
%     downlink_row_max = sum(downlink_tot');
%     downlink_tot_max = sum(downlink_row_max);
downlink_tot_max = sum(downlink_tot,2);
    if gen_tot_max < 1e3
        set(handles.StaticTotGen,'String',gen_tot_max);
        set(handles.StaticTotGenUnit,'String','bits');
    elseif (1e3 <= gen_tot_max) && logical(gen_tot_max < 1e6)
        set(handles.StaticTotGen,'String',gen_tot_max/1e3);
        set(handles.StaticTotGenUnit,'String','kbits');
    elseif (1e6 <= gen_tot_max) && logical(gen_tot_max < 1e9)
        set(handles.StaticTotGen,'String',gen_tot_max/1e6);
        set(handles.StaticTotGenUnit,'String','Mbits');
    elseif (1e9 <= gen_tot_max) && logical(gen_tot_max < 1e12)
        set(handles.StaticTotGen,'String',gen_tot_max/1e9);
        set(handles.StaticTotGenUnit,'String','Gbits');
    elseif gen_tot_max >= 1e12
        set(handles.StaticTotGen,'String',gen_tot_max/1e12);
        set(handles.StaticTotGenUnit,'String','Tbits');
    end
    %---editing static texts for total downloaded data
    if downlink_tot_max < 1e3
        set(handles.StaticTotDown,'String',downlink_tot_max);
        set(handles.StaticTotDownUnit,'String','bits');
    elseif (1e3 <= downlink_tot_max) & (downlink_tot_max < 1e6)
        set(handles.StaticTotDown,'String',downlink_tot_max/1e3);
        set(handles.StaticTotDownUnit,'String','kbits');
    elseif (1e6 <= downlink_tot_max) & (downlink_tot_max < 1e9)
        set(handles.StaticTotDown,'String',downlink_tot_max/1e6);
        set(handles.StaticTotDownUnit,'String','Mbits');
    elseif (1e9 <= downlink_tot_max) & (downlink_tot_max < 1e12)
        set(handles.StaticTotDown,'String',downlink_tot_max/1e9);
        set(handles.StaticTotDownUnit,'String','Gbits');
    elseif downlink_tot_max >= 1e12
        set(handles.StaticTotDown,'String',downlink_tot_max/1e12);
        set(handles.StaticTotDownUnit,'String','Tbits');
    end

    %---showing error messages
    if error_flag == 1
        set(handles.StaticRunError,'Visible','on');
        set(handles.StaticRunError,'String','A sensor buffer exceeds the defined value');
    elseif error_flag == 2
        set(handles.StaticRunError,'Visible','on');
        set(handles.StaticRunError,'String','A memory capacity exceeds the defined value');
    end

    set(handles.StaticSimulationCompleted,'Visible','on');
    set(handles.StaticBusy,'Visible','off');

catch me
    errmess=getReport(me,'extended','hyperlinks','off');
    isnewline=findstr(char(10),errmess);
    errmesscell{1}=errmess(1:isnewline(1)-1);
    for i=2:numel(isnewline)-1
```

```
        errmesscell{i}=errmess(isnewline(i)+1:isnewline(i+1)-1);
    end
    errmesscell{end+1}=errmess(isnewline(end)+1:numel(errmess));
    set(handles.error_text_edit,'String',...
        errmesscell,'Visible','on')
    assignin('base','me',me);
end









% --- Executes on selection change in popupmenu13.
function popupmenu13_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu13 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu13 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from popupmenu13


% --- Executes during object creation, after setting all properties.
function popupmenu13_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu13 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in popupmenu14.
function popupmenu14_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu14 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu14 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from popupmenu14


% --- Executes during object creation, after setting all properties.
function popupmenu14_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu14 (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in popupmenu16.
function popupmenu16_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu16 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu16 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from popupmenu16


% --- Executes during object creation, after setting all properties.
function popupmenu16_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu16 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pushbutton25.
function pushbutton25_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton25 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on selection change in popupmenu17.
function popupmenu17_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu17 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu17 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from popupmenu17


% --- Executes during object creation, after setting all properties.
function popupmenu17_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu17 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
```

```
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in ListGS.
function ListGS_Callback(hObject, eventdata, handles)
% hObject    handle to ListGS (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% Hints: contents = cellstr(get(hObject,'String')) returns ListGS contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListGS


% --- Executes during object creation, after setting all properties.
function ListGS_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListGS (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function Editdt_Callback(hObject, eventdata, handles)
% hObject    handle to Editdt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of Editdt as text
%        str2double(get(hObject,'String')) returns contents of Editdt as a double


% --- Executes during object creation, after setting all properties.
function Editdt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to Editdt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit23_Callback(hObject, eventdata, handles)
% hObject    handle to edit23 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit23 as text
%        str2double(get(hObject,'String')) returns contents of edit23 as a double


% --- Executes during object creation, after setting all properties.
function edit23_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit23 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in popupmenu19.
function popupmenu19_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu19 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu19 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from popupmenu19


