POLITECNICO DI MILANO
Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Costruzione di mappe basate su quadtree

AI & R Lab
Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Relatore: Ing. Matteo Matteucci
Correlatore: Ing. Davide Migliore

Tesi di Laurea di:
Mauro Brenna, matricola 671861

Anno Accademico 2006-2007

Ai miei genitori, Roberto e Laura, e alle mie nonne

Sommario

Questa tesi si colloca nel contesto di robotica mobile e, più precisamente affronta il problema del *mapping*, ovvero la generazione di una mappa 2D dell'ambiente usando i dati forniti dai sensori (un sensore laser), e le informazioni sulla posizione del robot ottenuta eventualmente come stima odometrica. Lo scopo della tesi esposta nei seguenti capitoli è, in primo luogo, quello di mostrare come sia possibile svolgere attività di *mapping* attraverso l'uso di una struttura dati quadtree in ambienti indoor reali; in secondo luogo è ottenere mappe globali consistenti con robot in movimento senza l'uso di informazioni odometriche per mezzo di tecniche di *scan matching*. Sono presentati immagini e risultati estratti in ambienti sia sintetici sia domestici, con informazioni sulla complessità degli algoritmi proposti. I risultati ottenuti hanno soddisfatto le aspettative: in molti casi si è raggiunto l'obiettivo della generazione di una mappa consistente a partire da dati rumorosi quali sono quelli ricevuti da un sensore laser reale. Raffinando gli algoritmi si ritiene inoltre possibile un utilizzo degli stessi in real time.

Ringraziamenti

Ringrazio Ivan Reguzzoni per l'aiuto che mi ha dato nello svolgimento della tesi, sia nella parte concettuale che di implementazione; questa tesi è in parte anche sua. Gli ingegneri Matteo Matteucci e Davide Migliore per i *brainstorming* all'AIRLab di Como e i preziosi suggerimenti. Elisa Maira che ha corretto le bozze di questa tesi e cercato di migliorare il mio italiano. Andrea Ratti per i consigli riguardo LaTeX, software con il quale la tesi è stata scritta. A tutti gli altri che ho mancato di citare, il mio grazie.

Indice

Capitolo 1

Introduzione

'Everywhere is walking distance if you have the time."

Steven Wright

Questa tesi si colloca nel contesto di robotica mobile e, più precisamente affronta il problema del *mapping*, ovvero la generazione di una mappa 2D dell'ambiente usando i dati forniti dai sensori (un sensore laser), e le informazioni sulla posizione del robot ottenuta eventualmente come stima odometrica.
In particolare l'approccio utilizzato ha permesso la costruzione di una *occupancy grid*, ovvero di una mappa che mostri in modo evidente le zone libere e attraversabili dal robot e le zone occupate da ostacoli quali pareti, porte chiuse, oggetti di vario tipo 'immersi' nell'ambiente, statico o dinamico[1], che il robot deve evitare.
Il *mapping*, come eposto nei prossimi capitoli, è un problema difficile a causa di fattori quali: la dimensione della mappa, l'incertezza del sensore e l'incertezza sulla stima della *pose*[2] del robot.
Lo scopo della tesi è permettere la costruzione di una mappa di tipo *occupancy grid* attraverso l'implementazione di una struttura dati quadtree[3] e la possibilità di ricostruire la mappa globale, benché non si abbiano informazioni precise riguardo la *pose* del robot in ogni istante di campionamento del laser, attraverso l'uso della tecnica di *scan matching*.

[1] L'ambiente è definito statico se non sono presenti altri robot o persone e gli oggetti non si spostano, oppure dinamico in caso contrario. Nei casi in esame si farà per lo più riferimento a ambienti statici.

[2] Include sia le coordinate cartesiane (x,y) e l'orientamento (angolo theta).

[3] Una struttura ad albero famosa in computer graphics per il suo utilizzo nella compressione di immagini binarie.

Il *mapping*, discusso in modo sistematico in [26], si oppone ad un problema duale, la localizzazione, che ha come obiettivo il calcolo della *pose* del robot nella mappa globale utilizzando i dati forniti dal sistema sensoriale. In [26][1][13] sono spiegati metodi per la corretta realizzazione di una *occupancy grid* di tipo classico: una semplice matrice 2D.
Il lavoro proposto si discosta da tali metodi nella ricerca di un *mapping* operante in real time attraverso una struttura dati ad albero di tipo quadtree [23] che permette di ridurre le risorse computazionali (qui inteso in termini di spazio di memoria occupata). In letteratura si possono trovare numerosi esempi di realizzazioni che sfruttano il quadtree o l'octree - la naturale estensione del quadtree alle tre dimensioni - affiancandolo alla griglia. Il procedimento usato negli articoli, che sono presentati nel Capitolo 2, permette però la compressione della mappa solo dopo la sua costruzione tramite matrice. Minori sono gli esempi di uso diretto di quadtree e.g. [9].
Per il lavoro di tesi si è prevalentemente fatto riferimento a [26] e al lavoro svolto da Grisetti durante il GMapping Camp 2005 la cui versione aggiornata è reperibile sul sito di OpenSLAM[4].
Molti sono i contributi personali non reperibili da altre fonti, che hanno permesso di ottenere algoritmi efficaci ed efficienti (ovvero che raggiungano l'obiettivo preposto e nel modo migliore possibile) nella fase di inserimento punti nel quadtree, nella gestione dei nodi vicini durante tale fase e nello *scan matcher*. Quest'ultimo è stato realizzato in più varianti per valutarne gli aspetti critici in termini di spazio e tempo computazionale. I metodi utilizzati si rifanno in parte al modello *IDC* [14][18] e GMapping Camp 2005.
I risultati ottenuti hanno soddisfatto le aspettative, in molti casi si è raggiunto l'obiettivo della generazione di una mappa consistente a partire da dati rumorosi quali sono quelli ricevuti da un sensore laser reale. Raffinando gli algoritmi si ritiene inoltre possibile un utilizzo degli stessi in real time.
Nel prossimo futuro si prevede di utilizzare il materiale qui esposto per lo sviluppo di un successivo lavoro di tesi che si focalizzerà sul *mapping* in tre dimensioni sfruttando come struttura dati l'octree. Per avere una scansione laser nella terza dimensione ci si servirà di un sistema basculante sul quale sarà montato il sensore laser range finder 2D. Esempi di *mapping* in 3D che sfruttano l'octree sono già presenti in letteratura [15][20].

Lo scopo della tesi esposta nei seguenti capitoli è, in primo luogo, quello di mostrare come sia possibile svolgere attività di *mapping* attraverso l'uso di una struttura dati quadtree in ambienti indoor reali; In secondo luogo

[4] http://openslam.org/ nella sezione Gmapping e GridSLAM.

è ottenere mappe globali consistenti con robot in movimento senza l'uso di informazioni odometriche per mezzo di tecniche di *scan matching*. Sono presentati immagini e risultati estratti in ambienti sia sintetici sia domestici con informazioni sulla complessità degli algoritmi proposti.

La tesi è strutturata nel modo seguente.

Nel Capitolo 2 si mostra lo stato dell'arte nei campi della ricerca nel campo del *mapping* e localizzazione.

Nel Capitolo 3 si illustra come è stato progettato la risoluzione dei problemi sopra citati e le difficoltà insorte.

Nel Capitolo 4 si descrive la soluzione proposta a livello logico.

Nel Capitolo 5 è brevemente descritta l'architettura del sistema utilizzata negli esperimenti.

Nel Capitolo 6 si espongono i risultati ottenuti e le valutazione sulla complessità degli algoritmi.

Nel Capitolo 7 sono delineate le future direzioni di ricerca e gli spunti che il lavoro esposto può offrire, sono inoltre riportate alcune conclusioni finali.

Nel Capitolo 8 è mostrata la documentazione a livello logico tramite UML come use case e component diagram e semplici schemi.

Nel Capitolo 9 si documenta in UML tramite class diagram e activity diagram il pacchetto software realizzato in modo piú dettagliato.

In Appendice A è presente una copia cartacea del software.

In Appendice B è possibile trovare utili riferimenti per un utilizzo corretto del pacchetto software.

Capitolo 2

Stato dell'arte

> *"The universe (which others call the Library) is composed of an indefinite and perhaps infinite number of hexagonal galleries, with vast air shafts between, surrounded by very low railings."*
>
> Jorge Luis Borges, "The Library of Babel" ["La Biblioteca de Babel"] (1941) First lines

Nella tesi svolta vengono affrontati due macro-problemi nell'ambito della robotica mobile. Il presente capitolo ha come scopo quello di analizzare le ricerche in corso nei due campi permettendo una visione più chiara del lavoro qui esposto.

2.1 Mapping

Il problema del *mapping* consiste nella generazione di una mappa globale[1] e consistente dell'ambiente (e.g. i muri non sono sovrapposti e rappresenti in modo fedele il mondo reale) avendo a disposizione i dati forniti dal sistema sensoriale del robot e le informazioni odometriche sulla *pose*[2]. Questo problema può essere considerato una sfida difficile da risolvere per numerosi motivi: [26]

- lo spazio delle ipotesi, ovvero di tutte le possibili mappe è enorme: infinite variabili nel caso di mappe continue, più di 10^5 dimensioni discretizzando la mappa. Il filtro di Bayes è pertanto inadatto al problema;

[1] si contrappone all'aggettivo locale e indica che è rappresento tutto l'ambiente esplorato dal robot, non solo zone vicine alla *pose* corrente utili, per esempio, ad evitare ostacoli.

[2] $x_t = (x, y, \theta)$, considerate esatte se si intende affrontare solo la tematica del mapping.

- l'apprendimento della mappa non è un problema a sé stante; in situazioni reali è necessario localizzare il robot nell'ambiente ad ogni scansione poiché l'odometria da sola non basta. Gli sforzi della ricerca nel fondere i due problemi convergono nelle tecniche di SLAM (Simultaneous Localization and Mapping problem).

Inoltre ad aumentare la difficoltà nella creazione di una mappa concorrono: [26]

- la dimensione dell'ambiente da ricostruire;
- il rumore che occorre nelle misure odometriche e nei sensori;
- possibili *perceptual aliasing* provocate da simmetrie nell'ambiente;
- i cicli: il robot deve essere in grado di riconoscere un ambiente già visitato.

Per ovviare ad alcuni di questi problemi in letteratura si è proceduto rappresentando la mappa in modo discreto utilizzando una visione a matrice o griglia (*grid*). Tra i possibili modelli di *mapping* [25][27] si può collocare l'*occupancy grid*.

2.1.1 Occupancy Grid

Thrun [25] si riferisce a *occupancy grid* come *mapping with known pose*. Questo metodo differenzia le zone occupate da ostacoli quali pareti, oggetti, persone, e dalle zone libere intese come aree nelle quali il robot può transitare senza rischi. Nell'approccio probabilistico a ogni cella della mappa è associato il valore di probabilità [26] che la stessa sia occupata. I limiti di questo tipo di *mapping* [25] sono la mancanza di un metodo efficiente per la stima dell'incertezza sulla *pose* e l'errore sul filtro di Bayes dato dall'assunzione di rumore indipendente, che aumenta se il sensore è inaccurato.

Le prime ricerche sull'utilizzo dell'*occupancy grid* in robotica si devono a Elfes [1]. Queste idee sono state poi sviluppate migliorandole computazionalmente e proponendo nuove varianti. Nella tesi si focalizzerà l'attenzione verso il problema del *mapping* che impiega strutture dati gerarchiche particolari denominate quadtree. Il quadtree è comunemente usato in computer graphics poiché permette di comprimere le informazioni dividendo ricorsivamente la mappa,o nel caso generico una immagine, in quattro sotto-quadrati (per dettagli si rimanda alla prossima sottosezione). È quindi possibile ottenere delle mappe utilizzando in numero minore risorse spaziali (spazio su disco, heap ...) attraverso l'uso di quadtree (2D) e octree (3D).

Payeur et al. [20] mostrano come sia possibile ottenere una rappresentazione a octree partendo dai dati di un laser 2D. Il sistema proposto costruisce una mappa di tipo *occupancy grid* nel modo seguente: i dati del sensore laser vengono elaborati in un blocco che calcola l'approssimante OPDF (*Occupancy Probability Distribution Function*, simile a quella proposta da Elfes [1]) dalla quale si ottengono delle griglie di tipo sferico che composte formano un *occupancy grid* cartesiano di tipo octree. Nelle successive pubblicazioni [21][19] si spiega come si è proceduto nella cosiddetta *sensor fusion* (la procedura che permette di ottenere da un insieme di dati eterogenei un unico dato 'somma' degli altri) e come l'incertezza originata da differenti fonti si propaghi nella rappresentazione dell'ambiente. La generazione di octree da dati grezzi del sensore non è comunque una ricerca recente; già nel 1984 Connolly [8] suggeriva come ottenerla: in primo luogo dall'immagine dei dati del sensore si genera una *mesh view of bicubic model*, in secondo luogo si perviene ad una *intensity image* e la formazione dei quadtree; Infine viene creato un octree a partire da quadtree.
Kraetzschmar et al. [17] (2004) mostrano come è stata impiegata con successo una struttura dati di tipo *probabilistic quadtree* per la creazione di mappe di risoluzione variabile. Lo scopo prefissato è di avere la possibilità di generare mappe di grandi dimensioni con buon dettaglio. Anche se non esplicitato l'uso del quadtree è delegato in un momento successivo al *mapping* vero e proprio per creare e comprimere la mappa globale. Nel *paper* si spiegano le proprietà dei nodi, l'uso della media dei sotto-rami per ottenere mappe di diversa risoluzione, nonché un tipo di mappa ritenuto più robusto, nella quale le zone sono colorate in base alla varianza.
Un esempio più recente è quello di Caveney and Hedrick [9] che utilizzano direttamente il quadtree nella costruzione della mappa e si avvalgono di un algoritmo di tipo *multiple Kalman filter*.
Una volta generata la mappa di tipo quadtree è anche possibile eseguire il *path planning* (scelta del percorso migliore per il raggiungimento di un punto obiettivo detto goal) senza conversioni [16][12] (aspetto che non è stato affrontato in questo lavoro). Per fornire una panoramica più generale si citano Jung e Gupta [15] che spiegano come usare gli octree per la costruzione di una *distance map* discreta per *path planning* e *collision detect*, e Biber e Straer [6] che propongono una rappresentazione alternativa per i dati laser denominata NDT (*Normal Distribution Transform*): anch'essa divide il piano in sotto celle come l'*occupancy grid*. Inoltre vengono presentati lo *scan matcher* utilizzato e la risoluzione al problema di SLAM. Tra i pregi del metodo in questione rientra la semplicità nello *scan matcher* poiché non sono necessarie corrispondenze esplicite tra punti e linee, e il calcolo che si

basa su formule analitiche risulta veloce e corretto.

2.1.2 Struttura Dati Quadtree

Questa struttura dati veniva inizialmente impiegata nella compressione di immagini in formato binario. L'albero permette di salvare solo i punti neri (o bianchi) come zone (quadrati) di colore uniforme insieme alla loro posizione.

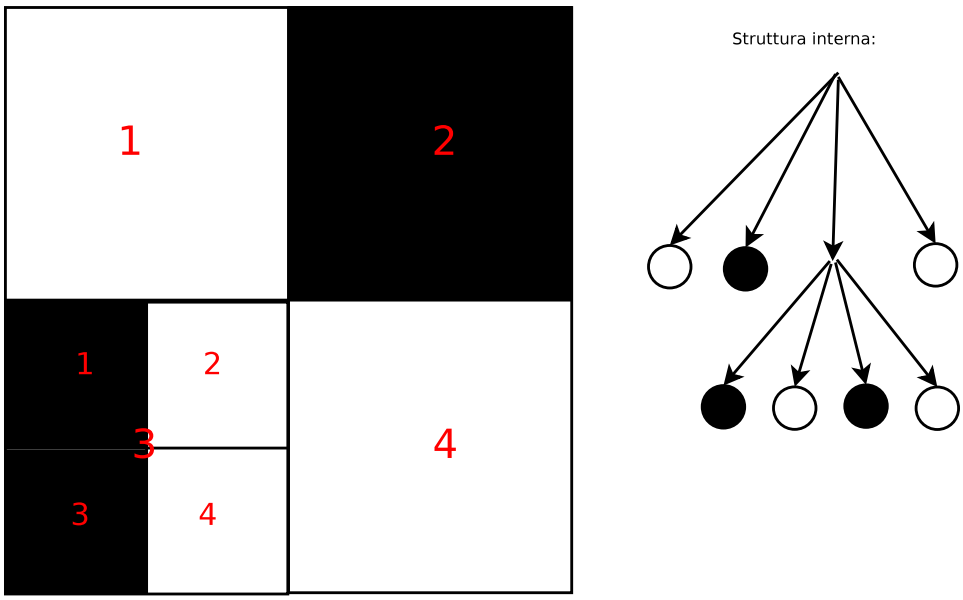


Figura 2.1: Rappresentazione di una struttura dati quadtree

La struttura dati originale è stata poi modificata e sono disponibili numerose varianti. In [23] Samet ne propone una panoramica focalizzandosi inoltre sui metodi di ricerca per i nodi vicini (si consiglia anche Bhattacharya [5]), metodi di rappresentazione alternativi, quadtree relativi a dati puntiformi e curvilinei. L'interesse per questo tipo di struttura dati è confermato dalle ricerche [5][11] e dall'utilizzo in numerosi campi, ad esempio, Edinéida [10].
Il quadtree utilizzato nella presente tesi come rappresentazione della mappa globale evolve in modo dinamico. In particolare il principio di funzionamento prevede, all'inserimento di un nuovo dato fornito dal sensore, la creazione dei sotto-rami fino al limite di massima risoluzione di ogni cella. Per maggiori dettagli sull'implementazione e i risultati ottenuti si rimanda ai successivi capitoli.

2.2 Localizzazione

Il problema della localizzazione nella robotica mobile riguarda la determinazione della *pose* del robot data la mappa dell'ambiente, i dati odometrici sulla *pose* e quelli ricevuti dal sistema di visione. La localizzazione può essere definita [26] come un problema di trasformazione delle coordinate. Le mappe sono descritte nel sistema di coordinate globali che è indipendente dalla *pose* del robot. La localizzazione è il processo dello stabilire corrispondenze tra il sistema di coordinate della mappa e quello delle coordinate locali del robot e permettere di esprimere la collocazione degli oggetti di interesse rispetto al proprio sistema di riferimento. Ad esempio, è naturale per l'uomo dire "questo oggetto è davanti a me" anziché esprimerlo come: "io sono in $(10, 10, 0°)$ e l'oggetto in $(20, 12)$".
Il problema nella determinazione della *pose* nasce dall'incertezza dei sensori che, essendo soggetti a rumori, rendono impossibile la determinazione esatta della *pose*. La tassonomia descritta in [26] fa dipendere la difficoltà del problema dalle seguenti caratteristiche:

Localizzazione locale/globale: La prima è un problema di position tracking con *pose* iniziale nota, la seconda non prevede la conoscenza della *pose*. Nella prima sorge il cosiddetto problema del *kidnapped robot*: se si sposta manualmente il robot durante la localizzazione esso non ritroverà la *pose* corretta;

Ambiente statico/dinamico: La dinamicità è data dalle persone, dalla luce[3], oggetti mobili, porte;

Approccio passivo/attivo: nel primo il modulo di localizzazione osserva il robot che compie azioni e si muove, indipendentemente da esso, nell'ambiente. Nel secondo il robot si muove al fine di minimizzare l'errore di localizzazione;

A robot singolo/multirobot: nel secondo caso esistono anche problemi di comunicazione.

2.2.1 Scan Matching

Con *scan matching* si intende una tecnica di localizzazione che sfrutta la somiglianza tra la scansione del laser attuale e la scansione di riferimento (anche la mappa globale stessa) per determinare la *pose* del robot.
Gutmann e Schlegel [14] comparano differenti approcci aventi come dati le

[3] nel caso dell'uso di telecamere.

scansioni 2D di un laser e la stima della *pose* attraverso odometria. Lo scopo è cercare la variante migliore per ambienti indoor simili ad un ufficio, che non presentino necessariamente muri ortogonali. Dalla classificazione si evince un modello *combined scan matching* che cerca di fondere i pregi di ogni classe. Secondo [14] vi sono tre tipi fondamentali di *scan matching*:

matching by assigning points to line segments: si estraggono le linee del *reference scan* e si confrontano rispetto all'ultimo *scan* preventivamente filtrato per cancellare dei punti spuri che non appartengono a possibili rette;

matching by cross correlation function: entrambi gli *scan* sono rimpiazzati da una rappresentazione stocastica a istogrammi. Si ricerca il massimo della funzione. Il metodo funziona bene solo in ambienti poligonali;

matching by point-to-point assignement, denominato idc: Non necessita di alcuna *feature* geometrica. La formula si esprime mediante matrici di errore covarianza.

In [7] Censi et al. è spiegato in maniera estensiva come realizzare uno *scan matcher* applicato nel dominio di Hough.
Biber e Strasser [3] mostrano la possibilità di confrontare N *scan* contemporaneamente, ottenendo così un metodo più robusto. Il metodo utilizzato è quello della cosiddetta funzione di energia, facendo riferimento a [2][4]. Un approccio differente alla localizzazione può essere quello del *particle filter* come spiegato in [26][22].

2.2.2 SLAM

Oltre alle tecniche proposte, alcune delle quali possono essere impiegate nello SLAM si citano Sujan et al. [24] che si servono del quadtree per decomporre immagini panoramiche e successivamente utilizzare un *information-based* SLAM, Grisetti et al. [13] che espongono il *grid mapping* tramite RBPF (Rao-Blackwellized Particle Filter) con pseudocodice allo scopo di ottenere un sistema robusto e un'alta qualità nella mappa risultante.

2.3 Realizzazioni

Nel lavoro svolto si è implementato un sistema basato su *occupancy grid* mapping applicando direttamente la struttura quadtree per l'inserimento dei dati senza passaggi intermedi. Lo *scan matcher* può essere considerato

di tipo *idc* e sfrutta una funzione di correlazione tra i punti ricevuti dall'ultimo *scan* e la mappa globale appresa. Si è cercato di seguire la logica programmativa KIS (*Keep It Simple*) e valutare i risultati ottenibili. Alcuni metodi di funzioni potrebbero essere ottimizzabili a scapito della leggibilità. La mappa proposta è di dimensioni variabili durante l'esecuzione, la creazione delle immagini è delegata ad un secondo tempo per permettere una versatilità maggiore. Con opportune modifiche (in *scan matcher* e ricezione *scan*) è teoricamente possibile un utilizzo in real time.

Capitolo 3

Impostazione del problema di ricerca

"What does a scanner see? he asked himself. I mean, really see? Into the head? Down into the heart? Does a passive infrared scanner like they used to use or a cube-type holo-scanner like they use these days, the latest thing, see into me - into us - clearly or darkly? I hope it does, he thought, see clearly, because I can't any longer these days see into myself. I see only murk. Murk outside; murk inside. I hope, for everyone's sake, the scanners do better. Because, he thought, if the scanner sees only darkly, the way I myself do, then we are cursed, cursed again and like we have been continually, and we'll wind up dead this way, knowing very little and getting that little fragment wrong too."

Philip K. Dick, "A Scanner Darkly"

L'obiettivo della ricerca è quindi la ricostruzione di una mappa globale che utilizzi direttamente una struttura ad albero di tipo quadtree. Questo ha posto in evidenza differenti problematiche poiché in quasi tutti i lavori in letteratura dei quali è stata presa visione la struttura dati è impiegata solo per comprimere la dimensione della mappa dopo che questa è stata generata. La fase di compressione è, infatti, distinta dall'elaborazione delle informazioni attuata per mezzo di una griglia (matrice) di dimensioni complessive predefinite, avente celle di dimensioni fisse.
Il quadtree è invece una struttura dati dinamica; ciò consente di avere celle di dimensione differente e di ingrandire la mappa a seconda delle esigenze e dei valori inseriti.
Saranno ora esposti brevemente una serie di problemi che si sono incontrati durante lo studio e l'implementazione del software. Una descrizione

dettagliata e la soluzione proposta per ognuno sarà esposta nel capitolo successivo.
Il primo problema affrontato è nato proprio dalla dimensione non uniforme delle celle: nell'albero le foglie possono essere presenti a diversi livelli e rappresentare zone quadrate di area differente. Si è quindi definita in modo preciso la funzione che identifica la probabilità che una determinata cella sia o meno occupata e che verrà di seguito chiamata **occprob** (Occupancy Probability). Inoltre è stato necessario capire come si debba comportare l'albero all'inserimento di un nuovo dato ricevuto dal sensore. La valutazione della funzione **occprob** non poteva essere fatta sempre in modo semplicistico, come sarà chiaro dalla lettura dei prossimi capitoli.
Un secondo problema è come rappresentare le zone libere, ovvero 'sicure' per il passaggio del robot; è evidente che una singola osservazione (un solo *scan* completo del sensore laser) non basta.
Un problema più complesso è quello dato dalla natura a griglia della mappa. La visione di un mondo discreto può facilmente causare errori nella gestione delle zone 'viste' dal sensore. Un segmento infatti deve essere inteso come un insieme di quadrati e non più come un insieme infinito di punti. Si è implementato l'algoritmo di Bresenham, ben noto in letteratura, per evitare questi errori e applicato una 'maschera' per vietare l'aggiornamento di alcuni punti.
Si è poi affrontato il problema dello *scan matching* cioè la ricerca della *pose* che massimizzi la probabilità dei dati acquisiti rispetto: la mappa globale, la *pose* del robot al passo precedente ed eventualmente l'odometria. Lo scopo è quello di migliorare/trovare la posizione attuale del robot in modo da inserire correttamente i dati e creare una mappa globale consistente. I metodi proposti nel prossimo capitolo si basano sulla massimizzazione di una funzione che lega i dati ricevuti dal sensore alla mappa costruita. La formalizzazione e corretta implementazione della funzione e dei metodi di massimizzazione ha posto numerose difficoltà e si sono resi necessari strumenti di debug per la loro verifica.

Capitolo 4

Progetto logico della soluzione del problema

"Any intelligent fool can make things better, more complex, and more violent. It takes a touch of genius, and a lot of courage to move in the opposite direction."

Albert Einstein

Si procederà ora approfondendo le problematiche enunciate e delineando le soluzioni proposte dal punto di vista concettuale e logico.

4.1 Occupancy Grid

L'*occupancy grid* è un tipo di mappa che differenzia le zone occupate da quelle libere. Il metodo più semplice per sfruttare le informazioni del sensore laser [26] è quello di avere una mappa discreta e assegnare ad ogni cella un valore di occupanza ovvero un valore che quantifichi in modo univoco la sua probabilità di essere occupata.
Un'enunciazione matematica formale identifica il problema del *mapping* in senso probabilistico come il calcolo di $p(m|z_{1:t}, x_{1:t})$ dove m è la mappa, $z_{1:t}$ l'insieme delle misure, $x_{1:t}$ l'insieme delle *pose* fino all'istante t. L'uso della occupancy grid permette di calcolare questa probabilità come:

$$p(m|z_{1:t}, x_{1:t}) = \prod_i p(m_i|z_{1:t}, x_{1:t})$$

dove m_i è la cella di indice i e $p(m_i)$ la probabilità della cella di essere occupata.
Nell'implementazione proposta sono presenti due variabili:

vis indica quante volte la cella è stata 'vista' dal sensore/visitata dall'algoritmo. Più precisamente si intende qualsiasi punto appartenente al segmento *pose-endpoint* dove il secondo termine si riferisce alla coppia (x, y) restituita dal sensore trasformata rispetto le coordinate assolute;

occ indica quante volte un punto finale ricevuto dal sensorela è appartenuto alla cella, ovvero quante volte è stata considerata occupata.

Il valore di ogni cella di essere occupata, **occprob**, è dato in generale dalla probabilità: $\frac{occ}{vis}$ (il risultato è sempre compreso tra 0 e 1).
A questo punto occorre osservare che: quando il laser colpisce un oggetto, e.g., una parete, e il sensore restituisce un nuovo punto, ciò significa che tutti i punti compresi dal segmento che ha per estremi la *pose* del sensore e il punto trovato sono da considerare 'liberi', in quanto il raggio del laser li ha attraversati. Quindi, all'inserimento di un nuovo dato, non solo si aggiornano i valori **vis** e **occ** della cella corrispondente, ma anche i **vis** dei punti sul segmento. Con la parola dato si intende sempre il valore restituito dal sensore come coppia (r, Θ) e trasformato in (x, y) tramite $x_t \bigoplus z_t$. È possibile osservarne il risultato in Figura 4.3
Se, ad esempio, il robot è posto in una stanza quadrata allora l'interno della stanza risulterà libero, i bordi/le pareti occupate e fuori dalla stanza sarà presente un valore convenzionale, ad esempio, 0.5 che rappresenti il concetto di informazione sconosciuta.

4.2 Struttura dati Quadtree

La struttura dati è mostrata in modo concettuale in figura:

Ogni nodo ha le stesse proprietà (nella mappa risultante i dati che contano sono solo quelli dei nodi foglia), in particolare possiede i valori:

vis: numero di volte in cui la cella è stata visitata;

occ: numero di volte in cui la cella è osservata come occupata;

dimlato: la dimensione del lato del quadrato in cm;

xcorner,ycorner: valori in coordinate cartesiane in riferimento assoluto del vertice in basso a sinistra del quadrato;

4 puntatori ai nodi che rappresentano le radici di sotto-rami.

La situazione iniziale è rappresentata, a livello di struttura interna, in figura:

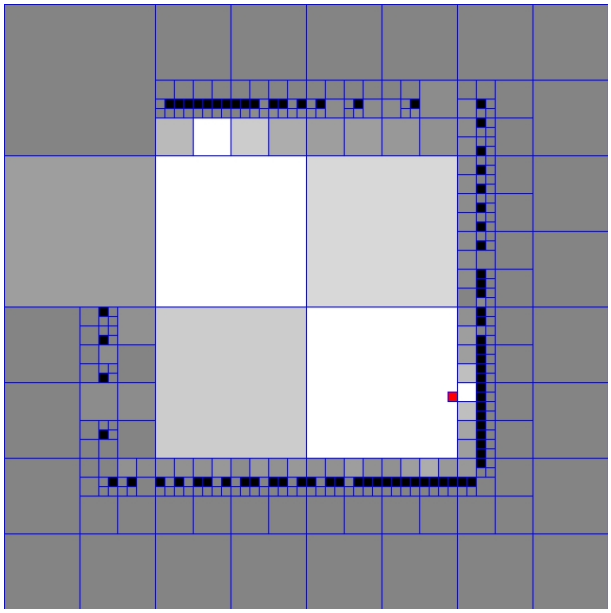
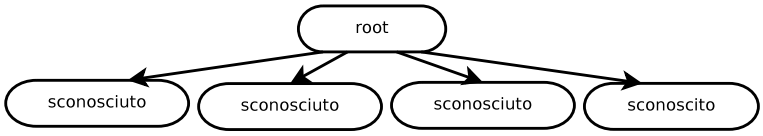


Figura 4.1: Simulazione di una stanza quadrata. Il robot , indicato come un quadrato rosso, è presente all'interno della stanza.



La mappa corrispondente è composta da quattro quadrati grigi. Ad ogni livello dell'albero la **dimlato** è dimezzata. L'inserimento di un punto (x,y) è così definito: si crea, se non presente, il sotto-albero che contiene il punto fino all'ottenimento di foglie di **dimlato** minore o uguale alla massima risoluzione della griglia voluta e si aggiornano i valori **occ** e **vis** della cella alla quale appartiene il punto. L'idea è quindi quella di ottenere un albero che abbia sotto-rami fino alla massima risoluzione solo se in una foglia a profondità massima vi è inserito un punto. Confrontando con la griglia di dimensioni fisse N, si avrà profondità dei rami pari a $log(N*N)$[1] solo nei sottorami interessati dai dati ricevuti dal sensore. Ad esempio se si immagina un robot in una stanza quadrata le aree fuori dalla stanza, sconosciute, corrisponderanno a sotto-alberi e foglie di livello vicino alla radice dell'albero. In questi casi se si ha una zona quadrata fuori dal perimetro della stanza di dimensione 1 m, allora la sua area sarà 1 m^2 e nel quadtree potrebbe essere rappresentata da una sola foglia mentre in una griglia fissa di celle

[1] valore limite valido solo se si inseriscono N*N valori distinti

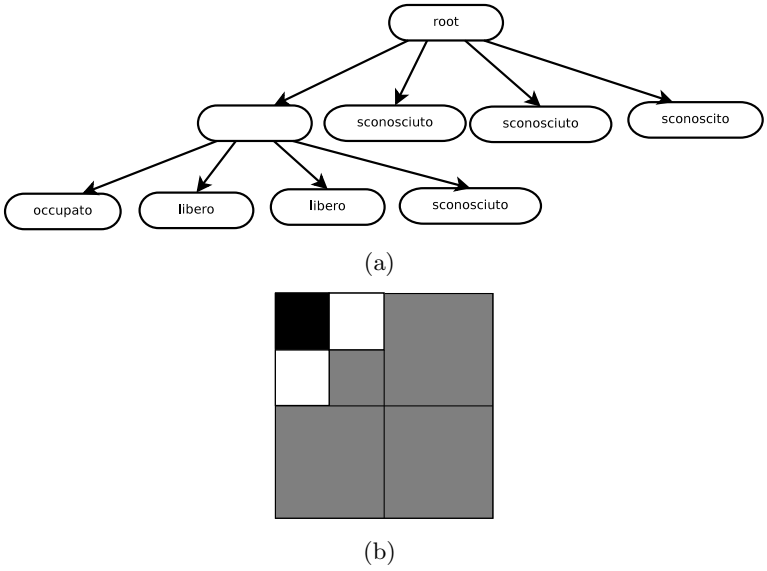


Figura 4.2: Esempio di struttura dati quadtree usata e mappa risultante

di risoluzione 1 cm si avranno sicuramente 100^2 celle con valori ancora non inizializzati, e di conseguenza un evidente spreco delle risorse di memoria.

4.3 Scan Matcher

Il problema dello *scan matcher* è quello di trovare la *pose* attuale del robot data l'ultima lettura del sensore, la mappa globale finora acquisita e informazioni di odometria.
Nel caso in esame sono stati impiegati due algoritmi di tipo *greedy* che sfruttano i dati, z_t, la mappa,m_{t-1} , e la *pose* del robot all'istante precedente, x_{t-1} , senza informazioni riguardo velocità e spostamenti. Il loro obiettivo è quello di massimizzare una funzione, detta **score**, che fornisce un valore di correlazione tra z_t e m_{t-1}.
Anche per per questa funzione si propongono due varianti elencandone pregi e difetti.

4.3.1 Massimizzazione per 'inseguimento'

Il primo tipo di *scan matcher* proposto si basa su un tutorial presentato da Grisetti in GMapping Camp 2005. I passaggi richiesti per la massimizzazione della funzione **score** sono i seguenti:

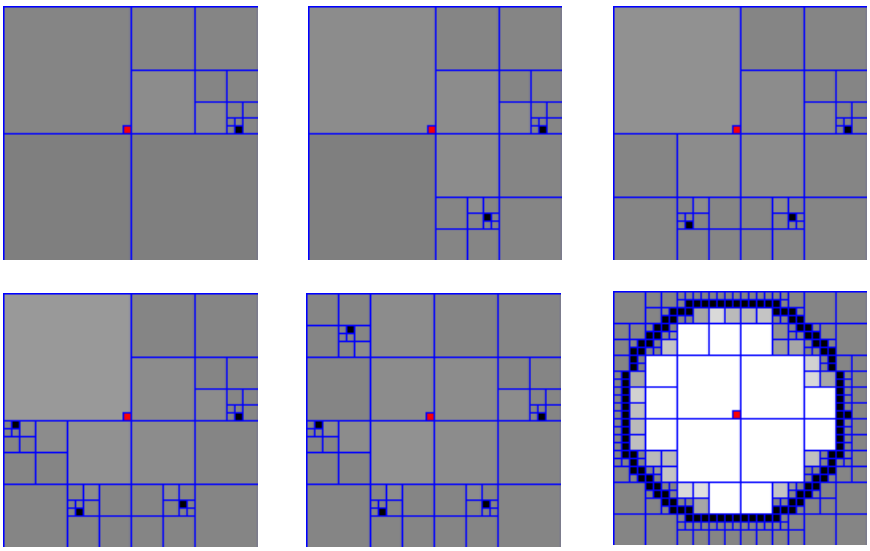


Figura 4.3: Questa sequenza mostra graficamente come funziona l'algoritmo di inserimento dati proposto in una stanza rotonda.

1. Valuta la funzione **score** per la *pose* $x_{t-1} = (x, y, \Theta)$, salvala come `migliorescore`;
2. Per un numero sufficiente di iterazioni esegui:
   - Valuta la funzione **score** per la *pose* $x^* = x_{t-1} + \Delta Spostamento$ dove $\Delta Spostamento$ in sequenza simula un movimento del robot in avanti, indietro, sinistra, destra, rotazione sinistra, rotazione destra;
   - Se la **score** ottenuta in uno dei movimenti è migliore della `migliorescore` sostiuiscila come `migliorescore`;
3. Restituisci la migliore.

Lo scopo dell'algoritmo è quello di arrivare per passi alla soluzione migliore. Il metodo simula un 'inseguimento' della x_t a partire da x_{t-1}. Un possibile problema è quello dei massimi locali causato dalla natura *greedy* dell'algoritmo.

4.3.2 Massimizzazione per 'campionamento e ordinamento'

L'alternativa proposta si può intendere come un campionamento da una gaussiana abbinato ad un algoritmo di *ranking*.

Il procedimento può essere descritto come:
Premessa.

- Da x_{t-1} genera un numero k di *pose* campionando una normale gaussiana di varianza σ^2.
  Più precisamente da $(x, y, \Theta) \longrightarrow (N(x, \sigma^2), N(y, \sigma^2), N(\Theta, \sigma^2))$
- Valuta la **score** di tutte le *pose* generate e ordina in base alla migliore **score**.

Algoritmo.

- Dalle prime j, $j < k$, *pose* candidate genera
  k *pose* (oppure k-j se si vuole mantenere le j per non perdere possibili massimi) con $\sigma' = \frac{\sigma}{NumeroIterazione}$
- valuta la **score** e ordina in base alla migliore

Terminazione.

- Restituisci la migliore.

Anche questo algoritmo è di tipo *greedy* ma il campionamento dalla gaussiana potrebbe 'far uscire'da un massimo locale.

4.4 Problemi Affrontati

4.4.1 Scelta della funzione occprob

Per generare mappe consistenti è necessario scegliere una buona funzione **occprob**. Si nota che la semplice formula $\frac{occ}{vis}$ non soddisfa appieno alcune ipotesi. Innanzitutto vincoli della formula sono $occ >= 0$, $vis >= 0$ e $vis >= occ$. Può accadere che $occ = vis = 0$; in questo caso si pone convenzionalmente $\frac{occ}{vis} = 0.5$ per rappresentare il valore di probabilità $\frac{1}{2}$, ovvero sconosciuto (né libero, né occupato). Se $occ = 1$ e $vis = 1$ allora $\frac{occ}{vis} = 1$ già dalla prima iterazione, si ritiene comunque corretto poiché si vuole individuare la soluzione più conservativa possibile, ovvero il robot non deve scontrarsi contro un eventuale ostacolo. Appena ne percepisce la presenza deve tenerne in considerazione la posizione ed evitarlo anche se lo ha trovato una sola volta (se non si ripresenta più la cella verrà aggiornata durante successive letture e il suo valore si riporterà vicino a zero). Per lo stesso motivo si vuole evitare che $occ = 0$ e $vis = 1$ comportino $\frac{occ}{vis} = 0$, cioè libero. Non è sufficiente un solo dato per stabilire che una zona è libera al passaggio. Per questo si è posto un bias di questo tipo:

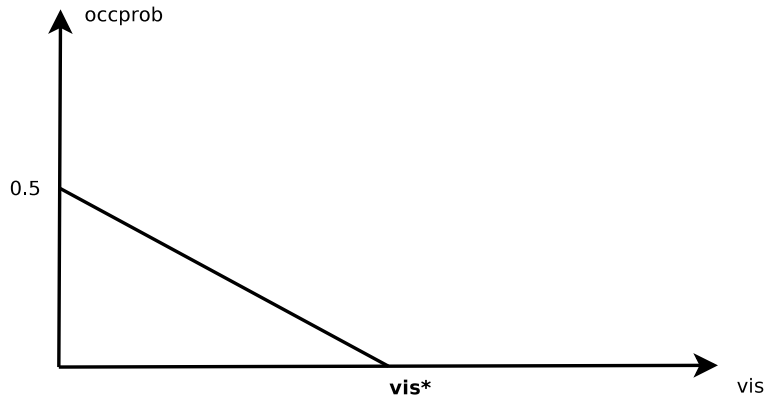


Figura 4.4: Bias impiegato per la funzione **occprob**

La figura mostra che se $occ = 0$ finché vale $vis < vis*$ la **occprob** non è calcolata come occ/vis ma dipende linearmente da vis. Il risultato restituito è compreso tra 0.5, valore sconosciuto, e 0, libero (Figura 4.3).

4.4.2 La natura discreta della mappa

Secondo quanto esposto precedentemente, inserendo un nuovo valore nella mappa tutti i punti del segmento sensore-punto[2] inserito sono considerati liberi e pertanto il loro valore di **vis** deve aumentare. Sorge il problema della natura discreta della mappa: il segmento si può considerare un insieme di quadrati di lato pari alla massima risoluzione anziché un insieme di punti. Perciò un approccio di aggiornamento dei quadrati che sfrutti l'intersezione di questo segmento - inteso in senso matematico - con la mappa produce risultati errati, (Figura 4.5).
Alcune foglie sono aggiornate anche se solo piccolissime parti del segmento le intersecano causando una percezione sbagliata dell'ambiente: molte più zone appaiono libere (questo comportamento ovviamente non è voluto). La soluzione proposta è di sfruttare l'algoritmo di Bresenham, per la rasterizzazione di linee, per ottenere l'insieme dei quadrati di dimensione pari alla massima risoluzione da aggiornare. In realtà se nel quadtree abbiamo un quadrato (foglia) di $dimlato > massimarisoluzione$ al quale appartengono più quadrati considerati in Bresenham non si vuole aggiornare più volte la foglia in esame poiché l'area del quadrato è molto maggiore dell'area dei quadrati intersecati dal segmento, e non è corretto diminuire molto il valore di **occprob**. Questo singolo accorgimento non è sufficiente; infatti, nelle

[2]il termine sensore è usato spesso come sinonimo di robot poichè si può far coincidere l'origine del sistema riferimento robot con quella del sensore laser.

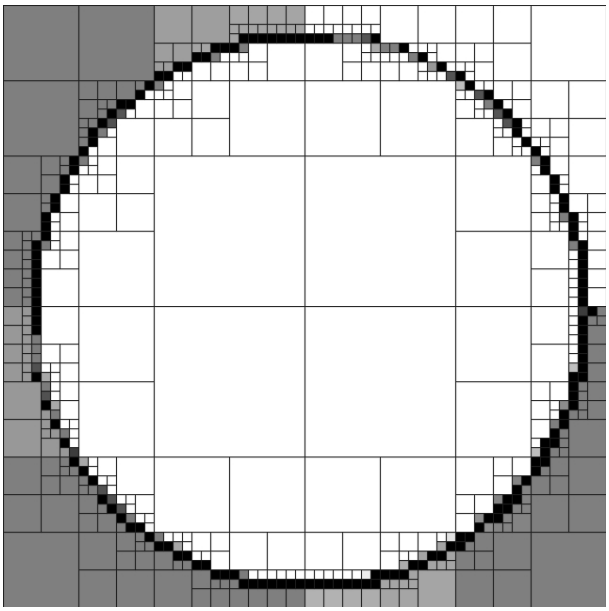


Figura 4.5: Il robot è interno alla stanza circolare simulata. Non dovrebbe apparire la parte libera esterna alla stanza né quadrati grigi, cioè meno occupati, vicino a quelli occupati

aree che rappresentano le pareti della stanza, può accade che, all'inserimento di un punto appartenente alla parete, si abbassi la **occprob** di un punto precedente anch'esso sulla parete. Immaginando nere le celle occupate e bianche le libere in un insieme di celle orizzontale, quando una cella viene scurita, la cella a fianco risulta più chiara del passo precedente. Per evitare questo effetto si utilizza una 'maschera' che vincola il non aggiornamento dei punti prossimi al punto inserito/aggiornato (Figura 4.6).

4.4.3 Funzioni Score implementate e considerazioni

Si sono implementate due funzioni di **score** entrambe esprimibili matematicamente nel seguente modo:

$$totscore = \sum_{i=1}^{N} score(z_t^i, m, x(t-1))$$

Nella prima variante, $score(z_t^i, m, x(t-1))$ non è altro che la probabilità della cella a cui appartiene il punto terminale (considerato sempre in coordinate assolute se non espressamente scritto) di essere occupata. In formule[3]:

[3]Per chiarezza si intende $z_t^i \oplus x(t-1)$ in questo modo: $z_t^i = (r, \alpha)$, $x(t-1) = (x, y, \Theta)$ e il risultato $(x', y', \Theta') = (x + r * cos(\alpha + \Theta), y + r * sin(\alpha + \Theta), \alpha + \Theta)$

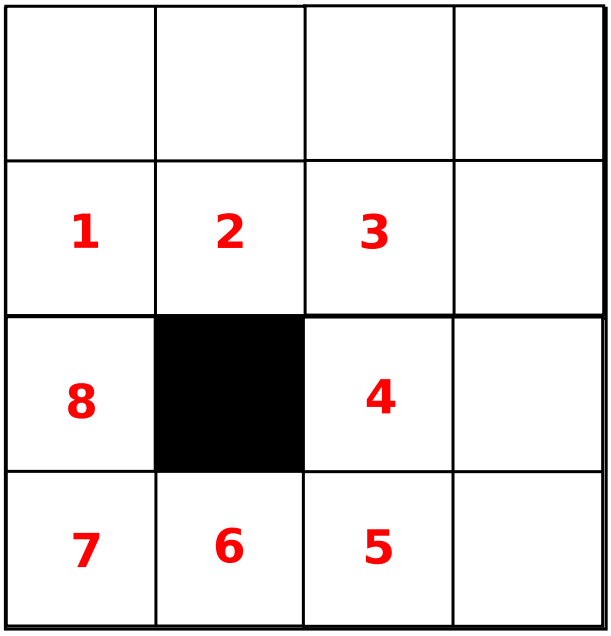


Figura 4.6: Funzionamento della maschera. Le celle circostanti a quella aggiornata non vengono modificate nell'inserimento

$$score(z_t^i, m, x(t-1)) = occprob_m(z_t^i \oplus x(t-1))$$

Si sfrutta cioè in modo diretto la conoscenza presente nella mappa attraverso la funzione **occprob** utilizzandola come parametro per misurare quanto è probabile che il prossimo punto ottenuto dal laser possa appartenere alla cella corrispondente. Per dettagli implementativi consultare Il Capitolo 9.