% --- Executes during object creation, after setting all properties.
function popupmenu19_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu19 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit24_Callback(hObject, eventdata, handles)
% hObject    handle to edit24 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit24 as text
%        str2double(get(hObject,'String')) returns contents of edit24 as a double


% --- Executes during object creation, after setting all properties.
function edit24_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit24 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
```

```
% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in ListMemToBus.
function ListMemToBus_Callback(hObject, eventdata, handles)
% hObject    handle to ListMemToBus (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns ListMemToBus contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListMemToBus


% --- Executes during object creation, after setting all properties.
function ListMemToBus_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListMemToBus (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function pay_act_logic_edit_Callback(hObject, eventdata, handles)
% hObject    handle to pay_act_logic_edit (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of pay_act_logic_edit as text
%        str2double(get(hObject,'String')) returns contents of pay_act_logic_edit as a double


% --- Executes during object creation, after setting all properties.
function pay_act_logic_edit_CreateFcn(hObject, eventdata, handles)
% hObject    handle to pay_act_logic_edit (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pushbutton26.
function pushbutton26_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton26 (see GCBO)
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
ACT_LOGIC(handles.pay_act_logic_edit)


% --- Executes during object creation, after setting all properties.
function uipanel15_CreateFcn(hObject, eventdata, handles)
% hObject    handle to uipanel15 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
%%%Choose between plus or times operation in overhead%%%



% --- Executes on button press in pushbutton28.
function pushbutton28_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton28 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
b = evalin('base','b');
index_selected = get(handles.ListBus,'Value');
b(index_selected).name = char(get(handles.EditBusName,'String'));
b(index_selected).data = str2num(get(handles.StaticDataRate2,'String'));
b(index_selected).EditSpeed = str2num(get(handles.EditSpeed,'String'));
b(index_selected).PopUpBit2 = get(handles.PopUpBit2,'Value');
b(index_selected).StaticPlusOrTimes1 = get(handles.StaticPlusOrTimes1,'String');
b(index_selected).StaticPlusOrTimes2 = get(handles.StaticPlusOrTimes2,'String');
b(index_selected).EditOverhead = str2num(get(handles.EditOverhead,'String'));
b(index_selected).EditCompression = str2num(get(handles.EditCompression,'String'));
list = get(handles.ListBus,'String');
list(index_selected) = {b(index_selected).name};
set(handles.ListBus,'String',list);
set(handles.ListSensToBus,'String',list);
set(handles.ListBusToMem,'String',list);
assignin('base','b',b);

% --- Executes on button press in pushbutton30.
function pushbutton30_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton30 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
t = evalin('base','t');
index_selected = get(handles.ListTx,'Value');
t(index_selected).name = char(get(handles.EditTxName,'String'));
t(index_selected).data = str2num(get(handles.StaticDataRate3,'String'));
memories = get(handles.ListMemToTx,'String');
index_memory_selected = get(handles.ListMemToTx,'Value');
memory_selected = memories(index_memory_selected);
t(index_selected).attached_to = char(memory_selected);
list= get(handles.ListTx,'String');
list(index_selected) = {t(index_selected).name};
set(handles.ListTx,'String',list);
index_selected2 = get(handles.ListGS,'Value');
GSs = get(handles.ListGS,'String');
GS_selected = GSs(index_selected2);
t(index_selected).GS = char(GS_selected);
t(index_selected).night = get(handles.RadioNightYES,'Value');
```

```
t(index_selected).index_GS_selected = get(handles.ListGS,'Value');
t(index_selected).num_GS_selected = length(t(index_selected).index_GS_selected);
t(index_selected).EditDataRateDownlink = str2num(get(handles.EditDataRateDownlink,'String'));
t(index_selected).PopUpBit4 = get(handles.PopUpBit4,'Value');
t(index_selected).StaticPlusOrTimes3 = get(handles.StaticPlusOrTimes3,'String');
t(index_selected).StaticPlusOrTimes4 = get(handles.StaticPlusOrTimes4,'String');
t(index_selected).EditOverhead2 = str2num(get(handles.EditOverhead2,'String'));
t(index_selected).EditCompression2 = str2num(get(handles.EditCompression2,'String'));
set(handles.StaticnGS,'String',t(index_selected).num_GS_selected);
assignin('base','t',t);


% --- Executes on button press in pushbutton29.
function pushbutton29_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton29 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
m = evalin('base','m');
index_selected = get(handles.ListMem,'Value');
m(index_selected).name = char(get(handles.EditMemName,'String'));
m(index_selected).data = str2num(get(handles.StaticCapacity,'String'));
buses = get(handles.ListBusToMem,'String');
index_bus_selected = get(handles.ListBusToMem,'Value');
bus_selected = buses(index_bus_selected);
m(index_selected).attached_to_upstream = char(bus_selected);
list = get(handles.ListMem,'String');
list(index_selected) = {m(index_selected).name};
set(handles.ListMem,'String',list);
set(handles.ListPlotMem,'String',list);
set(handles.ListMemToTx,'String',list);
m(index_selected).EditCapacity = str2num(get(handles.EditCapacity,'String'));
m(index_selected).PopUpBit3 = get(handles.PopUpBit3,'Value');
assignin('base','m',m);

% --- Executes on button press in pushbutton27.
function pushbutton27_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton27 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
s = evalin('base','s');
index_selected = get(handles.ListSens,'Value');
s(index_selected).name = char(get(handles.EditSensName,'String'));
s(index_selected).data = str2num(get(handles.StaticDataRate,'String'));
s(index_selected).buffer = str2num(get(handles.StaticBuffer,'String'));
s(index_selected).activation_condition = char(get(handles.pay_act_logic_edit,'String'));
s(index_selected).EditAmount = str2num(get(handles.EditAmount,'String'));
s(index_selected).EditEvery = str2num(get(handles.EditEvery,'String'));
s(index_selected).PopUpBit = get(handles.PopUpBit,'Value');
s(index_selected).PopUpTime = get(handles.PopUpTime,'Value');
s(index_selected).EditSensBuffer = str2num(get(handles.EditSensBuffer,'String'));
s(index_selected).PopUpBit5 = get(handles.PopUpBit5,'Value');

buses = get(handles.ListSensToBus,'String');
index_bus_selected = get(handles.ListSensToBus,'Value');
bus_selected = buses(index_bus_selected);
s(index_selected).attached_to = char(bus_selected);
list= get(handles.ListSens,'String');
```

```
list(index_selected) = {s(index_selected).name};
set(handles.ListSens,'String',list);
set(handles.ListPlotSens,'String',list);
assignin('base','s',s);



% --- Executes during object creation, after setting all properties.
function uipanel16_CreateFcn(hObject, eventdata, handles)
% hObject    handle to uipanel16 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
%%%Choose between plus or times operation in compression%%%



% --- Executes during object creation, after setting all properties.
function uipanel20_CreateFcn(hObject, eventdata, handles)
% hObject    handle to uipanel20 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
%%%choose between plus or times operation in overhead%%%



% --- Executes during object creation, after setting all properties.
function uipanel27_CreateFcn(hObject, eventdata, handles)
% hObject    handle to uipanel27 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
%%%choose between plus or times operation in overhead%%%



function error_text_edit_Callback(hObject, eventdata, handles)
% hObject    handle to error_text_edit (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of error_text_edit as text
%        str2double(get(hObject,'String')) returns contents of error_text_edit as a double


% --- Executes during object creation, after setting all properties.
function error_text_edit_CreateFcn(hObject, eventdata, handles)
% hObject    handle to error_text_edit (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
function EditCompression2_Callback(hObject, eventdata, handles)
% hObject    handle to EditCompression2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditCompression2 as text
%        str2double(get(hObject,'String')) returns contents of EditCompression2 as a double


% --- Executes during object creation, after setting all properties.
function EditCompression2_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditCompression2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditSensBuffer_Callback(hObject, eventdata, handles)
% hObject    handle to EditSensBuffer (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditSensBuffer as text
%        str2double(get(hObject,'String')) returns contents of EditSensBuffer as a double


% --- Executes during object creation, after setting all properties.
function EditSensBuffer_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditSensBuffer (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in PopUpBit5.
function PopUpBit5_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpBit5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
str_buffer = get(handles.EditSensBuffer,'String');
buffer = str2num(str_buffer);
%conversione in bit dei suoi multipli:
x = get(handles.PopUpBit5,'Value');
if x == 1
    buffer_new = buffer * 1;
```

```
elseif x == 2
    buffer_new = buffer * 1e3;
elseif x == 3
    buffer_new = buffer * 1e6;
elseif x == 4
    buffer_new = buffer * 1e9;
elseif x == 5
    buffer_new = buffer * 1e12;
elseif x == 6
    buffer_new = buffer * 8;
elseif x == 7
    buffer_new = buffer * 8e3;
elseif x == 8
    buffer_new = buffer * 8e6;
elseif x == 9
    buffer_new = buffer * 8e9;
elseif x == 10
    buffer_new = buffer * 8e12;

end
set(handles.StaticBuffer,'String',buffer_new);


% Hints: contents = cellstr(get(hObject,'String')) returns PopUpBit5 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpBit5


% --- Executes during object creation, after setting all properties.
function PopUpBit5_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PopUpBit5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pushbutton31.
function pushbutton31_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton31 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)



function EditFrom_Callback(hObject, eventdata, handles)
% hObject    handle to EditFrom (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
StartTime = str2num(get(handles.EditFrom,'String'));
GMATout = evalin('base','GMATout');
set(handles.StaticStartTime,'String',GMATout.UTCGregorian(StartTime));

% Hints: get(hObject,'String') returns contents of EditFrom as text
```

```
%        str2double(get(hObject,'String')) returns contents of EditFrom as a double


% --- Executes during object creation, after setting all properties.
function EditFrom_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditFrom (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function EditTo_Callback(hObject, eventdata, handles)
% hObject    handle to EditTo (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
CompletionTime = str2num(get(handles.EditTo,'String'));
GMATout = evalin('base','GMATout');
set(handles.StaticCompletionTime,'String',GMATout.UTCGregorian(CompletionTime));