Nella seconda variante, invece, $score(z_t^i, m, x(t-1))$ è la distanza minima, considerata negativa nella sommatoria, tra il punto terminale e la cella occupata (inteso come avere un valore di **occprob** $> BIAS$) più vicina nella mappa.

Entrambe le **score** presentano pregi e difetti. La prima è indubbiamente più veloce: percorrere l'albero fino alla massima risoluzione richiede $\Theta(log(N))$ con N nodi foglia, ma la funzione è molto irregolare al variare dei parametri di ingresso e avrà un maggior numero di massimi locali. La **score** basata sulla distanza è molto più lenta rispetto alla prima, almeno in una versione non ottimizzata, ma presenta un andamento migliore che ben si adatta a un uso come quello fatto nel metodo di *scan matching* per 'inseguimento'. Alcune osservazioni si possono evincere da Figura 4.7 e Figura 4.8.
In Figura 4.7, la prima score enunciata, non basandosi sulle distanze, re-

stituisce lo stesso valore in entrambe le situazioni. Infatti immaginando il quadrato inferiore come lo *scan* corrente e il superiore come quello di riferimento, le aree bianche come **occprob** $= 0$ e quelle nere come **occprob** $= 1$ il valore della score è 2 ovvero la **occprob** dei due punti di intersezione. Il massimo correttamente si ha quando le due figure sono sovrapposte. Nel caso della score basata sulla distanza, Figura 4.8, la lunghezza del segmenti minimi che collega ogni punto del nuovo *scan* a quello di riferimento sono diverse. Il risultato calcolato come sommatoria delle distanze con segno meno restituisce due valori differenti. Il massimo è il valore 0 che si avrà quando le figure combaciano.

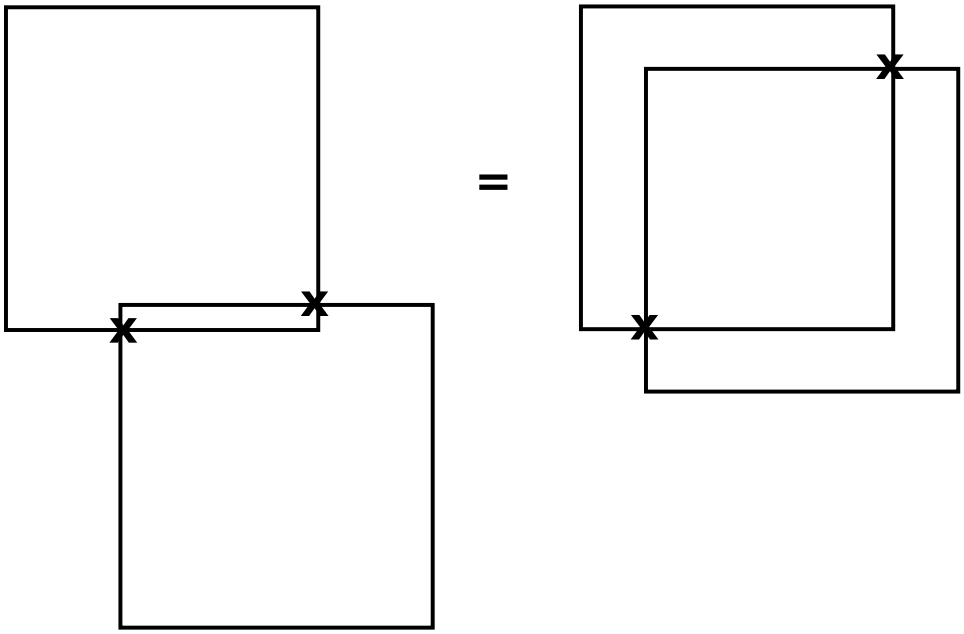


Figura 4.7: Esempio di **score** *non basata sulla distanza. Il risultato ottenuto è identico in entrambe le situazioni.*

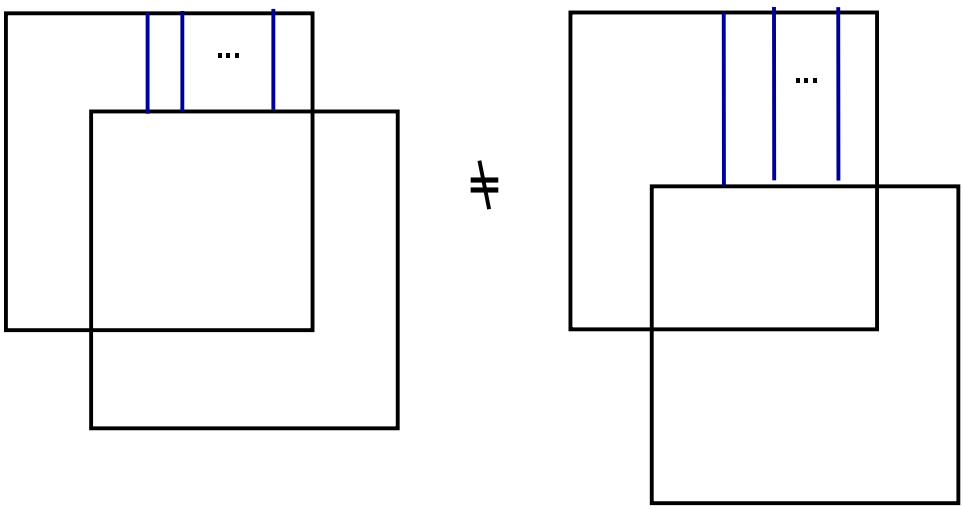


Figura 4.8: Esempio di **score** *basata sulla distanza.*

Capitolo 5

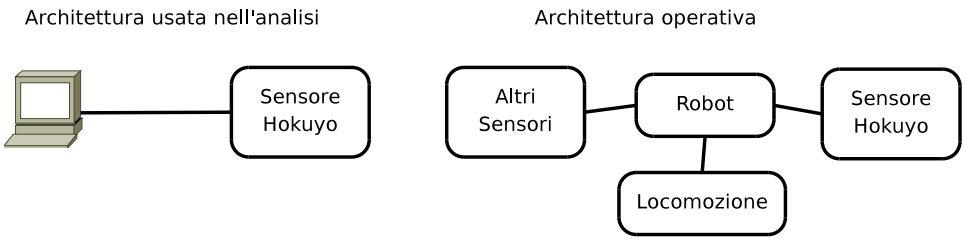
Architettura del sistema

"The whole visible world is perhaps nothing more than the rationalization of a man who wants to find peace for a moment. An attempt to falsify the actuality of knowledge, to regard knowledge as a goal still to be reached."

Franz Kafka, "Parables and Paradoxes"

5.1 Architettura Hardware

Per lo sviluppo della tesi si è utilizzato un computer[1] dotato di sistema operativo GNU/Linux collegato tramite porta USB al sensore laser Hokuyo URG-04LX. La figura illustra in modo schematico l'architettura utilizzata:



5.1.1 Hokuyo URG-04LX

Il sensore utilizzato è di tipo *scanning laser range finder*, capace di eseguire una scansione circolare lungo l'asse orizzontale (Figura 5.2). Dalle specifiche tecniche si ricavano i valori in tabella:

[1] descritto in dettaglio nella tabella tempi esecuzione in Capitolo 6

Figura 5.1: Foto del sensore laser utilizzato

Dimensioni	Peso	Consumo
$L50; W : 50; H70mm$	$160g$	$2.5W$
Accuratezza	Risoluzione	Area di Scansione
$\pm 10mm$	$0.36°$	$240°$

Il periodo di una scansione nominale è di circa $100ms$.[2] Sperimentalmente si è osservato che il sensore non funziona correttamente in presenza di ostacoli di colore nero posti non perpendicolarmente al piano di lettura.

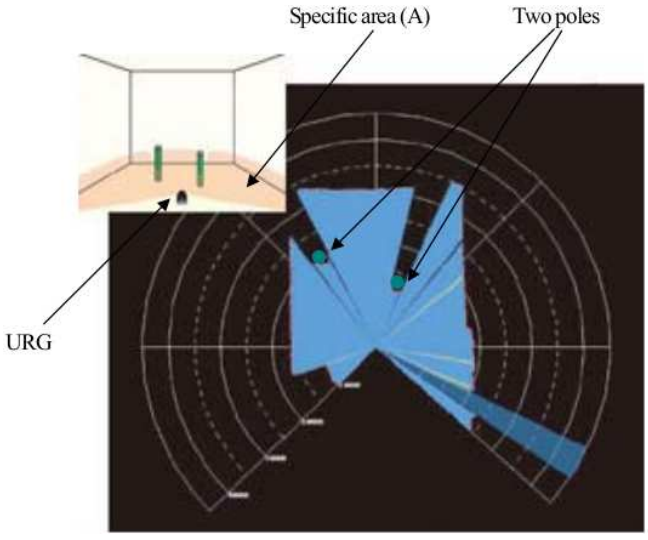


Figura 5.2: Esempio di scansione del sensore, Fonte Hokuyo Brochure PDF

[2]salvando i dati su file il tempo di esecuzione , che comprende l'inserimento di tutti i punti e l'algoritmo di Bresenham, cresce fino a circa un secondo per scansione

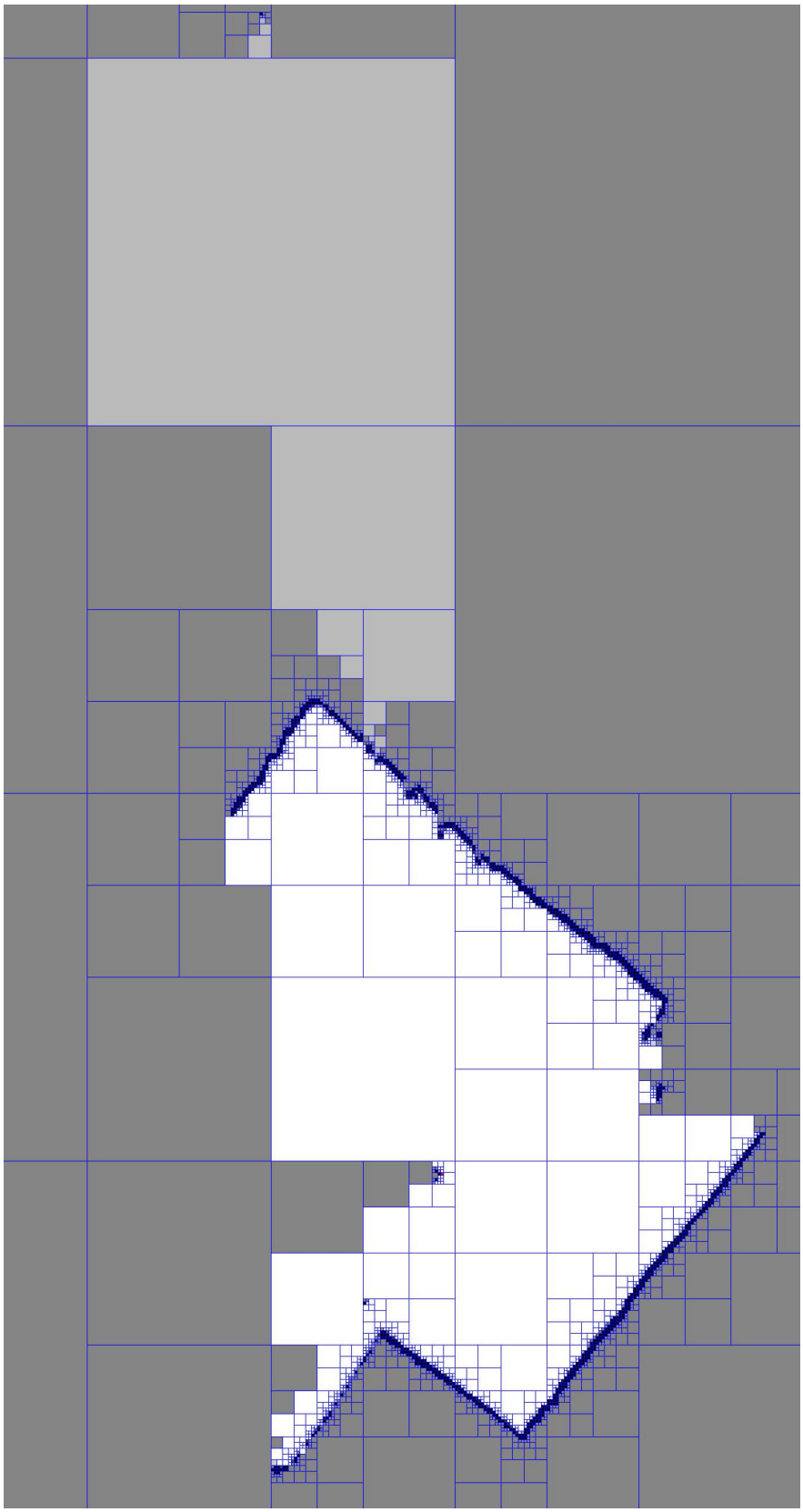
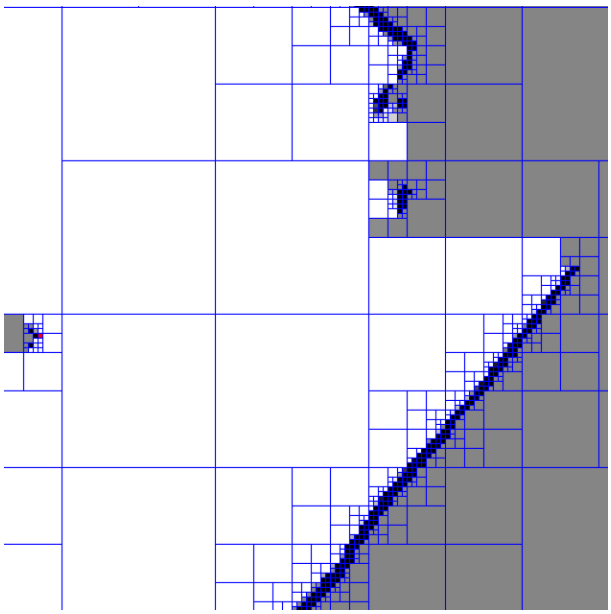
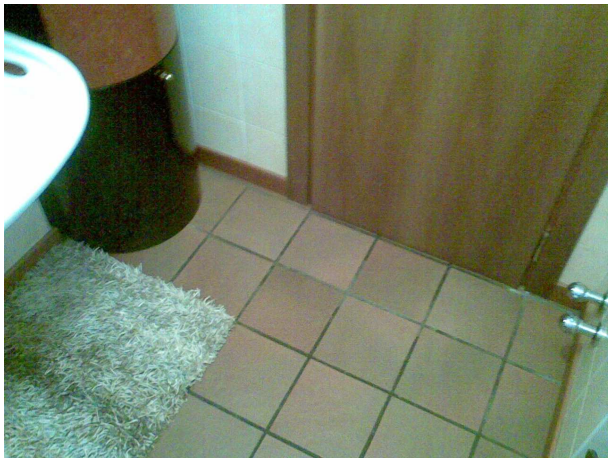


Figura 5.3: Immagine ottenuta della stanza presentata in foto in Figura 5.4. In alto si nota un errore nei dati ricevuti dal sensore.



(a) Dettaglio di Figura 5.3. L'oggetto nero che si trova tra le due pareti della stanza non è riconosciuto dal sensore.



(b) Foto della stanza considerata

Figura 5.4: Esempio in dettaglio

Capitolo 6

Realizzazioni sperimentali e valutazione

"Each of us is confined to a world of our own making."

Shawn Mikula

Per verificare la validità delle idee esposte è stato realizzato un pacchetto software costituito da una libreria sviluppata in C++. La libreria si divide logicamente in due parti: *quadtree* e *node*. La sezione *node* è inerente alla gestione dei nodi e della struttura dati quadtree a basso livello. Sono presenti metodi per l'inserimento dei punti e di flag dei verifica, il distruttore dell'albero, il metodo per la valutazione di **occprob**, le funzioni **score** usate negli *scan matcher* e altri metodi di utilità.
Il file *quadtree* opera a un livello più alto e si occupa dell'inserimento dei punti sfruttando l'algoritmo di Bresenham, la lettura dei dati dal sensore o log file, lo *scan matching* e di casi meno frequenti come l'ingrandimento dinamico della mappa. Si è inoltre creato un ambiente virtuale per simulazioni costituito da una stanza rettangolare e implementato sotto il nome di *virtualortoscan*.

6.1 Funzionamento

Per un corretto funzionamento, spiegato in modo dettagliato nel manuale utente, è sufficiente richiamare la libreria quadtree, istanziarne un oggetto e utilizzare i metodi pubblici per inserire dati, eseguire una versione dello *scan matcher*, ottenere l'output testuale. Questo può essere convertito in immagine attraverso la libreria `ImageMagick++` utilizzando un semplice script `bash`.

6.2 Risultati

È stato verificata il corretta generazione della mappa con pose del robot nota sia in simulazione che sperimentalmente. Come spiegato nei precedenti capitoli, per ottenere una mappa consistente si è reso necessario implementare l'algoritmo di Bresenham[1] e una maschera; quest'ultima impedisce l'aggiornamento dei quadrati aventi un lato o spigolo in comune con il quadrato aggiornato.
Vengono ora presentate alcune immagini ottenute, opportunamente commentate.
Gli output ottenuti in ambiente simulato con *pose* nota sono mostrati in Figura 6.1 e Figura 6.2.
In Figura 6.1 il robot è fermo in un punto pressoché centrale della mappa ed è sempre contraddistinto da un quadratino rosso. L'inserimento è stato effettuato in senso orario a punti equidistanti dal robot. Nell'immagine presentata si può osservare il corretto inserimento dei punti e il colore chiaro, ovvero zona a bassa probabilità di essere occupata, dei quadrati interni alla stanza circolare. In Figura 6.2 è visibile la mappa generata da due scansioni complete. Il robot è stato spostato istantaneamente da un punto nel quadrato grande in alto a sinistra a quello in basso a destra. Il laser simulato presenta una risoluzione di un grado tra una lettura e la successiva. Si possono notare i contorni più definiti laddove il robot era vicino al 'muro' e posto quasi frontalmente. I contorni della stanza appaiono molto scuri in tutto il perimetro grazie alla maschera applicata.

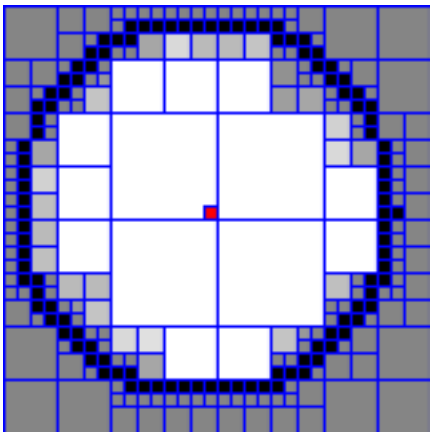


Figura 6.1: ultima immagine della sequenza di Figura 4.3

[1] Per dettagli si rimanda a: http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

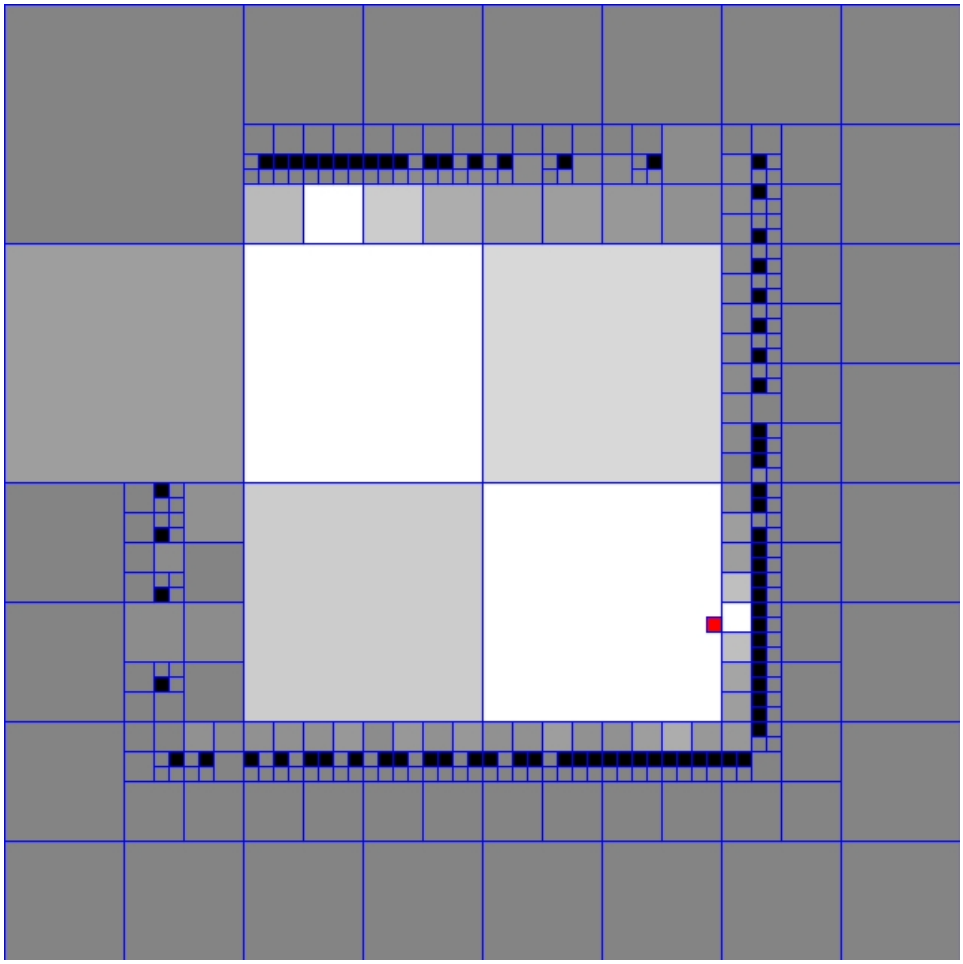


Figura 6.2: mappa generata da due scansioni in una stanza quadrata

Le Figure 6.4 e 6.5 si riferiscono a *mapping* con sensore fermo in ambienti reali.

Come si può osservare, in tutti i casi la mappa generata può ritenersi corretta. Il sensore, mantenuto nella stessa posizione durante la lettura, ha permesso di concentrarsi sul *mapping* disaccoppiando lo *scan matcher* e rimandando ad un secondo tempo la risoluzione del problema.

Nelle Figure da 6.6 a 6.15 sono mostrate le immagini ottenute utilizzando gli *scan matcher* di tipo *greedy* e le relative funzioni di **score** proposte nei capitoli precedenti, analizzate nei dettagli implementativi in Appendice B.