% Hints: get(hObject,'String') returns contents of EditTo as text
%        str2double(get(hObject,'String')) returns contents of EditTo as a double


% --- Executes during object creation, after setting all properties.
function EditTo_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditTo (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes during object creation, after setting all properties.
function StaticFrom_CreateFcn(hObject, eventdata, handles)
% hObject    handle to StaticFrom (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes on selection change in ListPlotSens.
function ListPlotSens_Callback(hObject, eventdata, handles)
% hObject    handle to ListPlotSens (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns ListPlotSens contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListPlotSens
```

```
GMATout = evalin('base','GMATout');
axes1 = handles.axes1;
s = evalin('base','s');
SENS_Buffer = evalin('base','SENS_Buffer');
PlotSens = get(handles.ListPlotSens,'Value');
LengthPlotSens = length(PlotSens);
col = hsv(LengthPlotSens);

if LengthPlotSens == 1
    hold(axes1,'off')

plot(axes1,GMATout.ElapsedDays,SENS_Buffer(PlotSens,:),'color',col(LengthPlotSens,:),'displayname',
s(PlotSens).name)
    legend(axes1,'location','ne')
else
    hold(axes1,'off')
    for j = 1 : LengthPlotSens

plot(axes1,GMATout.ElapsedDays,SENS_Buffer(PlotSens(j),:),'color',col(j,:),'displayname',s(j).name)
        hold(axes1,'on')
    end
    legend(axes1,'Location','ne');
end


% --- Executes during object creation, after setting all properties.
function ListPlotSens_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListPlotSens (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in PushLOAD.
function PushLOAD_Callback(hObject, eventdata, handles)
% hObject    handle to PushLOAD (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[filename, pathname, filterindex] = uigetfile({'*mat*'},'Select a file:');
evalin('base',['load(',char(39),pathname,filename,char(39),')']);
s = evalin('base','s');
b = evalin('base','b');
m = evalin('base','m');
t = evalin('base','t');
GMATout = evalin('base','GMATout');


set(handles.ListSens,'String',{s.name});
set(handles.ListPlotSens,'String',{s.name});
set(handles.ListBus,'String',{b.name});
set(handles.ListMem,'String',{m.name});
```

```
set(handles.ListPlotMem,'String',{m.name});
set(handles.ListTx,'String',{t.name});
set(handles.ListSensToBus,'String',{b.name});
set(handles.ListBusToMem,'String',{b.name});
set(handles.ListMemToTx,'String',{m.name});
set(handles.StaticnGS,'String',{t.num_GS_selected});

set(handles.ListSens,'Value',1);
set(handles.EditSensName,'String',s(1).name);
set(handles.StaticDataRate,'String',s(1).data);
set(handles.pay_act_logic_edit,'String',s(1).activation_condition);
set(handles.StaticBuffer,'String',s(1).buffer);
set(handles.EditAmount,'String',s(1).EditAmount);
set(handles.EditEvery,'String',s(1).EditEvery);
set(handles.PopUpBit,'Value',s(1).PopUpBit);
set(handles.PopUpTime,'Value',s(1).PopUpTime);
set(handles.EditSensBuffer,'String',s(1).EditSensBuffer);
set(handles.PopUpBit5,'Value',s(1).PopUpBit5);

set(handles.ListBus,'Value',1);
set(handles.EditBusName,'String',b(1).name);
set(handles.StaticDataRate2,'String',b(1).data);
set(handles.EditSpeed,'String',b(1).EditSpeed);
set(handles.PopUpBit2,'Value',b(1).PopUpBit2);
set(handles.StaticPlusOrTimes1,'String',b(1).StaticPlusOrTimes1);
set(handles.StaticPlusOrTimes2,'String',b(1).StaticPlusOrTimes2);
set(handles.EditOverhead,'String',b(1).EditOverhead);
set(handles.EditCompression,'String',b(1).EditCompression);

set(handles.ListMem,'Value',1);
set(handles.EditMemName,'String',m(1).name);
set(handles.StaticCapacity,'String',m(1).data);
set(handles.EditCapacity,'String',m(1).EditCapacity);
set(handles.PopUpBit3,'Value',m(1).PopUpBit3);

set(handles.ListTx,'Value',1);
set(handles.EditTxName,'String',t(1).name);
set(handles.StaticDataRate3,'String',t(1).data);
set(handles.EditDataRateDownlink,'String',t(1).EditDataRateDownlink);
set(handles.PopUpBit4,'Value',t(1).PopUpBit4);
set(handles.StaticPlusOrTimes3,'String',t(1).StaticPlusOrTimes3);
set(handles.StaticPlusOrTimes4,'String',t(1).StaticPlusOrTimes4);
set(handles.EditOverhead2,'String',t(1).EditOverhead2);
set(handles.EditCompression2,'String',t(1).EditCompression2);

%MULTIPLE SELECTION FOR GS LIST
 for i = 1 : t(1).num_GS_selected
     set(handles.ListGS,'Value',t(1).index_GS_selected(i));
     hold on
 end

fullpathname = handles.fullpathname;
file = fileread(fullpathname);
GMATout = evalin('base','GMATout');
GMATout = plot_GMAT_results(fullpathname,0);
x = fieldnames(GMATout);
set(handles.ListGS,'String',x);
```