Figura 6.3: Foto della scatola usata con oggetti al suo interno

Le mappe ottenute sono nella maggior parte dei casi[2] consistenti e di buona qualità. La funzione di **score** basata sulla distanza permette di ottenere risultati migliori ma, come si intuisce confrontando i tempi di esecuzione, è più lenta. La complessità computazionale della prima implementazione proposta non permette ancora un utilizzo in real time. Una seconda variante, basata su una euristica, rende molto più veloce il procedimento ma non si dimostra altrettanto affidabile. La **score** basata sulla valutazione diretta di **occprob** si rivela poco utile per i motivi delineati nei precedenti capitoli, tuttavia il suo tempo di esecuzione rimane il migliore.

6.3 Complessità

Viene di seguito fornita una tabella che riassume le complessità e i tempi di esecuzione dell'algoritmo.

	$Complessita'$	$TempoMedioEsecuzione$
$AumentoDimensioneMappa$	$\Theta(1)$	
$InserimentoPuntoEBresenham$	$\Theta(log(Nodi))$	$< 0.5ms$
$ScanMatching1$	$\Theta(score)$	$> 0.5s$
$ScanMatching2$	$\Theta(score)$	$< 0.3 \mid > 0.5s$
$score\_distance$	$\Theta(puntiacquisiti * Nodi)$	
$score\_occprob$	$\Theta(puntiacquisiti * log(Nodi))$	
$score\_nearestdist$	$\Theta(puntiacquisiti * log(Nodi))$	

[2] si fa riferimento soprattutto ai casi reali in ambienti piccoli, come le scatole

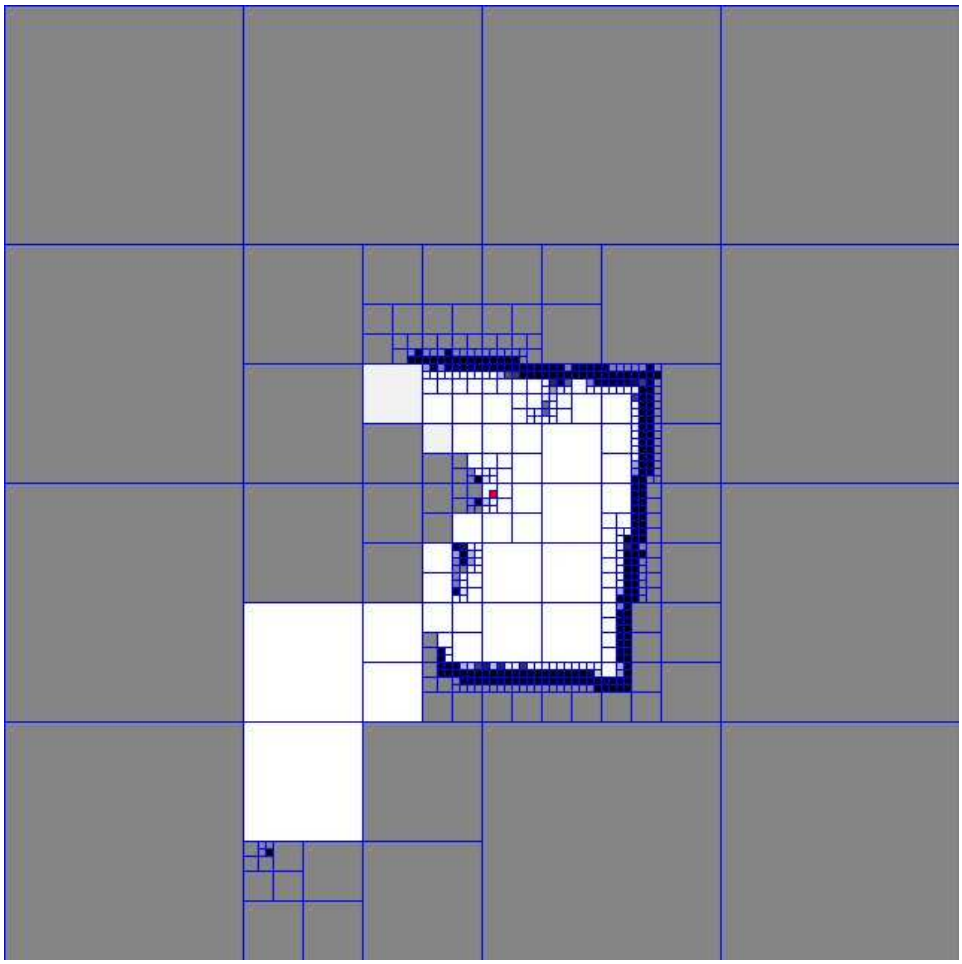


Figura 6.4: Scatola di un monitor nella quale è stato posto il sensore e mantenuto fermo durante le scansioni. I contorni sono quasi del tutto regolari. In alcune zone sono stati posti oggetti, come piccole scatole, di cui si riconosce la sagoma.

I valori di tempo medio per le funzioni di *scan matching* dipendono molto dalla complessità della **score** usata e dalla grandezza della mappa. Il computer di riferimento per valori è: AMD Athlon XP 2800, 512Mb RAM, S.O. Archlinux.
È importante osservare che lo *scan matcher greedy* implementato dipende dal numero di punti acquisiti nello *scan* oltre che da parametri che possono considerarsi costanti. Pertanto, avere un sensore che acquisisce molti più dati in una sola scansione rallenta il processo di confronto di questi con la mappa; tuttavia potrebbe aumentare la velocità di ricerca del massimo della funzione **score** nonché la precisione di confronti successivi avendo incorporato numerosi punti nella mappa globale. Sono presentate ora le mappe

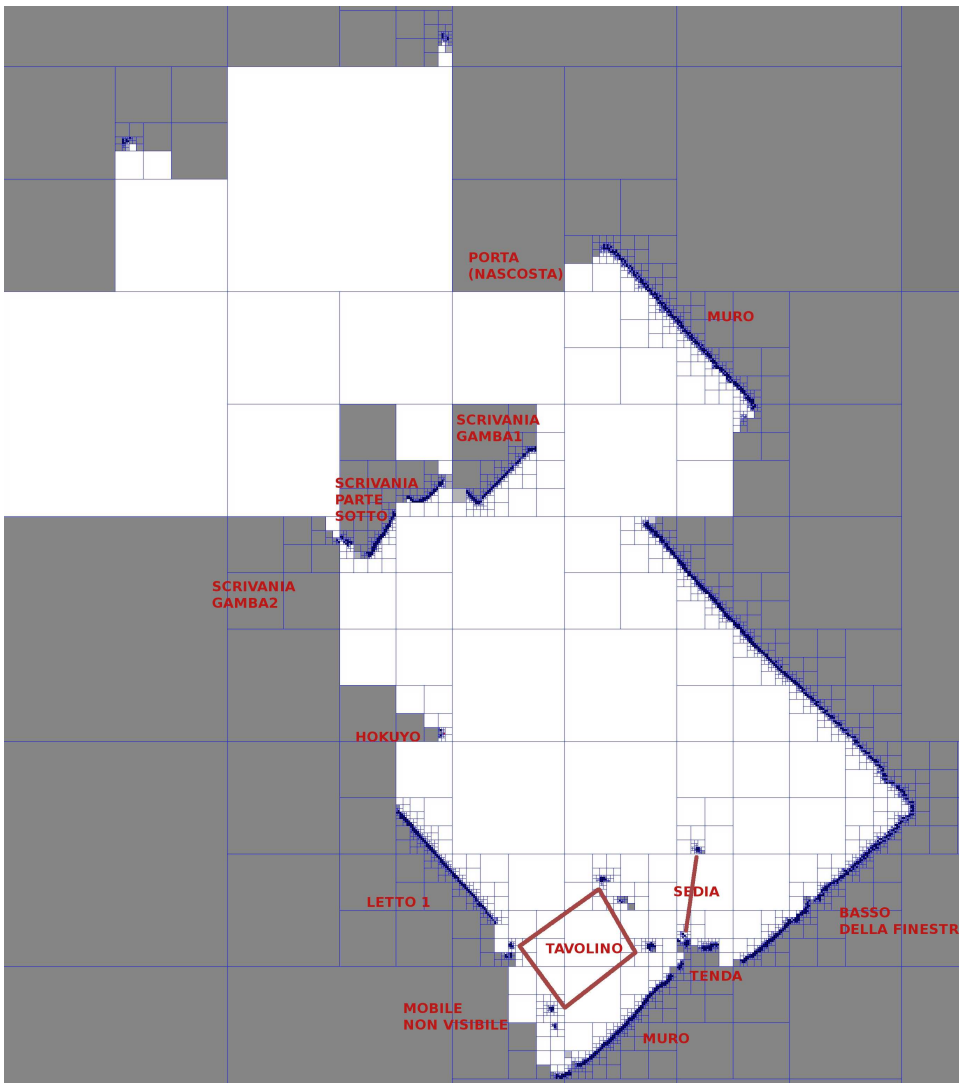


Figura 6.5: La stanza con sensore fermo è ben riconoscibile.

generate dagli algoritmi proposti. Nelle didascalie sono citati i nomi dei metodi utilizzati.

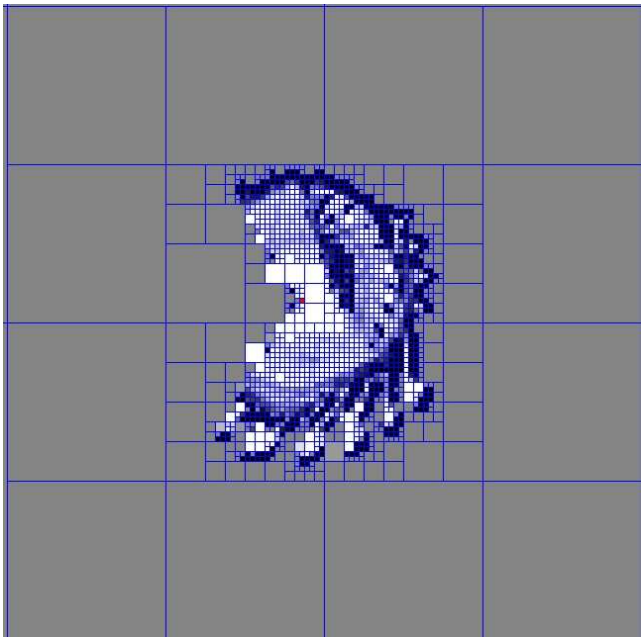


Figura 6.6: Immagine di una scatola senza scan matching inserendo i dati nella mappa con sensore supposto fermo. Sono stati impiegati 80 file log

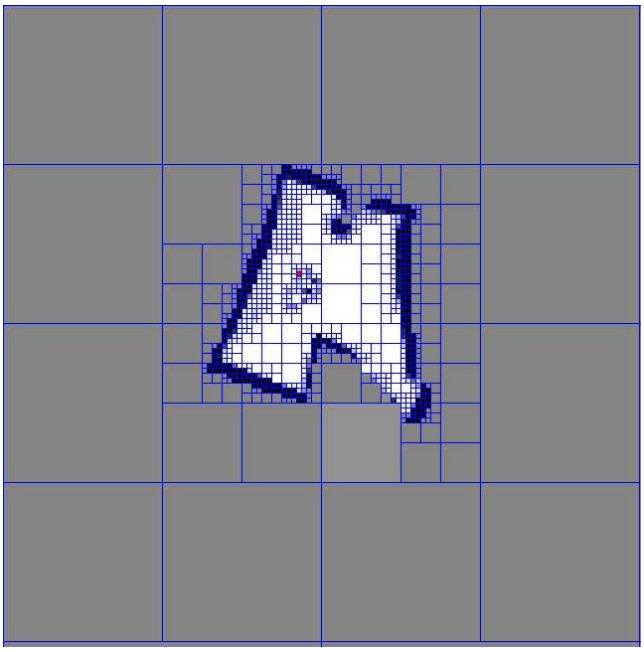


Figura 6.7: Stessi dati di Figura 6.6 ottenuti con **Scanmatcher1 score_distance**

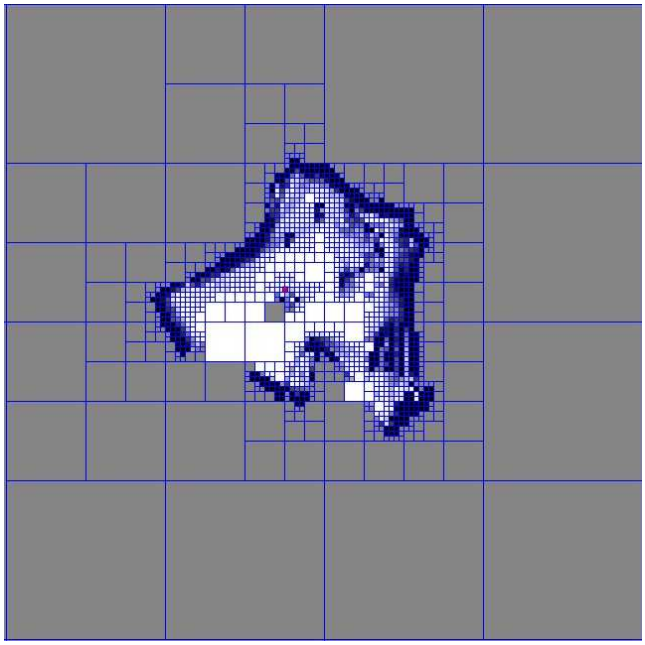


Figura 6.8: Stessi dati di Figura 6.6 ottenuti con **Scanmatcher2 score_distance**

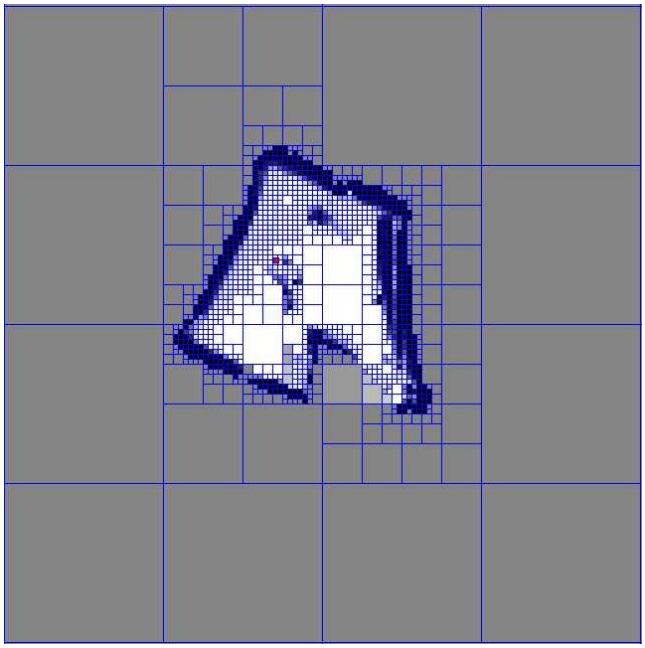


Figura 6.9: Stessi dati di Figura 6.6 ottenuti con **Scanmatcher1 score_nearestdist**

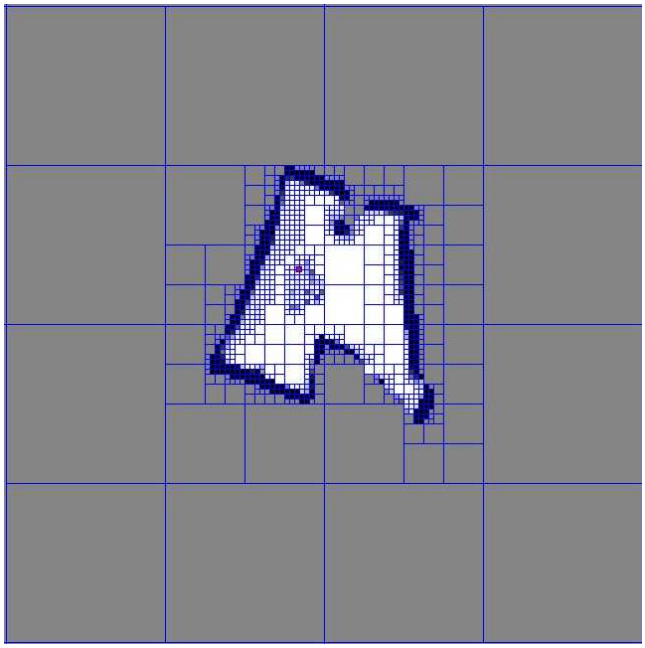


Figura 6.10: Stessi dati di Figura 6.6 ottenuti con **Scanmatcher1 score_occprob**

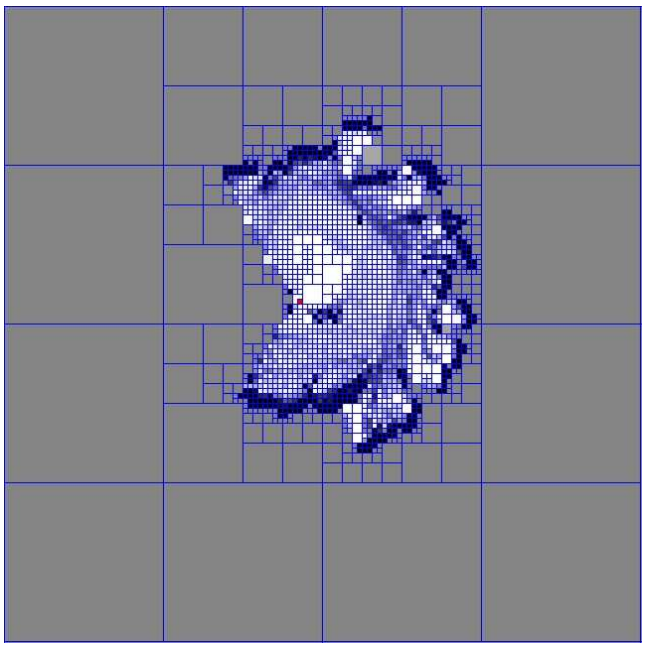


Figura 6.11: Immagine di una scatola senza scan matching ottenuta attraverso un secondo dataset. Il sensore è stato spostato all'interno della scatola in modo differente.

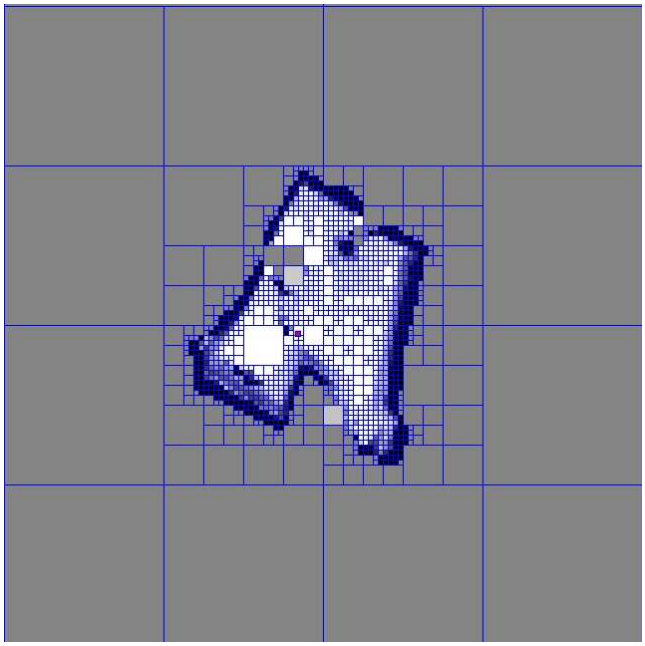


Figura 6.12: Stessi dati di Figura 6.11 ottenuti con **Scanmatcher1 score_distance**

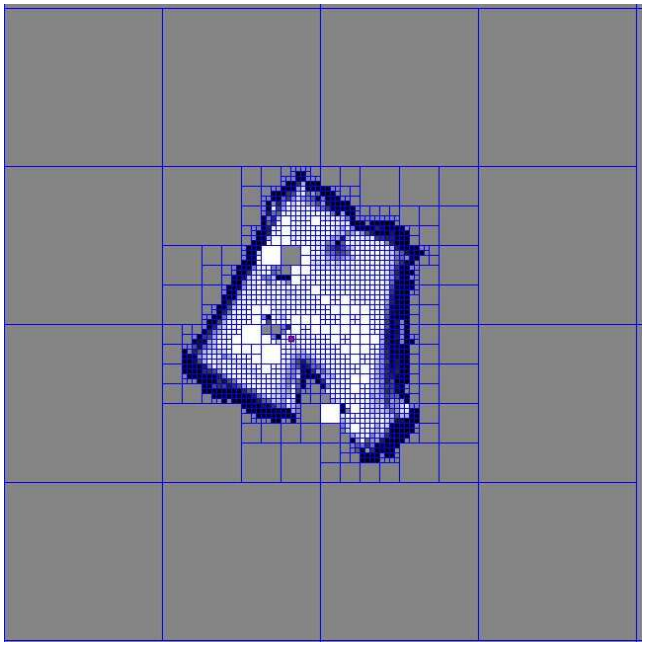


Figura 6.13: Stessi dati di Figura 6.11 ottenuti con **Scanmatcher2 score_distance**

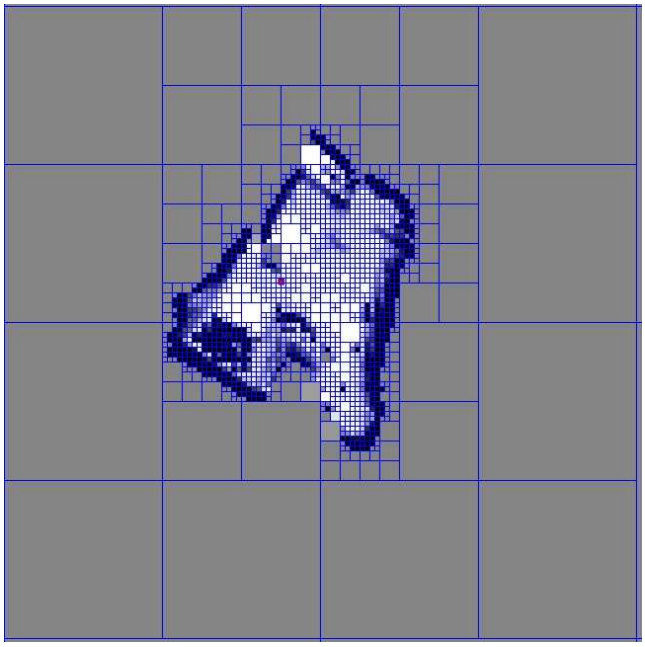


Figura 6.14: Stessi dati di Figura 6.11 ottenuti con **Scanmatcher1 score_nearestdist**

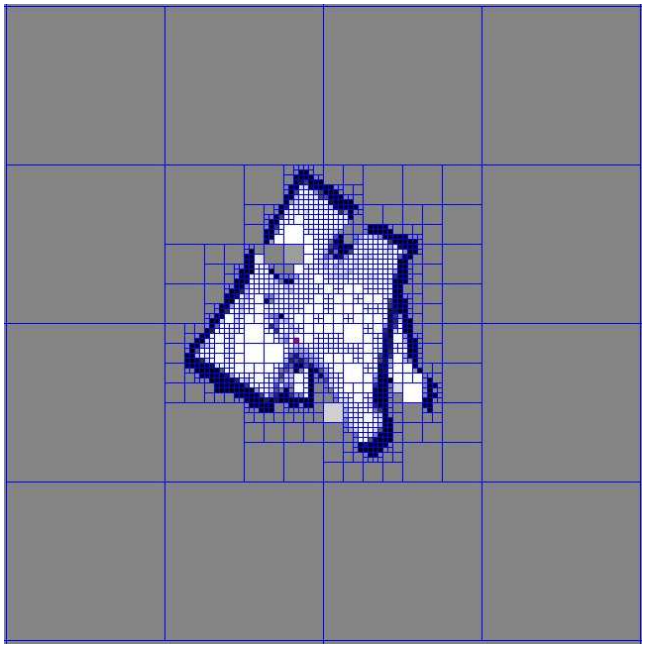


Figura 6.15: Stessi dati di Figura 6.11 ottenuti con **Scanmatcher1 score_occprob**