```
% --- Executes on button press in PushSAVE.
function PushSAVE_Callback(hObject, eventdata, handles)
% hObject    handle to PushSAVE (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
s = evalin('base','s');
b = evalin('base','b');
m = evalin('base','m');
t = evalin('base','t');
[filename, pathname, filterindex] = uiputfile({'*mat*'},'Save your file:');
save(filename);


% --- Executes when selected object is changed in uipanel15.
function uipanel15_SelectionChangeFcn(hObject, eventdata, handles)
% hObject    handle to the selected object in uipanel15
% eventdata  structure with the following fields (see UIBUTTONGROUP)
%      EventName: string 'SelectionChanged' (read only)
%      OldValue: handle of the previously selected object or empty if none was selected
%      NewValue: handle of the currently selected object
% handles    structure with handles and user data (see GUIDATA)
x = get(handles.RadioPlus1,'Value');
Times = 'x';
Plus = '+';
if x == 0
    set(handles.StaticPlusOrTimes1,'String',Times);
else
    set(handles.StaticPlusOrTimes1,'String',Plus);
end


% --- Executes when selected object is changed in uipanel16.
function uipanel16_SelectionChangeFcn(hObject, eventdata, handles)
% hObject    handle to the selected object in uipanel16
% eventdata  structure with the following fields (see UIBUTTONGROUP)
%      EventName: string 'SelectionChanged' (read only)
%      OldValue: handle of the previously selected object or empty if none was selected
%      NewValue: handle of the currently selected object
% handles    structure with handles and user data (see GUIDATA)
x = get(handles.RadioPlus2,'Value');
Times = 'x';
Plus = '-';
if x == 0
    set(handles.StaticPlusOrTimes2,'String',Times);
else
    set(handles.StaticPlusOrTimes2,'String',Plus);
end


% --- Executes when selected object is changed in uipanel20.
function uipanel20_SelectionChangeFcn(hObject, eventdata, handles)
% hObject    handle to the selected object in uipanel20
% eventdata  structure with the following fields (see UIBUTTONGROUP)
%      EventName: string 'SelectionChanged' (read only)
```

```
%      OldValue: handle of the previously selected object or empty if none was selected
%      NewValue: handle of the currently selected object
% handles    structure with handles and user data (see GUIDATA)
x = get(handles.RadioPlus3,'Value');
Times = 'x';
Plus = '+';
if x == 0
    set(handles.StaticPlusOrTimes3,'String',Times);
else
    set(handles.StaticPlusOrTimes3,'String',Plus);
end


% --- Executes when selected object is changed in uipanel27.
function uipanel27_SelectionChangeFcn(hObject, eventdata, handles)
% hObject    handle to the selected object in uipanel27
% eventdata  structure with the following fields (see UIBUTTONGROUP)
%      EventName: string 'SelectionChanged' (read only)
%      OldValue: handle of the previously selected object or empty if none was selected
%      NewValue: handle of the currently selected object
% handles    structure with handles and user data (see GUIDATA)
%%%Choose between plus or times operation in compression%%%
x = get(handles.RadioPlus4,'Value');
Times = 'x';
Plus = '-';
if x == 0
    set(handles.StaticPlusOrTimes4,'String',Times);
else
    set(handles.StaticPlusOrTimes4,'String',Plus);
end


% --- Executes during object creation, after setting all properties.
function PushAddSens_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PushAddSens (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes during object creation, after setting all properties.
function PushAddTx_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PushAddTx (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called



function EditImpulsive_Callback(hObject, eventdata, handles)
% hObject    handle to EditImpulsive (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of EditImpulsive as text
%        str2double(get(hObject,'String')) returns contents of EditImpulsive as a double


% --- Executes during object creation, after setting all properties.
```

```
function EditImpulsive_CreateFcn(hObject, eventdata, handles)
% hObject    handle to EditImpulsive (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in PopUpTime2.
function PopUpTime2_Callback(hObject, eventdata, handles)
% hObject    handle to PopUpTime2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns PopUpTime2 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from PopUpTime2


% --- Executes during object creation, after setting all properties.
function PopUpTime2_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PopUpTime2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes when selected object is changed in uipanel29.
function uipanel29_SelectionChangeFcn(hObject, eventdata, handles)
% hObject    handle to the selected object in uipanel29
% eventdata  structure with the following fields (see UIBUTTONGROUP)
%      EventName: string 'SelectionChanged' (read only)
%      OldValue: handle of the previously selected object or empty if none was selected
%      NewValue: handle of the currently selected object
% handles    structure with handles and user data (see GUIDATA)
x = get(handles.RadioFinite,'Value');
if x == 1
    set(handles.EditImpulsive,'Enable','off');
    set(handles.EditEvery,'Enable','on');
else
    set(handles.EditEvery,'Enable','off');
    set(handles.EditImpulsive,'Enable','on');
end