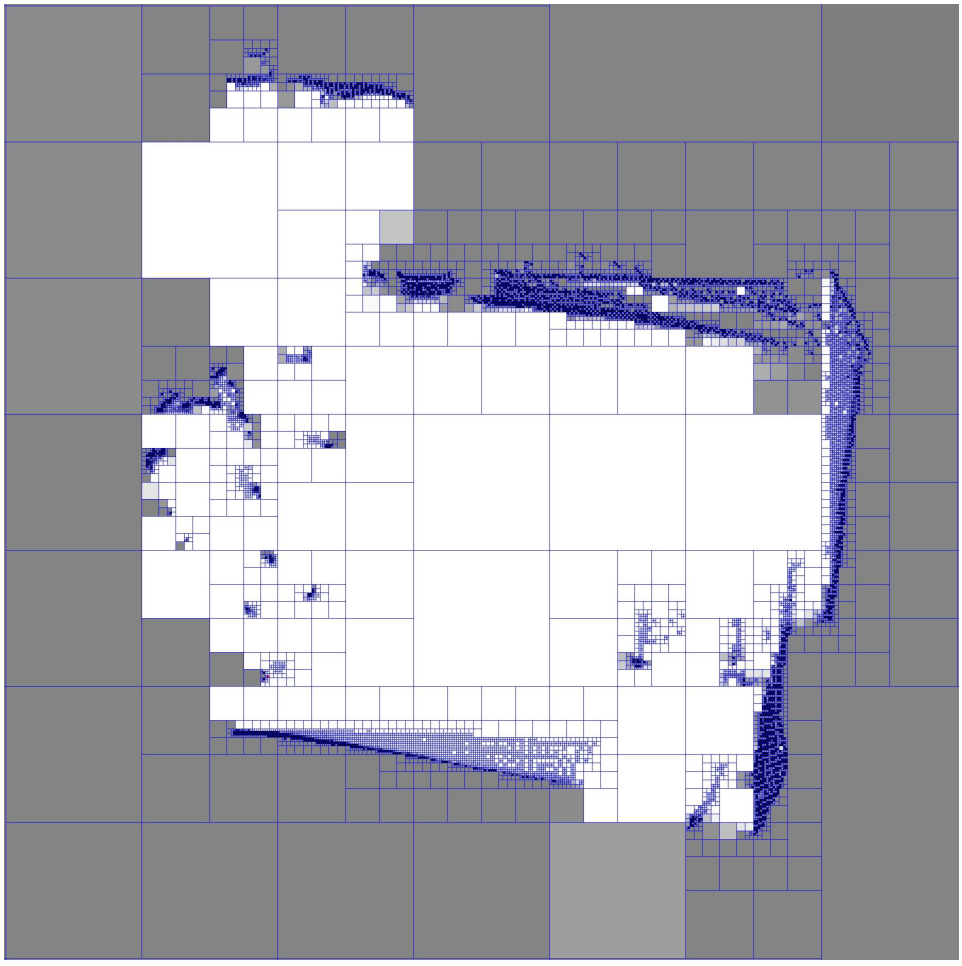


Figura 6.16: Immagine di una stanza senza scan matching. Sono stati impiegati 30 file log. Il sensore è stato spostato ad ogni iterazione.

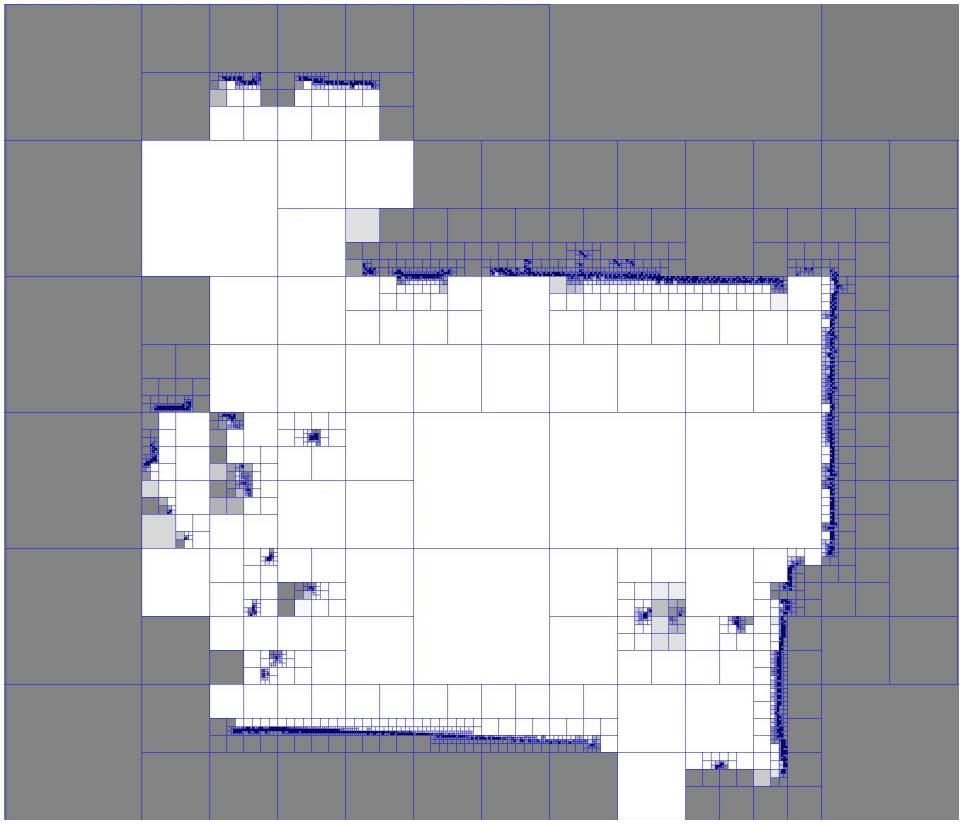


Figura 6.17: Stessi dati di Figura 6.16 ottenuti con **Scanmatcher1 score_distance**. *È necessario osservare come un buon metodo di scan matching riduca il numero di celle nella mappa e di conseguenza lo spazio su disco occupato.*

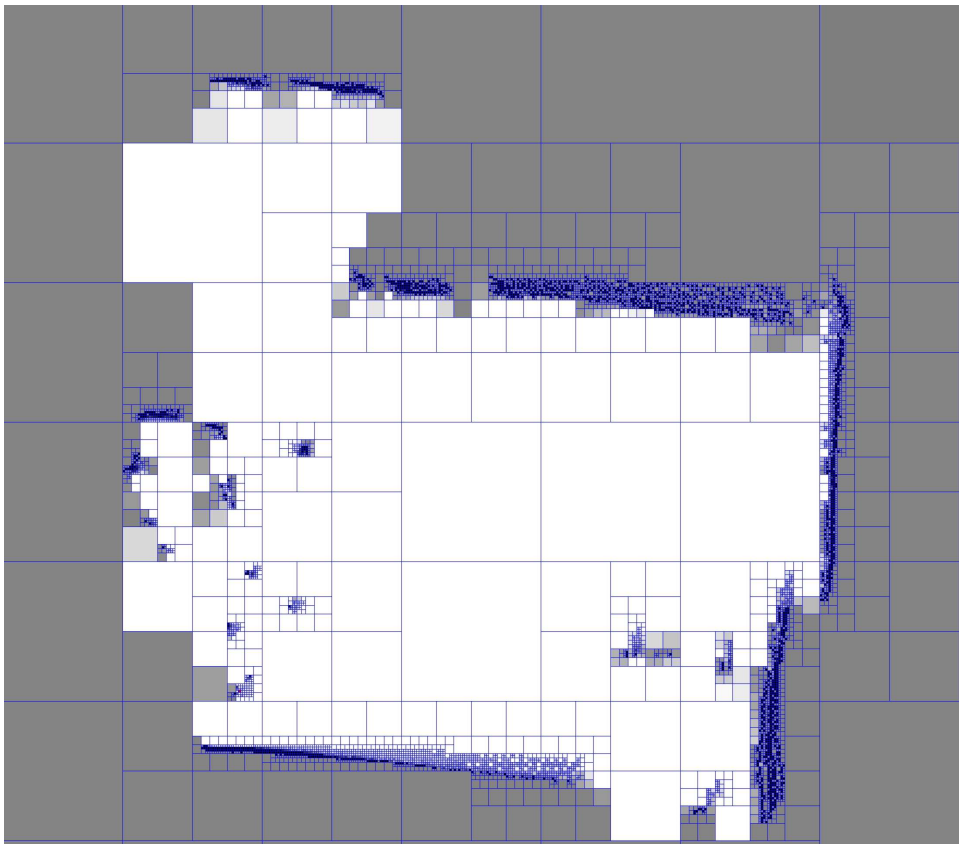


Figura 6.18: Stessi dati di Figura 6.16 ottenuti con **Scanmatcher2 score_distance**

Capitolo 7

Direzioni future di ricerca e conclusioni

> *"Life is like a game of cards. The hand that is dealt you represents determinism; the way you play it is free will."*
>
> Jawaharlal Nehru

Il lavoro affrontato ha permesso di porre l'accento su problemi che si possono incontrare solamente quando ci si accinge alla realizzazione di un software che debba funzionare in ambienti reali e sfrutti l'architettura hardware imposta, nonché il tipo di sensore con la risoluzione e gli errori caratteristici. Il risultato ottenuto mette in luce la difficoltà della costruzione di una mappa consistente e della realizzazione di uno *scan matcher* adeguato (che ottenga buoni risultati con minor costo computazionale possibile). Le immagini presentate nel corso dei capitoli mostrano come, dopo numerosi sforzi, si sono potute ottenere mappe di buona qualità.
La tesi si presta ad essere sviluppata in più di una direzione:

- l'estensione spaziale della mappa generata da 2D a 3D possibile grazie ad un supporto basculante su cui può essere posto il sensore laser, e lo studio della la legge di modo per generare l'octree corrispondente;
- la ricerca di una funzione di **score** basata sulla distanza che ottenga gli stessi risultati della migliore presentata in questi capitoli ma che sfrutti maggiormente la struttura dati come albero (per un utile riferimento considerare il metodo `score_nearestdist`).
- l'integrazione di un metodo più efficace per SLAM, ad esempio quelli presentati nel Capitolo 2;

- l'utilizzo di file in formato CARMEN[1] e la possibilità di far interagire il pacchetto software elaborato nell'ambiente di simulazione di CARMEN e verificare la correttezza della mappa e dello SLAM al variare dei parametri del robot e del sensore utilizzato.
- la migrazione dalla libreria `ImageMagick++`, considerata troppo lenta nel processamento di immagini con risoluzione maggiore di 2000x2000px, a OpenGL[2] per facilitare il passaggio al 3D.

[1]Carnegie Mellon Robot Navigation Toolkit, rilasciato sotto licenza GPL consultabile alla pagina http://carmen.sourceforge.net/. È compatibile coi *dataset* presenti in http://radish.sourceforge.net/. Altro progetto degno di nota è OpenSlam: http://openslam.org/

[2]http://www.opengl.org/

Capitolo 8

Documentazione del progetto logico

"Truthful words are not beautiful; beautiful words are not truthful.
Good words are not persuasive; persuasive words are not good."

Lao tzu

In questo capitolo si spiega il progetto software realizzato ad alto livello. Si è cercato di attenersi il più possibile allo standard UML 2.0 per rendere la documentazione fruibile ad pubblico più ampio.
Lo *use case* rappresentato in Figura 8.1 mostra l'interazione tra il robot e il prodotto software sviluppato. Si può constatare una divisione logica del software dei compiti di inserimento dati a basso livello e di *mapping*. L'inserimento dati può essere affiancato dalla funzione logica *scan matcher*, la quale si occupa di inserire i valori in modo che la mappa risulti consistente migliorando la *pose* del robot.

In Figura 8.2 è mostrato un *component diagram* che spiega come la suddivisione dei file in `node.cpp` e `quadtree.cpp` sottintenda una distinzione logica tra funzioni primarie del quadtree nel primo file, ed accessorie, nel secondo.
Lo schema in Figura 8.3, distaccandosi dallo standard UML, vuole esemplificare il principio di funzionamento logico del sistema;
Il sensore fornisce un insieme di dati, cioè, in senso lato, una mappa locale del robot. Nella generazione della mappa globale concorrono: la funzione logica che permette di aumentare la mappa in caso di necessità e quella di inserimento che sfrutta Bresenham per inserire i dati nella mappa globale (nota la *pose*). Lo *scan matcher* confronta la mappa locale ottenuta tramite

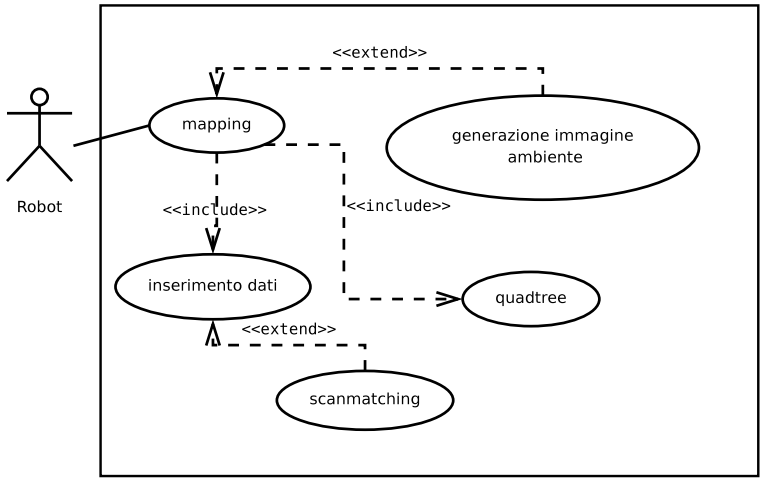


Figura 8.1: Use case

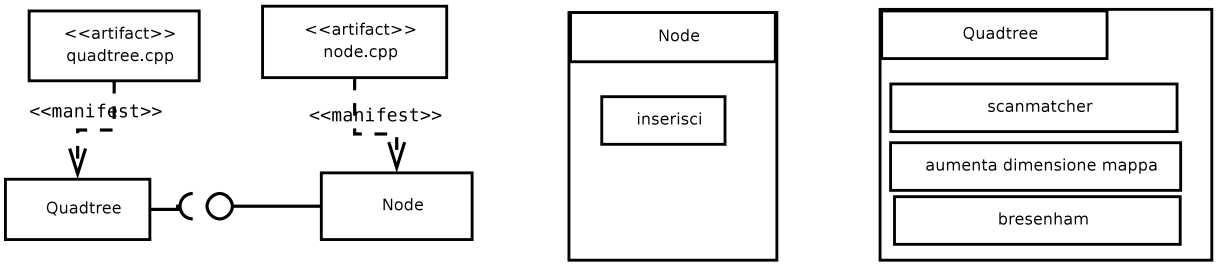


Figura 8.2:

l'ultima scansione con quella globale, restituendo la *pose* più probabile del robot, ossia quella che massimizza la $p(x_t|z_t, m)$.

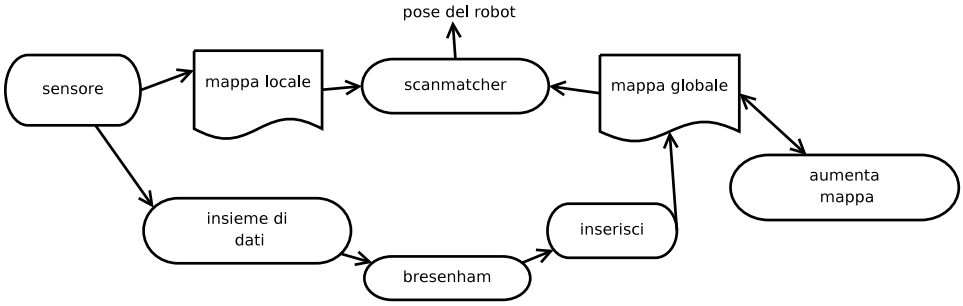


Figura 8.3:

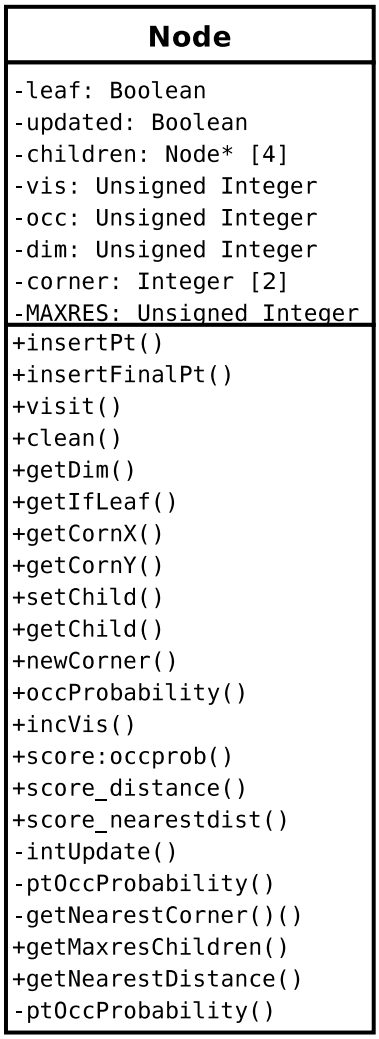
Capitolo 9

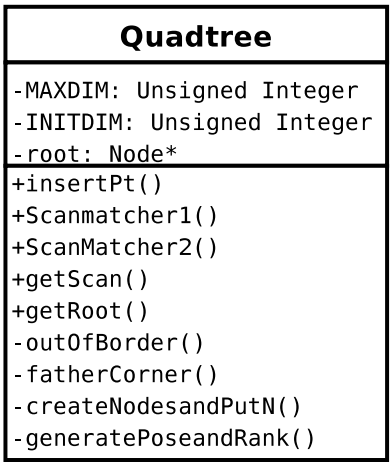
Documentazione della programmazione

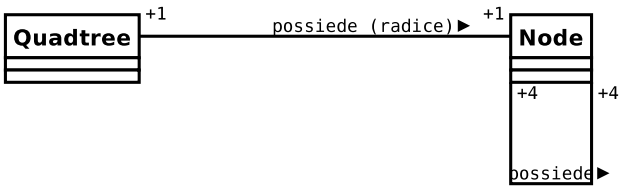
"Art is dangerous. It is one of the attractions: when it ceases to be dangerous, you don't want it."

Duke Ellington

In questo capitolo si mostra tramite UML la struttura del software e delle funzioni di maggiore interesse. Di seguito è illustrato in figura il *class diagram* con la maggior parte delle funzione membro utilizzate. Si è scelto di non scrivere gli argomenti delle funzioni membro per questioni di leggibilità.

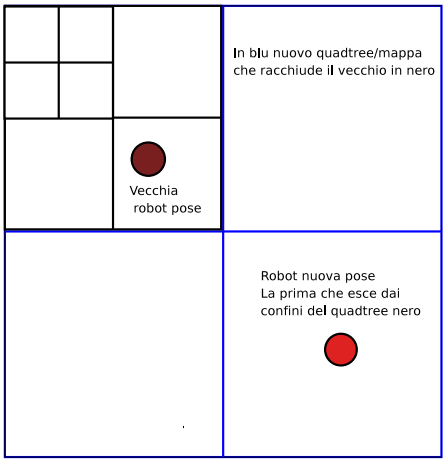




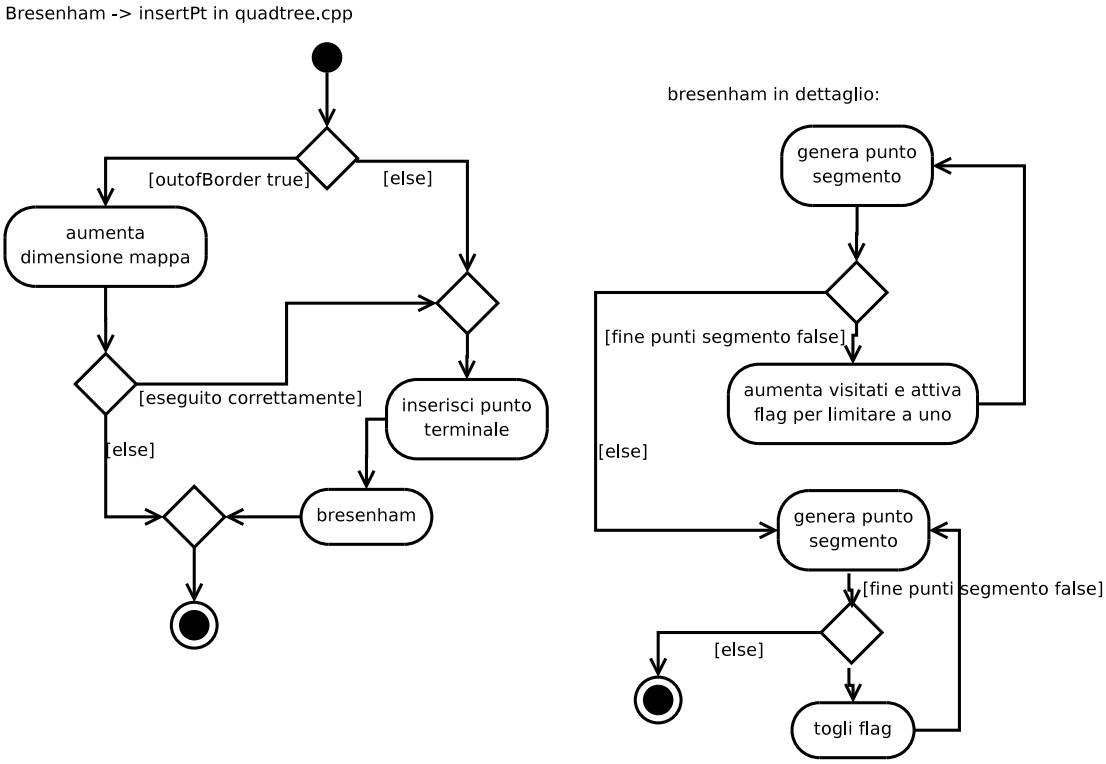


Sono di seguito mostrate più in dettaglio alcune funzioni presenti nel precedente *class diagram*.

Si è scelto di spiegare graficamente il funzionamento di **outofborder** per semplicità espositiva. Si noti come, a seconda della prima *pose* del robot che esce dai confini del quadtree, si sceglie di ingrandire la mappa:

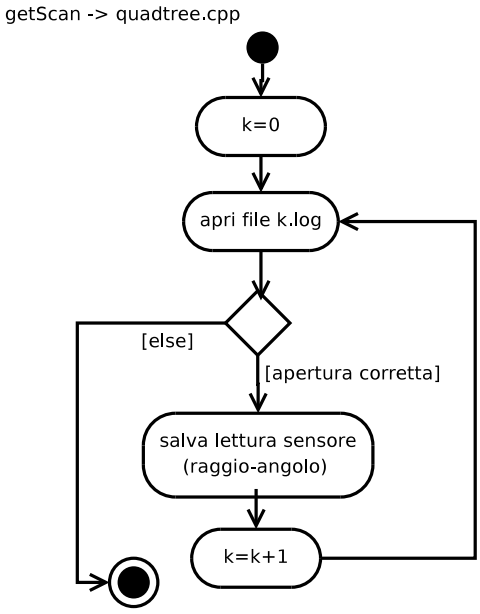


L'*activity diagram* mostra il comportamento di **insertPt** presente in `quadtree.cpp`. Come si evince dal diagramma, la funzione richiama a sua volta l'algoritmo di Bresenham che si occupa di aggiornare le probabilità delle celle intersecate dal segmento *pose* robot - valore restituito dal sensore laser.



getScan si occupa di restituire al programma la scansione numero k. L'indice può essere utilizzato per rappresentare il tempo discreto, ovvero $T = T_s * k$

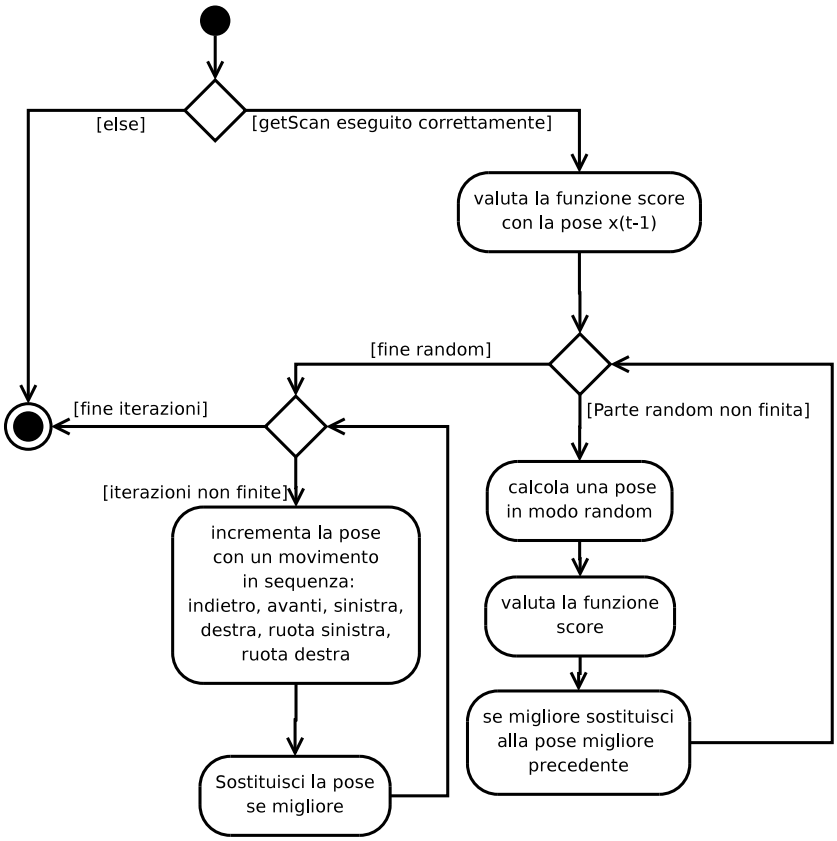
dove T_s =periodo di campionamento, oppure per indicare il nome dei file log considerati:



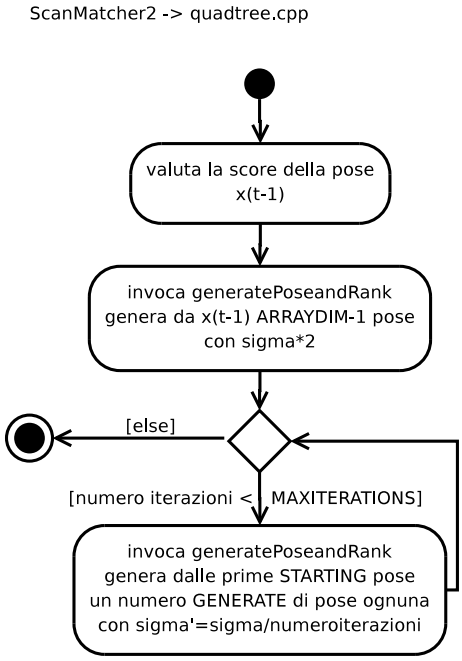
Scanmatcher1 è l'implementazione di un algoritmo *greedy* per lo *scan matching* che sfrutta **score**[1]:

[1]le score sono intercambiabili l'una con l'altra. Allo stesso modo è possibile scegliere Scanmatching1 o Scanmatching2

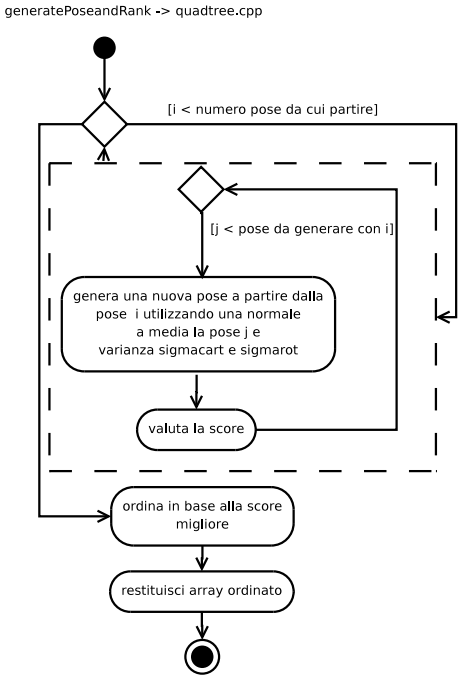
Scanmatcher1 -> quadtree.cpp



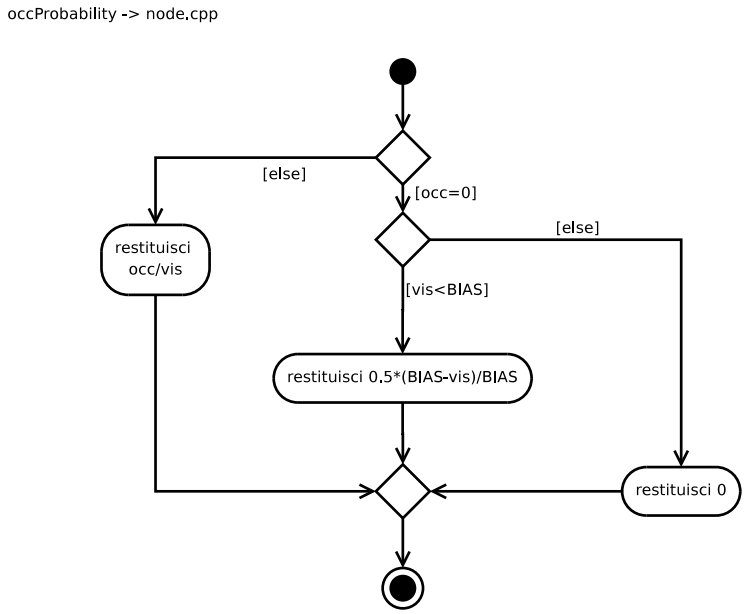
Scanmatcher2 è un'implementazione che sfrutta la libreria `GNU Scientific Library` per generare campioni da gaussiane e, da questi, nuove *pose*:



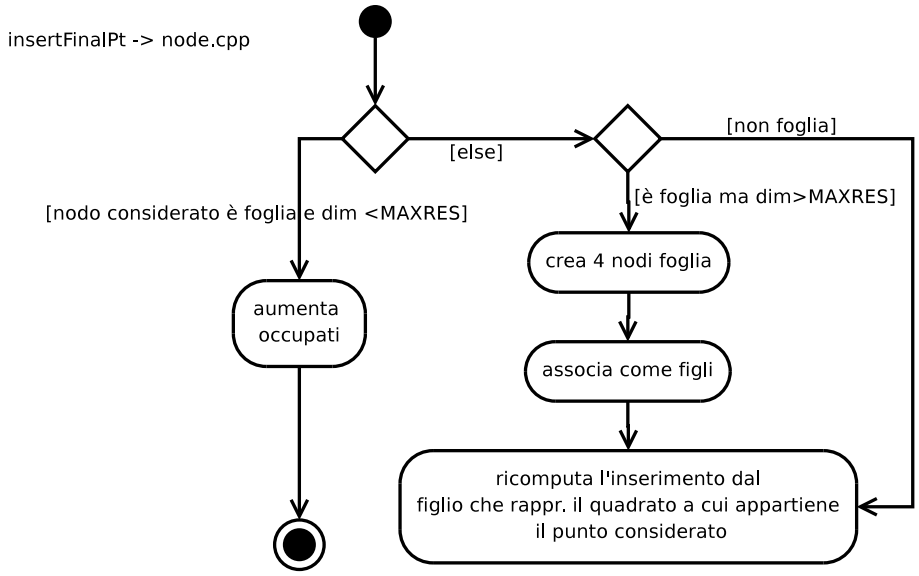
generatePoseandRank è una funzione di utilità per **Scanmatcher2**:



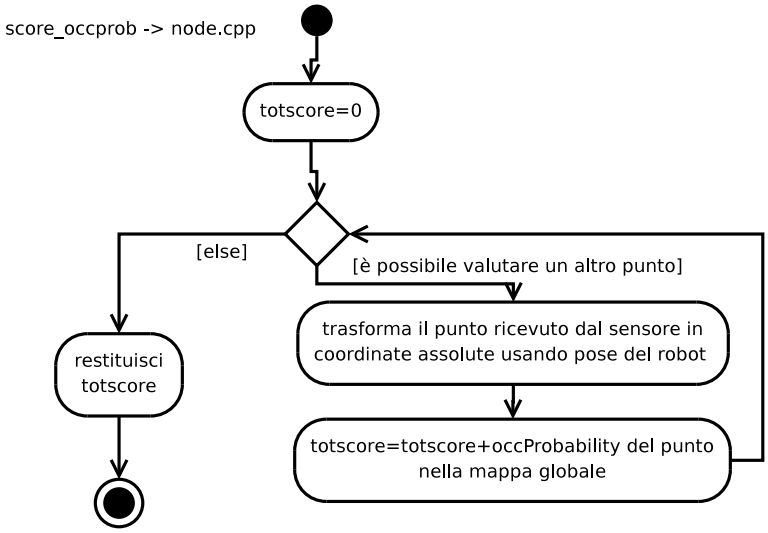
Alcune delle funzioni presenti in `node.cpp`:
occProbability restituisce il valore della probabilità della cella di essere occupata in base ai valori **occ** e **vis** (occupato, visto/visitato):



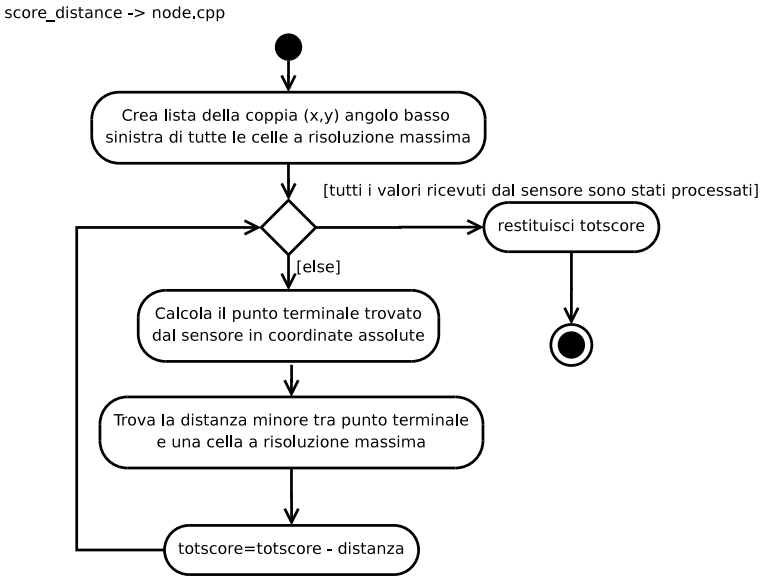
insertFinalPt è utilizzato prima di Bresenham per creare i sotto-rami fino a profondità pari a **MAXRES** (risoluzione massima) e aggiornare l'**occ** della cella alla quale appartiene il punto terminale, fornito dal sensore:



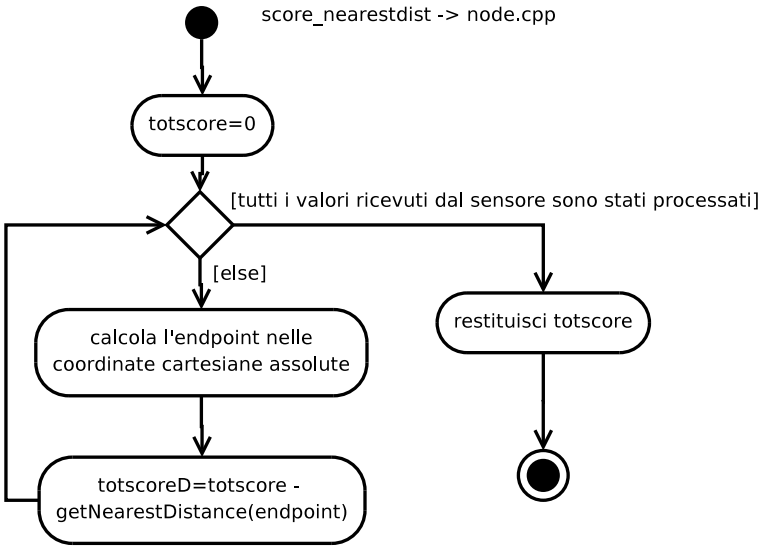
score_occprob rappresenta la funzione da massimizzare nell'algoritmo di *scan matching* e si basa direttamente sul valore **occprob** presente nelle celle della mappa:



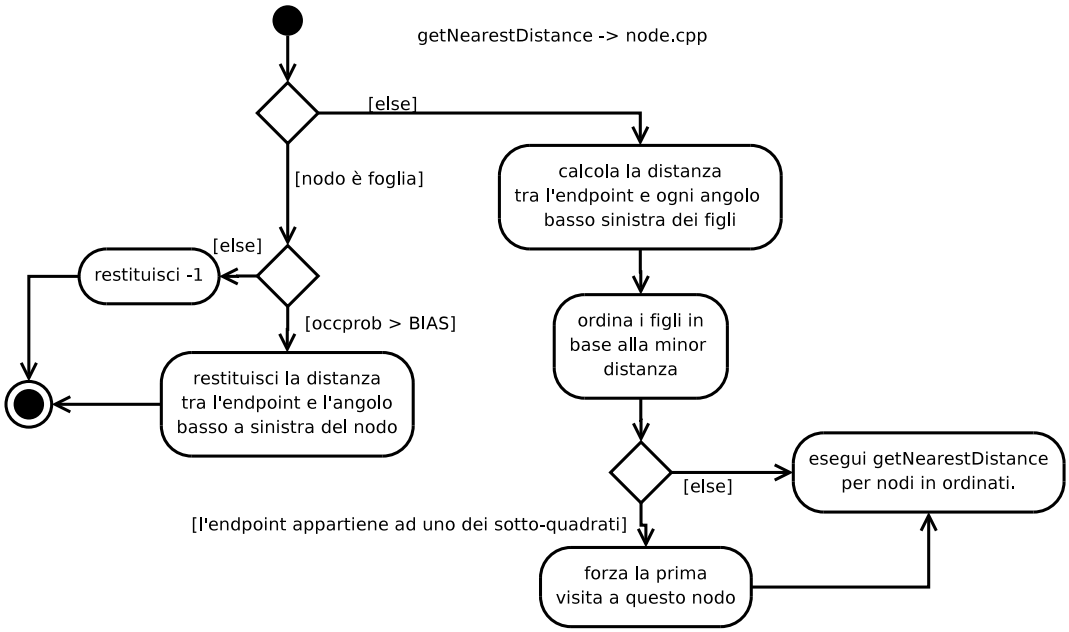
score_distance è una variante della funzione **score**. Si basa sulla distanza ma necessita la visita dell'intero quadtree per reperire la lista di angoli utilizzata nel confronto per il calcolo della distanza minima:



score_nearestdist utilizza un'euristica per ottenere un risultato simile a **score_distance** impiegando direttamente la struttura dati ad albero per ricercare la distanza minima:



getNearestDistance è la funzione invocata in **score_nearestdist**:



Bibliografia

[1] *A. Elfes. Using Occupancy Grids for Mobile Robot Perception and Navigation*, volume 22, Los Alamitos, CA, USA, 1989. IEEE Computer Society Press.

[2] *A. Fitzgibbon. Robust registration of 2D and 3D point sets. In Proc. British Machine Vision Conference, volume II, pages 411–420, Manchester, UK, September 2001.*, 2001.

[3] *P. Biber and W. Strasser. nScan-Matching: Simultaneous Matching of Multiple Scans And Application to SLAM. in Proc. of Intern. Conference on Robotics and Automation (ICRA'06), 2006*, May 15-19 2006.

[4] D. ahnel, D. Schulz, and W. Burgard. Map building with mobile robots in populated environments. in proc. of the ieee/rsj int. conf. on intelligent robots and systems (iros), 2002., 2002.

[5] P. Bhattacharya. Efficient neighbor finding algorithms in quadtree and octree. Master's thesis, 2001. M.T. Thesis, Dept. Comp. Science and Eng., India Inst. Technology, Kanpur.

[6] P. Biber and W. Straer. The normal distributions transform: A new approach to laser scan matching. in proceedings of the 2003 ieee/rsj international conference on intelligent robots and systems (iros'03), 2003. 2003.

[7] A. Censi, L. Iocchi, and G. Grisetti. Scan matching in the hough domain, in proc. of intern. conference on robotics and automation (icra'05), 2005. Technical report, 2005.

[8] C. Connolly. Cumulative generation of octree models from range data. In *Robotics and Automation. Proceedings. 1984 IEEE International Conference on*, volume 1, pages 25–32, Mar 1984.

[9] Y. Kang D. Caveney and J.K. Hedrick. Probabilistic mapping for unmanned rotorcraft using point-mass targets and quadtree structures. In *Proc. of ASME IMECE Orlando, FL, Nov.*, 2005.

[10] Mr. Aluir Porfino Dal Poz Edinéia Aparecida dos Santos Galvanin and Ms. Aparecida Doniseti Pires Souza. Laser scanning data segmentation in urban areas by a bayesian framework, Shaping the Change, XXIII FIG Congress, Munich, Germany,October 8–13 2006.

[11] D. Eppstein, M. T. Goodrich, and J. Z. Sun. The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data. *ArXiv Computer Science e-prints*, July 2005.

[12] S. Faibish and M. Abramovitz. Perception and navigation of mobile robots. In *Intelligent Control, 1992., Proceedings of the 1992 IEEE International Symposium on*, pages 335–340, 11-13 Aug. 1992.

[13] G. Grisetti, C. Stachniss, and W Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters, 2006.

[14] J. Gutmann and C. Schlegel. Amos: Comparison of scan matching approaches for self-localization in indoor environments. 1996.

[15] D. Jung and K. Gupta. Octree-based hierarchical distance maps for collision detection. Technical report, 1997.

[16] Subbarao Kambhampati and L. Davis. Multiresolution path planning for mobile robots. Robotics and Automation, IEEE Journal of [legacy, pre - 1988] 1986.

[17] Gerhard K. Kraetzschmar, Guillem Pagès Gassull, and Klaus Uhl. Probabilistic quadtrees for variable-resolution mapping of large environments. In M. I. Ribeiro and J. Santos Victor, editors, *Proceedings of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles*, Lisbon, Portugal, July 2004. Elsevier Science.

[18] F. Lu and E. Milios. Robot pose estimation in unknown environments by matching 2d range scans, 1994.

[19] P. Payeur. Dealing with uncertain measurements in virtual representations for robot guidance. In *Virtual and Intelligent Measurement Systems, 2002. VIMS '02. 2002 IEEE International Symposium on*, pages 56–61, 19-20 May 2002.

[20] P. Payeur, P. Hebert, D. Laurendeau, and C.M. Gosselin. Probabilistic octree modeling of a 3d dynamic environment. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 2, pages 1289–1296vol.2, 20-25 April 1997.

[21] P. Payeur, D. Laurendeau, and C.M. Gosselin. Range data merging for probabilistic octree modeling of 3d workspaces. In *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, volume 4, pages 3071–3078vol.4, 16-20 May 1998.

[22] Ioannis M. Rekleitis. A particle filter tutorial for mobile robot localization. (TR-CIM-04-02), 2004.

[23] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[24] Vivek Anand Sujan, Marco Antonio Meggiolaro, and Felipe Augusto Weilemann Belo. A new technique in mobile robot simultaneous localization and mapping. *Sba: Controle & Automação Sociedade Brasileira de Automatica*, 17:189 – 204, 06 2006.

[25] S. Thrun. Robotic mapping: A survey. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millenium*. Morgan Kaufmann, 2002. to appear.

[26] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT Press, 2005.

[27] Zezhong Xu, Jilin Liu, Zhiyu Xiang, and Han Li. Map building for indoor environment with laser range scanner. In *Intelligent Transportation Systems, 2002. Proceedings. The IEEE 5th International Conference on*, pages 136–140, 2002.