% --- Executes on selection change in Listdt.
function Listdt_Callback(hObject, eventdata, handles)
% hObject    handle to Listdt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
```

```
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns Listdt contents as cell array
%        contents{get(hObject,'Value')} returns selected item from Listdt


% --- Executes during object creation, after setting all properties.
function Listdt_CreateFcn(hObject, eventdata, handles)
% hObject    handle to Listdt (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in listbox17.
function listbox17_Callback(hObject, eventdata, handles)
% hObject    handle to listbox17 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns listbox17 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from listbox17


% --- Executes during object creation, after setting all properties.
function listbox17_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox17 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in listbox18.
function listbox18_Callback(hObject, eventdata, handles)
% hObject    handle to listbox18 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns listbox18 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from listbox18


% --- Executes during object creation, after setting all properties.
function listbox18_CreateFcn(hObject, eventdata, handles)
% hObject    handle to listbox18 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
```

```
% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pushbutton38.
function pushbutton38_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton38 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on button press in pushbutton39.
function pushbutton39_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton39 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)



function edit37_Callback(hObject, eventdata, handles)
% hObject    handle to edit37 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit37 as text
%        str2double(get(hObject,'String')) returns contents of edit37 as a double


% --- Executes during object creation, after setting all properties.
function edit37_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit37 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pushbutton40.
function pushbutton40_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton40 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)



function edit36_Callback(hObject, eventdata, handles)
% hObject    handle to edit36 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Hints: get(hObject,'String') returns contents of edit36 as text
%        str2double(get(hObject,'String')) returns contents of edit36 as a double


% --- Executes during object creation, after setting all properties.
function edit36_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit36 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in popupmenu26.
function popupmenu26_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu26 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu26 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from popupmenu26


% --- Executes during object creation, after setting all properties.
function popupmenu26_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu26 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit35_Callback(hObject, eventdata, handles)
% hObject    handle to edit35 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit35 as text
%        str2double(get(hObject,'String')) returns contents of edit35 as a double


% --- Executes during object creation, after setting all properties.
function edit35_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit35 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
```

```
% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pushbutton37.
function pushbutton37_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton37 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on selection change in popupmenu24.
function popupmenu24_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu24 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu24 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from popupmenu24


% --- Executes during object creation, after setting all properties.
function popupmenu24_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu24 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on selection change in popupmenu25.
function popupmenu25_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu25 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns popupmenu25 contents as cell array
%        contents{get(hObject,'Value')} returns selected item from popupmenu25


% --- Executes during object creation, after setting all properties.
function popupmenu25_CreateFcn(hObject, eventdata, handles)
% hObject    handle to popupmenu25 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: popupmenu controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
```

```
end



function edit33_Callback(hObject, eventdata, handles)
% hObject    handle to edit33 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit33 as text
%        str2double(get(hObject,'String')) returns contents of edit33 as a double


% --- Executes during object creation, after setting all properties.
function edit33_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit33 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit34_Callback(hObject, eventdata, handles)
% hObject    handle to edit34 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit34 as text
%        str2double(get(hObject,'String')) returns contents of edit34 as a double


% --- Executes during object creation, after setting all properties.
function edit34_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit34 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in pushbutton36.
function pushbutton36_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton36 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on button press in pushbutton41.
```

```
function pushbutton41_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton41 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes during object creation, after setting all properties.
function pushbutton28_CreateFcn(hObject, eventdata, handles)
% hObject    handle to pushbutton28 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called


% --- Executes on selection change in ListPlotMem.
function ListPlotMem_Callback(hObject, eventdata, handles)
% hObject    handle to ListPlotMem (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: contents = cellstr(get(hObject,'String')) returns ListPlotMem contents as cell array
%        contents{get(hObject,'Value')} returns selected item from ListPlotMem

GMATout = evalin('base','GMATout');
axes3 = handles.axes3;
m = evalin('base','m');
MEM_occupation = evalin('base','MEM_occupation');
PlotMem = get(handles.ListPlotMem,'Value');
LengthPlotMem = length(PlotMem);
col = hsv(LengthPlotMem);

if LengthPlotMem == 1
    hold(axes3,'off')

plot(axes3,GMATout.ElapsedDays,MEM_occupation(PlotMem,:),'color',col(LengthPlotMem,:),'displayname'
,m(PlotMem).name)
    legend(axes3,'location','ne');
else
    hold(axes3,'off')
    for j = 1 : LengthPlotMem

plot(axes3,GMATout.ElapsedDays,MEM_occupation(PlotMem(j),:),'color',col(j,:),'displayname',m(j).nam
e)
        hold(axes3,'on')
    end
    legend(axes3,'location','ne');
end

% --- Executes during object creation, after setting all properties.
function ListPlotMem_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ListPlotMem (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: listbox controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
```

```
end


% --- Executes on button press in PushClearAll.
function PushClearAll_Callback(hObject, eventdata, handles)
% hObject    handle to PushClearAll (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
evalin('base',['clear all'])
OrigDlgH = ancestor(hObject, 'figure');
delete(OrigDlgH);
Memory_Budget;