Appendice A

Listato

Di seguito sono riportati in formato cartaceo i file:

- `node.h`
- `node.cpp`
- `quadtree`.h
- `quadtree.cpp`
- `main.cpp`

node.h

```
/*************************************************************************
 *  Copyright (C) 2007 by Mauro Brenna, Ivan Reguzzoni                   *
 *  malloblenne@gmail.com                                                *
 *                                                                       *
 *  This program is free software; you can redistribute it and/or modify *
 *  it under the terms of the GNU General Public License as published by *
 *  the Free Software Foundation; either version 3 of the License, or    *
 *  (at your option) any later version.                                  *
 *                                                                       *
 *  This program is distributed in the hope that it will be useful,      *
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of       *
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the        *
 *  GNU General Public License for more details.                         *
 *                                                                       *
 *  You should have received a copy of the GNU General Public License    *
 *  along with this program;If not, see <http://www.gnu.org/licenses/>.  *
 *                                                                       *
 *************************************************************************/
#ifndef _NODE_H
#define _NODE_H
#include <iostream>
#include <vector>
#include <cmath>
class Node
{
friend std::ostream &operator<<(std::ostream &, Node &);

public:
Node(bool l,uint v,uint oc, uint d, int x, int y);
~Node();
bool insertPt (const float&xr,const float &yr, const float &xpt, const float &ypt);
bool insertFinalPt (const float &xpt, const float &ypt);
bool visit (const float &xpt, const float &ypt);
bool clean (const float &xpt, const float &ypt);
uint getDim() const {return dim;}
//int setLeaf(bool tf) { leaf=tf; }
bool getIfLeaf() const { return leaf;}
int getCornX() const {return corner[0];}
int getCornY() const {return corner[1];}
```

```
void setChild(int i,Node *n) {children[i]=n;}
Node* getChild(int pos) { return children[pos];}
void newCorner(int array[4][2]);
float occProbability() const;
void incVis(int inc=1) {vis=vis+inc;} // incrementa vis default=1
float score_occprob(const float &xr,const float &yr, const float &theta,const std::vector<float>
&r,const std::vector<float> &angle);
float score_distance(const float &xr,const float &yr, const float &theta,const std::vector<float>
&r,const std::vector<float> &angle);
float score_nearestdist(const float &xr,const float &yr, const float &theta, const std::vector<float>
&r, const std::vector<float> &angle);
private:
Node* intUpdate (const float &xr, const float &yr, const float &xpt, const float &ypt);
void getNearestCorner(const float &xpt, const float &ypt, int &cornerx,int &cornery);
void getMaxresChildren(std::vector<int> &x,std::vector<int> &y);
float getNearestDistance(const float &xpt, const float &ypt);
float  ptOccProbability (const float &xpt,const float &ypt );
bool leaf; // if leaf=true else branch
bool updated;
Node *children[4]; // i figli del nodo NW=0 NE=1 SW=2 SE=3
uint vis; //visited
uint occ; //occupied
uint dim; // dimensione del lato! (non area)
int corner[2]; // (x,y) dell'angolo basso-sinistro del quadrato
static const uint MAXRES=1; // 1 cm risoluzione max
};

#endif /* _NODE_H */
```

node.cpp

```
/***************************************************************************
 *   Copyright (C) 2007 by Mauro Brenna, Ivan Reguzzoni                    *
 *   malloblenne@gmail.com                                                 *
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 3 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 *                                                                         *
 *   You should have received a copy of the GNU General Public License     *
 *   along with this program;If not, see <http://www.gnu.org/licenses/>.   *
 *                                                                         *
 ***************************************************************************/
#include node.h
#include <cassert>
#include <iostream>
#include <iomanip>        // per decimali corretti

Node::Node(bool l,uint v,uint oc, uint d, int x, int y)
{
  leaf = l;
  vis  = v;
  occ  = oc;
  dim  = d;
  updated = false;
  corner[0] = x;
  corner[1] = y;

  for(int n=0; n<4; n++)
    {
      children[n] = NULL;
    }
}
```

```
// ---------------
Node:: ~Node()
{
  for(int n=0; n<4; n++)
    {
      delete children[n];
      children[n] = NULL;
    }
}


// ---------------
// Visualizza Node
std::ostream &operator<< (std::ostream &output, Node &n)
{
  // uncomment e comment in modo opportuno. Ora \'e abilitato x Magick++
  if ( n.getIfLeaf() ) // leaf==1
    {
//       output << (  << n.getCornX() << ,  << n.getCornY() << ,  << n.getDim()
//           << ,  << std::setprecision( 2 ) << std::setiosflags( std::ios::fixed | std::ios::showpoint
)
//             << n.occProbability() << ')';
//
//       mul=moltiplicatore x pixel (pixel/cm) [dichiarato nel cpp di magick++]
//
//       output << image.fillColor(ColorGray(<< std::setprecision( 2 )
//             << std::setiosflags( std::ios::fixed | std::ios::showpoint ) <<(1-n.occProbability() )
<<));
//          << std::endl <<image.draw( DrawableRectangle(<<(int)n.getCornX()<<*mul,<<(int)((int)n.getCornY()+(int)n.ge
//          <<*mul,<<(int)((int)n.getCornX()+(int)n.getDim())<<*mul,<<(int)n.getCornY()<<*mul
)); << std::endl;

      output << image.fillColor(ColorGray(<< std::setprecision( 2 )
            << std::setiosflags( std::ios::fixed | std::ios::showpoint ) <<(1-n.occProbability() )
<<));
        << std::endl <<image.draw( DrawableRectangle((<<(int)n.getCornX()<<-x_o)*mul,(<<(int)((int)n.getCornY()+(
        <<-y_o)*mul,(<<(int)((int)n.getCornX()+(int)n.getDim())<<-x_o)*mul,(<<(int)n.getCornY()<<-y_o)*mul
)); << std::endl;
```

```
    }

  else
    {
//      output << [1:  << *(n.getChild(0)) <<  2: << *(n.getChild(1))
//             <<  3: << *(n.getChild(2)) <<  4: << *(n.getChild(3)) << ];
      output << *(n.getChild(0)) << std::endl << *(n.getChild(1))
          << std::endl << *(n.getChild(2)) << std::endl << *(n.getChild(3)) << std::endl;
    }

  return output;
}



// ---------
float Node::occProbability() const
{
  assert (vis>=occ);
  // principalmente fa occ/vis
  // if (vis == 0) return 0.5; // presumo sia free...o si pu\'o mandare 0.5 unknown
  // idea: si usa un bias...finch\'e ho occ=0 all'aumentare di vis abbasso la prob da 0.5 seguendo
una legge



  // retta passante x due punti... tra (0,0.5) e (bias,0)
  static const uint BIAS=20; // x volte di vis sono suff affinch\'e mi convinca che sia bianco.
  if (occ==0)//ricorda caso vis=0!!
    {
     if (vis < BIAS)
       {
        // retta (0,0.5),(BIAS,0) x due punti: (y-0)/(0.5-0)= (vis-BIAS)/(0-BIAS)
        return (0.5*(BIAS-vis)/(BIAS));
       }
     else return 0;
    }
  else return ((float)occ/vis);

  }
```

```
// ---------------
// Restituisce l'occProbability di un punto dato
float Node:: ptOccProbability(const float &xpt,const float &ypt )
{
 //std::cout<<Punto considerato x ptoccprob:  << xpt <<    << ypt << std::endl;

 if (this->leaf==true)
   if (this->occProbability()<= 0.5)
     return 0;
   else
     return this->occProbability();
 else // a branch
   {
     int cornerx,cornery;
     uint tempdim=this->getDim()/2;
     for ( int i=0; i<4; i++) // x ogni figlio
       {
        // verifico sempre se nel quadrato cade il punto
        cornerx=children[i]->getCornX();
        cornery=children[i]->getCornY();
        tempdim=children[i]->getDim();
       //std::cout << dom:appartiene a  << cornerx <<    << cornery <<    << tempdim <<
std::endl;
        if ( (xpt >= cornerx) && (xpt < (cornerx+tempdim)) && (ypt >= cornery) && (ypt <
(cornery+tempdim)) )
            return children[i]->ptOccProbability(xpt,ypt);
       }
     //assert( false);                    // deve cadere in un punto del quadrato per forza
      //Sono fuori dal quadtree con questo punto. Probabilmente o \'e distante dai punti nella
mappa oppure i punti della mappa inseriti sono vicini al bordo e non posso comunque stabilire a
che probabilit\'a sono.
     return 0; // errore
   }

}


// ---------------
// Score che sfrutta il metodo ptOccProbability
float Node::score_occprob(const float &xr,const float &yr, const float &theta, const std::vector<float>
```

```
&r, const std::vector<float> &angle)
{
  //scorrendo l'indice i del vector fino a vector size
  //  1) computi l'endpoint. fai theta+angle ed \'e m=tan(quello) \'e m retta... anzi
  //      usi r*cos(angle) r*sin(angle) hai la x e y in coordinate relative... sommi a quelle di xr e
yr trovi coord assolute endpoint
  //  2) (scoretotale fuori inizializzata a 0)
  //       scoretotale+=ptOccProbability(xendpoint,yendpoint);
  //  3) fine ciclo ritorni valore

  float tempx, tempy;
  float totscore=0 ;
  for ( int i=0; (uint)i<r.size() ; i++) // there's a warning but i is always positive.
    {
      tempx=xr+r.at(i)*cos(theta+angle.at(i));
      tempy=yr+r.at(i)*sin(theta+angle.at(i)); // wrt absolute frame
       totscore+=ptOccProbability(tempx,tempy); // total score. Correlation function between
last scan and map.
    }

  return totscore;
}

// ---------------
void Node::getMaxresChildren(std::vector<int> &x,std::vector<int> &y)
{
  if (this->getIfLeaf()) // controlla se \'e nodo foglia
    {

      if (this->dim <= Node::MAXRES) // se \'e alla massima profondit\'a
        {

          // poi fai un check se \'e occprobability >0.50
          if (this->occProbability() > 0.50)
           {
             x.push_back(corner[0]);
             y.push_back(corner[1]);
             //std::cout << inserito << std::endl;
            }
          return;
```

```
      }
    else
      return;

  }
 else // \'e un ramo devo scendere
  {
    for (int n=0; n<4; n++) // controlla ogni sottoramo possibile
      {
        children[n]->getMaxresChildren(x,y);
      }
  }

}


void Node::getNearestCorner(const float &xpt, const float &ypt, int &cornerx,int &cornery)
{
 if (this->getIfLeaf()) // controlla se \'e nodo foglia
  {
   if (this->dim <= Node::MAXRES) // se \'e alla massima profondit\'a
     {
       //\'e dove andrebbe a cadere sicuramente il punto.
       cornerx=corner[0];
       cornery=corner[1];
       return;
     }
   else //sono in foglia non a risoluzione massima
     {
       // non esiste il sottoalbero, nell'insert viene creato.
       // scelgo di ritornare il floor del punto
       cornerx=(int)floor(xpt);
       cornery=(int)floor(ypt);
       return;
     }
  }
 else // \'e un ramo devo scendere in uno solo dei figli.
   {
     uint tempdim=0;
```

```
      int  tcornerx=0,tcornery=0; //temporanei servono per l'if
      for (int i=0; i<4; i++) // controlla ogni sottoramo possibile
        {
           // verifico sempre se nel quadrato cade il punto
           tcornerx=children[i]->getCornX();
           tcornery=children[i]->getCornY();
           tempdim=children[i]->getDim();
            if ( (xpt >= tcornerx) && (xpt < (tcornerx+tempdim)) && (ypt >= tcornery) &&
(ypt < (tcornery+tempdim)) )
              return children[i]->getNearestCorner(xpt,ypt,cornerx,cornery);
         }
    }
 // fine
}

// Metodo per trovare la distanza minima che sfrutta il quadtree
float Node::getNearestDistance(const float &xpt, const float &ypt)
{
  //fine ricorsione
  static const float BIAS=0.5;
  if (this->getIfLeaf() && this->dim <= Node::MAXRES )
    {
      if (this->occProbability() > BIAS)
       return sqrt( pow( (xpt - this->getCornX())  ,2) + pow( (ypt - this->getCornY()),2));
    }
  else if (this->getIfLeaf())
      return -1; // le distanze non sono mai negative. Lo uso per check
  else
    {
      //idea prendi il punto (xpt,ypt) confrontalo con i cornerx,y del nodo in cui sei.
      // fai un rank di questi nodi e visitali in sequenza.
      //int childrencorners[4][2];
      float childrendistance[4]; // 4 \'e il numero dei figli
      float tempdistance=10000;
      float tempmin=10000;
      int tempidx=0;
      bool taken[4]={false,false,false,false};
      int childrenvisitorder[4]={0,1,2,3};
      for (int i=0; i<4; i++)
       {
```

```
        childrendistance[i]=sqrt( pow( (xpt - children[i]->getCornX())  ,2) + pow( (ypt -
children[i]->getCornY()),2));
    }
    //trova il migliore
    //taken[4]={false,false,false,false};
    taken[0]=false,taken[1]=false,taken[2]=false;taken[3]=false;
    int indexvisit=0;
    // Forza il figlio in cui cade il punto a esser il migliore o allunghi l'algoritmo!
    int cornerx,cornery;
    uint tempdim;
    for ( int i=0; i<4; i++) // x ogni figlio
      {
       // verifico sempre se nel quadrato cade il punto
       cornerx=children[i]->getCornX();
       cornery=children[i]->getCornY();
       tempdim=children[i]->getDim();
        if ( (xpt >= cornerx) && (xpt < (cornerx+tempdim)) && (ypt >= cornery) && (ypt
< (cornery+tempdim)) )
            {
             //indexvisit \'e =0
             childrenvisitorder[indexvisit]=i; // forzi a visitare prima questo
             indexvisit++;
             taken[i]=true;
             break; // e non controllare +. Ce ne pu\'o esser solo 1.
            }
      }

    // Fine verifica caduta figlio in un nodo

    for (int j=0; j<4; j++)
     {
      //riporto il minimo in alto
      tempmin=100000;
      for (int k=0; k<4; k++)
        {
         if (  taken[k]==false  && childrendistance[k] < tempmin)
           {
            tempmin=childrendistance[k];
            tempidx=k;
           }
```

```
            } // end for k
          taken[tempidx]=true; // lo prendo e non lo considero piu nei minimi.
          childrenvisitorder[indexvisit]=tempidx;
          indexvisit++; //  non \'e detto sia=j vedi se il figlio cade in un nodo

        }
    //visita in base al rank
    for (int i=0; i<4; i++)
    {
      tempdistance=children[childrenvisitorder[i]]->getNearestDistance(xpt,ypt);
      if ( tempdistance != -1)
            return tempdistance;
    }
    // se non trovi nulla tra questi figli.
    return -1;
  }
  return -1; // non arriva mai qui. solo con break
}

// Score che sfrutta la distanza con getNearestDistance
float Node::score_nearestdist(const float &xr,const float &yr, const float &theta, const std::vector<float>
&r, const std::vector<float> &angle)
{
  float totscore=0;
  float tempx,tempy;
  for ( int i=0; (uint)i< r.size(); i++) // per ogni beam dello scan
   {
     tempx=(xr+ r.at(i)*cos(theta+angle.at(i)) );
     tempy=(yr+ r.at(i)*sin(theta+angle.at(i)) ); // wrt absolute frame
     totscore=totscore - this->getNearestDistance(tempx,tempy);
   }
   return totscore;
}



// ---------------
// Score che sfrutta la distanza
float Node::score_distance(const float &xr,const float &yr, const float &theta, const std::vector<float>
&r, const std::vector<float> &angle)
{
```

```
float totscore=0;
std::vector<int> x,y;

// cerca la lista di figli a massima risoluzione e ritorna cornerx e cornery
this->getMaxresChildren(x,y);

// ---- DEBUG ----
// std:: cout << Finito di inserire valori nel vettore << x.size() << std::endl;
// ----------

// METODO GREZZO LE CONFRONTO CON TUTTE (COSTOSO)
int distance=0, tempdistance=1000;
int tempx,tempy; //        ERRORE!!
//int sigma;

for ( int i=0; (uint)i<r.size(); i++ ) // per ogni beam dello scan
 {
   // trasformo in interi i punti terminali

   //tempx=(int)floor(xr+ r.at(i)*cos(theta+angle.at(i)) );
   //tempy=(int)floor(yr+ r.at(i)*sin(theta+angle.at(i)) ); // wrt absolute frame
   tempx=(xr+ r.at(i)*cos(theta+angle.at(i)) );
   tempy=(yr+ r.at(i)*sin(theta+angle.at(i)) ); // wrt absolute frame



   // cerco il cornerX,Y piu vicino.
   int cornerx=0,cornery=0;

   this->getNearestCorner(tempx,tempy,cornerx,cornery);
   // trovo la cella nella mappa di dimensioni MAXRES pi\'u vicina a lui
   // cio\'e quella con la distanza minore!!!
   distance=(pow((cornerx-x.at(0) ),2)+pow((cornery-y.at(0)),2));

   for(int h=1; (uint)h<x.size(); h++)
    {
      tempdistance=(pow((tempx-x.at(h) ),2)+pow((tempy-y.at(h)),2));
      if (tempdistance < distance) distance=tempdistance;
    }
   // ho trovato la distanza minore!
   // ora la sommo opportunamente nella totscore
```

```
      totscore=totscore - sqrt(distance);
   }

  return totscore;

}
```

```
// INSERISCI SOLO IL PUNTO (SENZA VISITATI)

bool Node::insertFinalPt(const float &xpt, const float &ypt)
{
//std::cout <<  entrato << std::endl;
//per ricorsione
if (this->leaf==true && this->dim <=MAXRES)
   {
    // not vis++ just to avoud double increment with brasenham
    if (occ==0 ) //succede perch\'e il nodo \'e di default a occ=0,vis=1
     {
      // anche se non si fa l'if in due volte diventa nero.
      occ++;
      updated=true; // vis \'e gi\'a a uno
     }
    else
     {
      occ++;
      updated=false; // per sicurezza
     }

    return true;
   }
else if ( leaf==true && dim > MAXRES )
   {
    //devo spezzare perch\'e non sono a MAXRES
    leaf=false; // diventa un ramo
    //creo 4 nodi
    int newcorn[4][2]; // coordinate dei nuovi angoli
```

```
    newCorner(newcorn); // this->
    int newdim=dim/2; // nuove lunghezze lati
    for (int j=0; j<4; j++)
        children[j]=new Node(true,1,0,newdim,newcorn[j][0],newcorn[j][1]);

    for ( int i=0; i<4; i++) // x ogni figlio
    {
    // verifico sempre se nel quadrato cade il punto
    int cornerx=children[i]->getCornX();
    int cornery=children[i]->getCornY();
    uint tempdim=children[i]->getDim();

     if ( (xpt >= cornerx) && (xpt < (cornerx+tempdim)) && (ypt >= cornery) && (ypt <
(cornery+tempdim)) )
            return children[i]->insertFinalPt(xpt,ypt);
    }
   }
else //this->leaf == false Stai visit il ramo almeno per la 2a volta (gi\'a creato)
   {
    for ( int i=0; i<4; i++) // x ogni figlio
    {
    // verifico sempre se nel quadrato cade il punto
    int cornerx=children[i]->getCornX();
    int cornery=children[i]->getCornY();
    uint tempdim=children[i]->getDim();
     if ( (xpt >= cornerx) && (xpt < (cornerx+tempdim)) && (ypt >= cornery) && (ypt <
(cornery+tempdim)) )
            return children[i]->insertFinalPt(xpt,ypt);
    }
   }
   return true;
}

bool Node::visit(const float &xpt, const float &ypt)
{
//std::cout <<  entrato << std::endl;
//per ricorsione
if (this->leaf==true && !updated)
{
   vis++;
```

```
  updated = true;
  return true;
}
else if (this->leaf==true && updated)
  return true;  // non aggiorno i visitati
else
{
   for ( int i=0; i<4; i++) // x ogni figlio
   {
   // verifico sempre se nel quadrato cadeil punto
   int cornerx=children[i]->getCornX();
   int cornery=children[i]->getCornY();
   uint tempdim=children[i]->getDim();
    if ( (xpt >= cornerx) && (xpt < (cornerx+tempdim)) && (ypt >= cornery) && (ypt <
(cornery+tempdim)) )
          return children[i]->visit(xpt,ypt);
   }
  }
return true;
}


bool Node::clean(const float &xpt, const float &ypt)
{
//std::cout <<  entrato << std::endl;
//per ricorsione
if (this->leaf==true)
{
      updated = false;
      return true;
}
else
{
   for ( int i=0; i<4; i++) // x ogni figlio
   {
   // verifico sempre se nel quadrato cade il punto
   int cornerx=children[i]->getCornX();
   int cornery=children[i]->getCornY();
   uint tempdim=children[i]->getDim();
    if ( (xpt >= cornerx) && (xpt < (cornerx+tempdim)) && (ypt >= cornery) && (ypt <
```

```
(cornery+tempdim)) )
            return children[i]->clean(xpt,ypt);
    }
   }
return true;
}


bool Node::insertPt(const float &xr,const float &yr, const float &xpt, const float &ypt)
{
//std::cout <<  entrato << std::endl;
//per ricorsione
if (this->leaf==true && this->dim <= MAXRES)
   {
    // punto trovato
    if ( occ == 0 )
       {
        vis=1; //causa creazione nodi
        occ++; // * (cambiare i valori...)
        return true;
       }
    else
       {
        vis++; //*
        occ++; //*
        return true;
       }
   }
else if ( leaf==true && dim > MAXRES)
   {
    //devo spezzare perch\'e non sono a MAXRES
    leaf=false; // diventa un ramo
    //creo 4 nodi
    int newcorn[4][2]; // coordinate dei nuovi angoli
    newCorner(newcorn); // this->
    int newdim=dim/2; // nuove lunghezze lati
    for (int j=0; j<4; j++)
       {
        children[j]=new Node(true,1,0,newdim,newcorn[j][0],newcorn[j][1]);
```

```
      }
    //aggiornointersezioni e scegli dove andare
    Node *prox=intUpdate(xr,yr,xpt,ypt);
    // andiamo avanti con la ricorsione sul nodo prescelto
    return prox->insertPt(xr,yr,xpt,ypt);
   }
else //this->leaf == false Stai visit il ramo almeno per la 2a volta (gi\'a creato)
   {
    //aggiornointersezioni e scegli dove andare
    Node *prox=intUpdate(xr,yr,xpt,ypt);
    //std::cout << *prox << std::endl;
    return prox->insertPt(xr,yr,xpt,ypt);
   }

}




// ---------------
// Aggiorna le intersezioni e ritorna il punto per il prox inserimento in profondit\'a
Node* Node:: intUpdate (const float &xr, const float &yr, const float &xpt, const float &ypt)
{
//aggiorna il vis solo delle foglie
assert (children[0]!=0 && children[1]!=0 && children[2]!=0 && children[3]!=0) ; // leaf=0

Node *next; // prossimo nodo nella ricorsione dell'insert
int cornerx, cornery,tempdim; // var utili x brevita
float temp;
bool ptfall=false, robotfall=false, touch=false; // indica se cade il punto o il robot nel quadrato
float intersection[2]; //punto di intersezione quadrato retta R-PT
bool mzero=false,minf=false; // pendenza retta R-PT se m=0 o m=inf
// CREO IL RETTANGOLO 'AREA DI MINACCIA DEL SEGMENTO R-PT
/*
Vertici:
C D
A B
*/
float menace[4][2];
```

```
if (xr <= xpt ) //xr \'e a sinistra o uguale
  {
   if (yr <= ypt) // yr \'e in basso o uguale
     {
      menace[0][0]=xr; menace[0][1]=yr;//(x,y) di A
      menace[1][0]=xpt; menace[1][1]=yr; //(x,y) di B
      menace[2][0]=xpt; menace[2][1]=ypt; //(x,y) di C
      menace[3][0]=xr; menace[3][1]=ypt; //(x,y) di D
     }
   else // xr < xpt ed retta m<0
     {
      menace[0][0]=xr; menace[0][1]=ypt;//(x,y) di A
      menace[1][0]=xpt; menace[1][1]=ypt; //(x,y) di B
      menace[2][0]=xpt; menace[2][1]=yr; //(x,y) di C
      menace[3][0]=xr; menace[3][1]=yr; //(x,y) di D
     }
  }
else // xr \'e a destra di xpt
  {
   if (yr <= ypt) // xr \'e a dx e m retta < 0
     {
      menace[0][0]=xpt; menace[0][1]=yr;//(x,y) di A
      menace[1][0]=xr; menace[1][1]=yr; //(x,y) di B
      menace[2][0]=xr; menace[2][1]=ypt; //(x,y) di C
      menace[3][0]=xpt; menace[3][1]=ypt; //(x,y) di D
     }
   else  // xr \'e a dx ed m >0
    {
      menace[0][0]=xpt; menace[0][1]=ypt;//(x,y) di A
      menace[1][0]=xr; menace[1][1]=ypt; //(x,y) di B
      menace[2][0]=xr; menace[2][1]=yr; //(x,y) di C
      menace[3][0]=xpt; menace[3][1]=yr; //(x,y) di D
    }
  }

//FINE CREAZIONE
//sono inizializzate a false
if (yr == ypt) mzero= true; // posso dividere per (ypt-yr)
if (xr == xpt) minf= true; // posso dividere per (xpt -xr)
// CONTROLLO SU PENDENZA RETTA
```

```
// FINE CONTROLLO

for ( int i=0; i<4; i++) // x ogni figlio
  {
   // verifico sempre se nel quadrato cade il robot e/o il punto
   // qualche var che aiuta
   ptfall=false; robotfall=false;
   cornerx=children[i]->getCornX();
   cornery=children[i]->getCornY();
   tempdim=children[i]->getDim();
    //std::cout << figlio considerato: << cornerx << , <<cornery << ,  << tempdim << std::endl;

   // CASO CADUTA PUNTO
    if ( (xpt >= cornerx) && (xpt < (cornerx+tempdim)) && (ypt >= cornery) && (ypt < (cornery+tempdim)) )
     {
      // per forza se ci cade questo \'e il next da valutare
      ptfall=true;
      next=children[i];
     }

    // CASO CADUTA ROBOT NEL QUADRATO
   if ( (xr >= cornerx) && (xr < (cornerx+tempdim)) && (yr >= cornery) && (yr < (cornery+tempdim)) )
     {
      // se cade robot e nn punto questo ramo \'e sicuramente attraversato, aggiornare vis
      robotfall=true; // prima di aggiornare serve 1 controllo (caso entrambi bool=true)

      }

   //Applico decisione del caso caduta robot
   if (robotfall==true && ptfall==false)
    {
     children[i]->incVis(1); // incremento di 1 il vis del nodo
     if ((children[i]->getIfLeaf()==false))
        {
          Node *tempzero; //non serve neppure guardarlo
              tempzero=children[i]->intUpdate(xr,yr,xpt,ypt); // attenzione qui next=0 xk non
```

```
trova il punto!!
        }
    }
  else if ( robotfall==false && ptfall==false) // pu\'o essere attraversato dalla retta o meno
    {
      //std::cout << sto considerando   << *(children[i]) << std::endl; // cancellare
      //Guardo se la retta interseca il quadrato in 0,1 punti.
      // in 0 punti. Non lo tocca -> nessun aggiornamento
      // in 1 punto. Potrebbe toccarlo
      // valuto g(x)=retta passante per (xr,yr) (xpt,ypt)
      //Voglio fare se g(x)=corner[1] (la y dell'angolo)
      // uso : ((y-yr)/(ypt-yr) = (x-xr)/(xpt-xr))
      temp=0; // il risultato potrebbe esser float..lo tronco

      // avendo scartato gli altri2casi basta che si verifichi una ipotesi. (basta che tocca1lato)

      //verifica che il quadrato sia all'interno del segmento robot-punto e tocchi il segmento
      // e non la retta generata.
      touch=false; // VEDI SE RETTA INTERSECA UN LATO DEL QUADRATO

      //se interno al segmento verifico se lo tocca o meno
      if (mzero == false ) temp=( ((((double)(cornery-yr))/(double)(ypt-yr))*(xpt-xr))+xr ) ;
// rappr una x
      //std:: cout << temp << std::endl;
    if ( ((mzero ==false) && (temp < ((float)(cornerx+tempdim))) && (temp > ((float)cornerx)))
||  ((minf==true) && (temp < ((float)(cornerx+tempdim)))  && (temp >= ((float)cornerx)) )
)
        { // tocchi la retta y=cornery
         touch=true;
         intersection[0]=temp; intersection[1]=(float)cornery;  // interseca in (x,y)
        }
      else
        {
         if (mzero==false) temp= ( ((double)((cornery+tempdim)-yr)/(double)(ypt-yr))*(xpt-
xr)+xr ) ; // rappr una x
          //std:: cout << temp << std::endl;
              if ( ((mzero==false) && (temp < ((float)(cornerx+tempdim)))  && (temp >
((float)cornerx)))  ||  ((minf==true) && (temp < ((float)(cornerx+tempdim)))  && (temp >=
((float)cornerx)) ) )
          { // tocchi la retta y=corner[1]+tempdim all'interno del quadrato
```

```
            touch=true;
           intersection[0]=temp; intersection[1]=(float)(cornery+tempdim);  // interseca in (x,y)
           }
          else
           {
           // Voglio fare se g(corner[0])
           if (minf == false) temp = ( (((((double)(cornerx-xr))/(double)(xpt-xr))*(ypt-yr))+yr
); //rappr una y
           //std:: cout << temp << std::endl;
                if ( ((minf == false) && (temp < ((float)(cornery+tempdim))) && (temp >
((float)cornery))) || ((mzero==true) && (temp < ((float)(cornery+tempdim))) && (temp >=
((float)cornery)) ))
              {  // tocchi la retta x= corner[0] all'interno del quadrato considerato
               touch=true;
               intersection[0]=(float)cornerx; intersection[1]=temp;  // interseca in (x,y)
              }
             else
              {
                if (minf ==false ) temp = ( (((((double)((cornerx+tempdim)-xr))/(double)(xpt-
xr))*(ypt-yr))+yr ); //rappr una y
               //std:: cout << temp << std::endl;
                   if ( ((minf== false) && (temp < ((float)(cornery+tempdim))) && (temp >
((float)cornery))) || ((mzero==true) && (temp < ((float)(cornery+tempdim))) && (temp >=
((float)cornery)) ) )
                 {  // tocchi la retta x= corner[0]+tempdim all'interno del quadrato considerato
                  touch=true;
                   intersection[0]=(float)(cornerx+tempdim); intersection[1]=temp;// interseca in
(x,y)
                 }
              }
           }
         }
        // FINE CHECK TOUCH

                  // problema nel caso non sia foglia.... facciamo un check...bisogna chiamare
ricorsivamente l'update
        if ((children[i]->getIfLeaf()==false) && (touch==true))
         {
           Node *tempzero; // in teoria forse non serve neppure guardarlo
              tempzero=children[i]->intUpdate(xr,yr,xpt,ypt); // attenzione qui next=0 xk non
```

```
trova il punto!!
        }

            //Gestione casi minf o mzero == true
            if ( (mzero==true) || (minf==true) )
             {

                //un caso alla volta
                // se true entrambi sono lo stesso punto...non faccio nulla
              if ((mzero == true) && (minf == false))
                {

                 //assert(touch==false); // in teoria
                 touch=false; //lo gestisco qui
               // devi vedere se xr<xpt o viceversa...
                 float xtemp[2]; //uso una var temporanea x togliere casi
               if (xr<xpt) {xtemp[0]=xr; xtemp[1]=xpt;} else {xtemp[0]=xpt; xtemp[1]=xr;}
                 // verifico la coordinata x di corner se sta tra i due (c' 1dim di mezzo)
                 if ((cornerx > (xtemp[0]-tempdim) )&& ( cornerx < xtemp[1]) )
                  {

                        //check sulla y...se c' intersezione (yr=ypt) Se  compreso:
                   if ( (cornery <= yr) && ((cornery+tempdim) > yr)) children[i]->incVis(1);
                  }
                }
                else if ( (minf==true) && (mzero== false))
                 {
                  // Caso simmetrico
                  //assert (touch==false);// in teoria (assert?)
                  touch=false; //lo gestisco qui
                  float ytemp[2]; // var temp come sopra (ytemp[0]=basso)
                if (yr<ypt) {ytemp[0]=yr; ytemp[1]=ypt;} else {ytemp[0]=ypt; ytemp[1]=yr;}
                  // Controllo se la y  nel range
                     if (((cornery +tempdim )> (ytemp[0]) )&& ( cornery  < ytemp[1]) ) //
dovrebbe prendere anche i casi strani.
                   {
                        //check sulla x...se c' intersezione (xr=xpt) Se  compreso:
                   if ( (cornerx <= xr) && ((cornerx+tempdim) > xr)) children[i]->incVis(1);
                   }
```

```
                    }
                }

                // c' touch che toglie disguinge da sottoinsieme caso minf e mzero
        // se toccato verifico che il punto intersezione sta nell'area minacciata
    if ( (children[i]->getIfLeaf()==true) && (touch==true) && (intersection[0] > ((float)menace[0][0]))
&& (intersection[0] < ((float)menace[1][0])))
          { // tra xA e xB
           if ( (intersection[1] > ((float)menace[1][1])) && (intersection[1] < ((float)menace[2][1]))
) // tra yB e yC
             {
              // effetivamente allora soddisfa entrambe le ipotesi quindi lo aggiorno a bianco
              children[i]->incVis(1);
             }
          }
      }

  }

return next; // ritorno il next

}



// ---------------
// Da this->dim e x,y (corner) restituisce i nuovi corner dei 4 lati dei 4 subsquare

void Node:: newCorner(int array[4][2])
{

// ricordare il -1 (da 0 a 64 -> dim32 -> coordinata a 31)

/*          DISPOSIZIONE
         ---------------
        |          |             |
        |   1          |   2      |
        |          |             |
        |------ |--  ----|
        |          |             |
```

```
        |   3     |  4        |
        |           |                |
        |-------|-------|
*/

// array[4][2] contiene coppie di corner[0] e [1]

int newdim=dim/2;                //la nuova dimensione sar\'a la met\'a della precedente

array[0][0]=corner[0];            //corner[0] (x) di 1
array[0][1]=corner[1]+newdim;     //corner[1] (y) di 1

array[1][0]=corner[0]+newdim;     //corner[0] (x) di 2
array[1][1]=corner[1]+newdim;     //corner[1] (y) di 2

array[2][0]=corner[0];            //corner[0] (x) di 3
array[2][1]=corner[1];            //corner[1] (y) di 3

array[3][0]=corner[0]+newdim;     //corner[0] (x) di 4
array[3][1]=corner[1];            //corner[1] (y) di 4
}
```