% --- Executes on button press in pushbutton45.
function pushbutton45_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton45 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
b = evalin('base','b');
deleted_index = get(handles.ListBus,'Value');
currentListBus = get(handles.ListBus,'String');
newListBus = currentListBus;
newListBus(deleted_index,:) = [];
b(deleted_index) = [];
set(handles.ListBus,'String',newListBus);
set(handles.ListSensToBus,'String',newListBus);
set(handles.ListBusToMem,'String',newListBus);
set(handles.ListBus,'Value',1);
assignin('base','b',b);


% --- Executes on button press in PushLoadResults.
function PushLoadResults_Callback(hObject, eventdata, handles)
% hObject    handle to PushLoadResults (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[filename, pathname, filterindex] = uigetfile({'*mat*'},'Select a file:');
evalin('base',['load(',char(39),pathname,filename,char(39),')']);
GMATout = evalin('base','GMATout');
s = evalin('base','s');
b = evalin('base','b');
m = evalin('base','m');
t = evalin('base','t');
axes1 = handles.axes1;
axes3 = handles.axes3;
SENS_Buffer = evalin('base','SENS_Buffer');
MEM_occupation = evalin('base','MEM_occupation');
data_rate_mean = evalin('base','data_rate_mean');
downlink_tot = evalin('base','downlink_tot');
gen_tot = evalin('base','gen_tot');
error_flag = evalin('base','error_flag');
set(handles.ListPlotSens,'String',{s.name});
set(handles.ListPlotMem,'String',{m.name});


% --- Executes on button press in PushSaveResults.
function PushSaveResults_Callback(hObject, eventdata, handles)
```

```
% hObject    handle to PushSaveResults (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
SENS_Buffer = evalin('base','SENS_Buffer');
MEM_occupation = evalin('base','MEM_occupation');
data_rate_mean = evalin('base','data_rate_mean');
downlink_tot = evalin('base','downlink_tot');
gen_tot = evalin('base','gen_tot');
error_flag = evalin('base','error_flag');
s = evalin('base','s');
b = evalin('base','b');
m = evalin('base','m');
t = evalin('base','t');
GMATout = evalin('base','GMATout');
[filename, pathname, filterindex] = uiputfile({'*mat*'},'Save your file:');
save(filename);


% --- Executes on button press in PushSavePlots.
function PushSavePlots_Callback(hObject, eventdata, handles)
% hObject    handle to PushSavePlots (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[FILENAME, PATHNAME] = uiputfile('*.fig', 'Select file to save');
f2=figure;
a2=copyobj(handles.axes1,f2);
set(a2,'units','normalized','Position',[0.13 0.11 0.775 0.815]);
saveas(f2,[PATHNAME FILENAME],'fig');
%close(f2);


% --- If Enable == 'on', executes on mouse press in 5 pixel border.
% --- Otherwise, executes on mouse press in 5 pixel border or over ListSens.
function ListSens_ButtonDownFcn(hObject, eventdata, handles)
% hObject    handle to ListSens (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on button press in pushbutton49.
function pushbutton49_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton49 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
[FILENAME, PATHNAME] = uiputfile('*.fig', 'Select file to save');
f2=figure;
a2=copyobj(handles.axes3,f2);
set(a2,'units','normalized','Position',[0.13 0.11 0.775 0.815]);
saveas(f2,[PATHNAME FILENAME],'fig');
%close(f2);


% --- Executes when selected object is changed in uipanel22.
function uipanel22_SelectionChangeFcn(hObject, eventdata, handles)
% hObject    handle to the selected object in uipanel22
% eventdata  structure with the following fields (see UIBUTTONGROUP)
%      EventName: string 'SelectionChanged' (read only)
```

```
%      OldValue: handle of the previously selected object or empty if none was selected
%      NewValue: handle of the currently selected object
% handles    structure with handles and user data (see GUIDATA)


% --- Executes during object creation, after setting all properties.
function PushImport_CreateFcn(hObject, eventdata, handles)
% hObject    handle to PushImport (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called
```

Bibliografia

(s.d.). Tratto da http://www.navipedia.net/index.php/Satellite_Eclipses

Ciaburro, G. (s.d.). *MANUALE MATLAB*. Tratto da http://osiris.df.unipi.it/~ferrante/Tecnologie_fisiche/matlab.pdf

ESEO MISSION. (s.d.). Tratto da European Space Agency: http://www.esa.int/Education/ESEO_mission

General Mission Analysis Tool. (s.d.). Tratto da GMAT User Guide: http://gmat.sourceforge.net/docs/R2013a/html/index.html

Moler, C. (s.d.). *Experiments with MATLAB*. Tratto da MathWorks: https://it.mathworks.com/moler/exm/chapters.html

Wiley J. Larson, J. R. (s.d.). *SPACE MISSION ANALYSIS AND DESIGN*. Space Technology Library.

Ringraziamenti

Desidero ringraziare tutti coloro che mi hanno seguito nel percorso di tesi, conclusosi con questo elaborato.

Anzitutto ringrazio il professor Paolo Tortora, non solo per il supporto che mi ha dato, ma anche per il continuo impegno nel fornire agli studenti un'istruzione di qualità, grazie alla quale ho coltivato la mia passione nel modo migliore. Lo ringrazio infine per i preziosi consigli che mi hanno guidato nella scelta degli studi futuri. Proseguo con il ringraziare il dottor Gilles Mariotti, che mi ha accompagnato durante tutto il lavoro di tesi. Grazie alla sua guida e disponibilità, ho avuto modo di trarre da questa esperienza un grande giovamento a livello tecnico e personale.

Un ringraziamento particolare è rivolto agli amici speciali che hanno camminato con me in questi mesi. Grazie a Federica, la cui sensibilità e forza d'animo mi ha sempre aiutato a ritrovare il giusto spirito nei momenti più difficili. Grazie a Edo, fedele compagno di viaggio dal primo all'ultimo giorno; la sua passione per il cosmo e per la conoscenza, insieme con la sua solarità innata, hanno arricchito le nostre sedute di studio e la nostra amicizia. Grazie a Robi che, con trasparenza e saggezza, è stato un supporto per superare le sfide più difficili, oltre che un amico per il quale nutro una profonda stima. Grazie a Gabri, grazie a Pera e grazie a Simo: aver condiviso questo percorso con voi ha lasciato il segno in ogni mia giornata; è grazie a voi che ricorderò sempre questi anni con il sorriso. Un ringraziamento affettuoso va ad Alessandra, una sorella più che un'amica; non riesco ad immaginare questo cammino senza la tua presenza. Un ringraziamento immenso va alla Lello ed alla sua disponibilità, senza la quale non sarei arrivato così lontano; sei stata un riferimento essenziale, giorno dopo giorno, dall'inizio alla fine. Grazie a Fede, per la sua energia contagiosa. Grazie ad Ado, per essere un uomo dalle infinite risorse, dal quale ogni giorno è possibile prendere spunto per migliorare. Grazie a Denise, per avermi dimostrato di esserci sempre, per il confronto e la condivisione, per l'insaziabile passione per lo spazio e per la natura, che hanno alimentato il mio desiderio di superare i propri limiti. Grazie a Bianca, per il supporto e l'immancabile carica che mi ha trasmesso.

È il cuore che parla nel ringraziare Silvia, che più di chiunque atro ha creduto in me, che ogni giorno ha scelto di percorrere quest'avventura insieme, una strada in cui non sono mancati per lei scelte e sacrifici; questo traguardo l'abbiamo raggiunto insieme, sei stata la mia dolcissima stella binaria.

Ultimi, ma non per importanza, sono i ringraziamenti che rivolgo alla mia amata famiglia. Ringrazio il nonno Mamo e la nonna Nani: grazie al vostro saggio ed affettuoso supporto, mi avete guidato in piccole e grandi scelte, per gli studi e per la vita. Grazie al nonno Carmine ed alla nonna Laura, per

essere stati sempre al mio fianco, per tutto il calore e la gioia, forte dei quali ho potuto guardare sempre avanti. Grazie allo chef prof. nonché zio Guido, per il suo affetto e per avermi insegnato che lottando si realizzano i sogni più grandi. Grazie allo zio Stefano, al suo sorriso contagioso ed alla sua forza di divorare la vita, sei un modello per me. Un ringraziamento speciale va allo zio Alfredo: mi hai fatto crescere con una chitarra in mano, mi hai fatto conoscere la musica, che ha stravolto la mia vita; quello che sono oggi lo devo anche a te, perché senza la musica sarei una persona diversa.

Concludo questo testo ringraziando le persone a cui il lavoro stesso è dedicato: i miei genitori e mio fratello, il cui sostegno mi ha spinto a fare sempre del mio meglio, il cui amore mi ha nutrito ogni giorno, i cui sacrifici mi permettono di vivere la vita che desidero, loro che sono per me il sinonimo di felicità. Grazie babbo per essere il mio coach nella vita, il mio preparatore per l'IronMan della quotidianità, per avermi dimostrato che testa e cuore portano laggiù dove non pensavi di arrivare. Mi hai insegnato che, arrivati allo stremo delle forze, è necessario visualizzare un punto (lontano ma non troppo) e concentrarsi sul gesto atletico, tanto nella corsa quanto nella vita. Grazie mamma per essere la mia personal trainer, anche fuori dalla palestra: mi hai allenato a vivere ogni giorno con l'energia, la voglia di fare, di migliorare, di non fermarsi mai. Grazie per essere l'implacabile supereroina che sei. Grazie per essere sempre stata la prima persona con cui potevo condividere la mia giornata. Infine, grazie a Brando, il mio piccolo grande uomo. Per l'immenso affetto che mi dimostri, per la tua innata tenerezza. Perché i tuoi occhi pieni di gioia sono capaci di squarciare il buio dei momenti più neri. Perché il tuo spontaneo sostegno ha per me un valore inestimabile.

Un ultimo pensiero va al piccolo Jacopo che vive ancora in me, che non mi abbandona nonostante passi il tempo, che vive di sogni più che di timori, di punti di forza più che di punti deboli, che ha voglia di viversela e non di accontentarsi.