quadtree.h

```
/*****************************************************************************
 *   Copyright (C) 2007 by Mauro Brenna, Ivan Reguzzoni                      *
 *   malloblenne@gmail.com                                                   *
 *                                                                           *
 *   This program is free software; you can redistribute it and/or modify    *
 *   it under the terms of the GNU General Public License as published by    *
 *   the Free Software Foundation; either version 3 of the License, or       *
 *   (at your option) any later version.                                     *
 *                                                                           *
 *   This program is distributed in the hope that it will be useful,         *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of          *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the           *
 *   GNU General Public License for more details.                            *
 *                                                                           *
 *   You should have received a copy of the GNU General Public License       *
 *   along with this program;If not, see <http://www.gnu.org/licenses/>.     *
 *                                                                           *
 *****************************************************************************/
#ifndef _QUADTREE_H
#define _QUADTREE_H

#include <cmath>
#include <vector>
#include node.h
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
//Per le funzioni random si usa la GNU Scientific Library
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h> // per le gaussiane
//



using namespace std;

// Struct utilizzata nel metodo generatePoseandRanK
struct posewithscore
{
```

```
  float x;
  float y;
  float theta;
  float score;
};

class Quadtree
{

friend std::ostream &operator<<(std::ostream &, Quadtree &);

public:
Quadtree(uint maxdim, uint initdim);
~Quadtree();
//inserimento un punto alla volta (controlla anche se sfora la dim...e provvede)
bool insertPt (const float &xr,const float &yr, const float &xpt, const float &ypt);
bool Scanmatcher1(float &xr, float &yr, float &theta, uint k, std::vector<float> &r, std::vector<float>
&angle );
bool Scanmatcher2(float &xr, float &yr, float &theta, uint k, std::vector<float> &r, std::vector<float>
&angle );
bool getScan(uint k, std::vector<float> &r, std::vector<float> &angle);

private:
Node* getRoot() {return root;}
void createNodesandPutN(Node *n,int childpos);
void fatherCorner(Node *n,float x, float y);
bool outOfBorder(const float &x,const float &y, Node *n, bool &limitReached); // check se
il punto o il robot sono fuori e crea quadrato + grosso
 void generatePoseandRank(posewithscore*  array,const int ARRAYDIM, const float star-
tingpose,const float generatefromone,const float sigmacart,const float sigmarot, const std::vector<float>
&r, const std::vector<float> &angle );
Node *root; // radice dell'albero
const uint MAXDIM; // max dim del quadtree. Serve quando lo ingrandisce
const uint INITDIM; // dimensione quadtree iniziale..Poi si pu\'o espandere

};

#endif /* _QUADTREE_H */
```

quadtree.cpp

```
/*************************************************************************
 *   Copyright (C) 2007 by Mauro Brenna, Ivan Reguzzoni                  *
 *   malloblenne@gmail.com                                               *
 *                                                                       *
 *   This program is free software; you can redistribute it and/or modify *
 *   it under the terms of the GNU General Public License as published by *
 *   the Free Software Foundation; either version 3 of the License, or   *
 *   (at your option) any later version.                                 *
 *                                                                       *
 *   This program is distributed in the hope that it will be useful,     *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of      *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the       *
 *   GNU General Public License for more details.                        *
 *                                                                       *
 *   You should have received a copy of the GNU General Public License   *
 *   along with this program;If not, see <http://www.gnu.org/licenses/>.  *
 *                                                                       *
 *************************************************************************/
#include quadtree.h

// ----------
// Costruttore

Quadtree::Quadtree(uint maxdim, uint initdim)
: MAXDIM(maxdim),INITDIM(initdim)              // inizializzazione delle const
{
  root = new Node(false,0,0,initdim,0,0);        // coordinate assolute (0,0)
  int array[4][2];
  root->newCorner(array);
  uint newdim= initdim/2;

  for (int j=0; j<4; j++)
    {
      root->setChild(j,new Node(true,0,0,newdim,array[j][0],array[j][1]));
//    std::cout << Nodo : << j <<  cornerx:  << array[j][0] << cornery:  << array[j][1] <<
std::endl;
    }

//fine ho inizializzato 1 nodo radice + 4 nodi foglia
```

```
}


// ----------
//Distruttore

Quadtree:: ~Quadtree()
{
delete root;
}

// Visualizza Node
std::ostream &operator<< (std::ostream &output, Quadtree &q)
{

output << Image image( Geometry(<< q.getRoot()->getDim()<<*mul ,<< q.getRoot()->getDim()
     << *mul), Color(\white\) );<< std::endl
     << image.strokeColor(\blue\); << std::endl
     << image.fillColor(\green\); << std::endl
     << image.strokeWidth(1); << std::endl << std::endl
     << std::endl << const int x_o= << q.getRoot()->getCornX() << ; << std::endl
     << std::endl << const int y_o= << q.getRoot()->getCornY() << ; << std::endl
     << *(q.getRoot());

return output;
}

// Retrieve a scan
/*
INPUT
t=k*Ts  where Ts=1/fs; sampling; k integer (the filename of log files)
OUTPUT
r= raggio ; angle= angolo ottenuto dal sensore x la misura
*/
bool Quadtree::getScan(uint k, std::vector<float> &r, std::vector<float> &angle)
{
/*
1) apri il file di nome k o k.log
Del tipo:
```

```
r angle
r angle
.....
2) parser salvi r e angle di ogni riga fino alla fine file pushando nel vector
3) Ottieni OUTPUT
*/
 char buffer[30];
 sprintf(buffer,%d,k);
 string sbuff(buffer);
 string file = logfiles/ + sbuff + .log;
 cerr << file << endl;

 ifstream* in = new ifstream(file.c_str());// allocazione dinamica, dato che il nome del file si
sa a run-time
 if(!in->is_open())
   {
     // se il file non \'e stato trovato
     // avverti l'utente
     cerr << Errore: file  + file +  non trovato << endl;
     return false;
   }
string f; // float temp per salvare numero corrente

float temp;
 while(*in >> f)
   {

      temp=atof(f.c_str());
      r.push_back(temp/10); // aggiungila al vettore di float (raggio)
      *in >> f;
      temp=atof(f.c_str());
      angle.push_back(temp); // angles
      // formato file: raggio angolo /n ...
    }

  return true; // file founded and processed
}
```

```
//Scanmatching greedy tutorial Grisetti
/*
  Ingresso: pose tempo t-1 x(t-1) [where t=k*Ts      Ts=1/fs; frequency sampling]
  Output: pose tempo t   x(t)
*/
bool Quadtree::Scanmatcher1(float &xr, float &yr, float &theta,uint k, std::vector<float> &r,
std::vector<float> &angle)
{
  // chiedo al laser o un logfile o altro di darmi i punti di questa scansione  z(t)

  //std::vector<float> r,                // raggi endpoint trovati wrt robot
  //                   angle;          // angoli delle miser wrt robot [coordinate relative robot]


  // ---- Debug ----
  // cout << endl << Sono in ScanMatcher <Metodo_1> << endl << endl;
  // ----------

  if (!(getScan(k,r,angle)))          // where t = k*Ts      Ts=1/fs; frequency sampling
    return false;

  float bestxr = xr,                        // the best value for pose at time t
        bestyr = yr,
        besttheta = theta,
        bestmovexr = 0,
        bestmoveyr = 0,
        bestmovetheta = 0;

  // score e' una semplice funzione da max/min-imizzare
  float bestscore = root->score_distance(bestxr,bestyr,besttheta,r,angle);

  float testxr,testyr,testtheta;          //test pose
  float testscore;


  // ---- PARTE RANDOM ----
//
//  const float MOV=2, ROT=10*3.14/180; // 2 cm 2gradi
//  const int ITER=100;
```

```
//  float testrot,testmove;
//  for (int i=0; i<ITER; i++)
//    {
//     switch (rand()%2)
//       {
//         case 0: testrot= theta + ((float)rand() / (float)(RAND_MAX))*ROT; break;
//         case 1: testrot= theta - ((float)rand() / (float)(RAND_MAX))*ROT; break;
//         default: break;
//       }
//     testmove=((float)rand() / (float)(RAND_MAX))*MOV;
//     testxr= xr + testmove*cos(testrot);
//     testyr= yr + testmove*sin(testrot);
//     testscore=root->score_distance(testxr,testyr,testrot,r,angle);
//
//     if (bestscore < testscore)
//       {
//         bestxr=testxr;
//         bestyr=testyr;
//         besttheta=testrot;
//         bestscore=testscore;
//       }
//    }
//
  // ---- FINE PARTE RANDOM ----

  // Ho i dati dello scan e la posizione precedente...procedo con un algoritmo di discesa del gradiente



  const int MAXITERATIONS=60;

  //float searchstep=INITIALSEARCHSTEP;
  int iterations = 0;
  int cntswitch  = 0;                    // counter per switch (0-5)
  const float MOVE   = 0.1;              // minor spostamento possibile in avanti,dietro,sx,dx [DEFAULT = 0.1]
  const float ROTATE = 1*(3.14/180);     // minor spost possibile in gradi                     [DEFAULT = 1]
  testscore = 0;
```

```
float maxmovescore;
bool end = false;


while ( (iterations < MAXITERATIONS) && (end==false) )
 {

   maxmovescore  = bestscore;
   bestmovexr    = bestxr;
   bestmoveyr    = bestyr;
   bestmovetheta = besttheta;

   cntswitch  = 0;

   for(cntswitch=0;cntswitch<6;cntswitch++)
    {
     // 0 = backward
     // 1 = forward
     // 2 = left
     // 3 = right
     // 4 = rotate_left
     // 5 = rotate_right

     switch(cntswitch)
      {
        case 0: { testxr = bestxr-MOVE;       testyr = bestyr;       testtheta = besttheta;};
break;
        case 1: { testxr = bestxr+MOVE;       testyr = bestyr;       testtheta = besttheta;};
break;
        case 2: { testxr = bestxr;       testyr = bestyr-MOVE;       testtheta = besttheta;};
break;
        case 3: { testxr = bestxr;       testyr = bestyr+MOVE;       testtheta = besttheta;};
break;
       case 4: { testxr = bestxr;       testyr = bestyr;       testtheta = besttheta-ROTATE;};
break;
      case 5: { testxr = bestxr;       testyr = bestyr;       testtheta = besttheta+ROTATE;};
break;
        default: break;
      }
```

```
        // Calcolo la score dopo avere effettuato lo spostamento
        testscore = root->score_distance(testxr,testyr,testttheta,r,angle);

        // --- DEBUG ---
         //cout<< maxmovescore -->  << maxmovescore <<  testscore -->  << testscore <<
endl;
        // ----------

        // Trovata score migliore tra le 6 proposte
        if ( maxmovescore < testscore )
          {
            maxmovescore  = testscore;
            bestmovexr    = testxr;
            bestmoveyr    = testyr;
            bestmovetheta = testttheta;
          }

      }

    // --- DEBUG ---
    //cout << endl << bestscore --> << bestscore <<  maxmovescore -->  << maxmovescore
<< endl << ----- << endl;
    // ----------

    // Aggiorno la score nel caso ne trovi una migliore della precedende
    if ( bestscore < maxmovescore )
      {
        bestscore = maxmovescore;
        bestxr    = bestmovexr;
        bestyr    = bestmoveyr;
        besttheta = bestmovetheta;

        end = false;
        iterations++;
      }
    else
      {
        end=true;       // se alla fine si aggiorna qualcosa riporto a false
      }
```

```
    }

  //return updated pose
  xr    = bestxr;
  yr    = bestyr;
  theta = besttheta;

  return true;
}


// Metodo utilizzato da Scanmatching che sfrutta distrib normali. Genera le pose e valuta la score
// restituendo un array ordinato per score
/*
 Ingresso: array[ARRAYDIM] di pose e score, da quante pose generarne altre, quante generarne
da ogni pose (lei stessa non e' compresa), deviazione standard delle gaussiane, l'ultimo scan ri-
cevuto.
 Output: array ordinato in base allo score piu' alto.
 */
void Quadtree::generatePoseandRank(posewithscore*  array, const int ARRAYDIM, const
float startingpose, const float generatefromone, const float sigmacart, const float sigmarot, con-
st std::vector<float> &r, const std::vector<float> &angle )
{
  //Ipotesi: array gi\'a ordinato in base alla score
   for(int i=0;i<(ARRAYDIM -1); i++ ) assert( array[i].score>=array[i+1].score);

   //1) Genero le pose e valuto le score
   posewithscore temp;
   int index=startingpose;
   temp.x=0;temp.y=0;temp.score=-20000;
   //inizializzo random per gaussiane
   gsl_rng * ra = gsl_rng_alloc (gsl_rng_taus);

   for (int i=0; i<startingpose;i++)
     {
       for(int j=0; j<generatefromone; j++)
         {
           //Per ogni pose in i genero generatefromone pose in piu'
           temp.x=array[i].x + gsl_ran_gaussian( ra, sigmacart);
```

```
        temp.y=array[i].y + gsl_ran_gaussian( ra, sigmacart);
        temp.theta=array[i].theta + gsl_ran_gaussian( ra, sigmarot);
        // VALUTO LA SCORE
        temp.score=root->score_distance(temp.x,temp.y,temp.theta,r,angle);
        //Inserisco nel posto giusto
        array[index].x=temp.x;
        array[index].y=temp.y;
        array[index].theta=temp.theta;
        array[index].score=temp.score;
        index++; // aggiorno la posizione dove inserire nuovi dati
      }
  }
// Finita la generazione si ordina in base alla score

//utilizzo shellsort, alla peggio O(n^2) http://en.wikipedia.org/wiki/Shell_sort
int i, j, increment;


increment =ARRAYDIM / 2;

while (increment > 0)
{
  for (i=increment; i < ARRAYDIM; i++)
  {

    j = i;
    temp = array[i];
    while ((j >= increment) && (array[j-increment].score < temp.score)) // <  e non >
    {
      array[j] = array[j - increment];
      j = j - increment;
    }
    array[j] = temp;
  }


  if (increment == 2)
     increment = 1;
  else
     increment = (int) (increment / 2.2);
```

```
  }
  //Fine ordine, algoritmo terminato
  //postcondizione deve essere ordinato
}


//Scanmatching che sfrutta distribuzioni normali
/*
  Ingresso: pose tempo t-1 x(t-1) [where t=k*Ts      Ts=1/fs; frequency sampling]
  Output: pose tempo t   x(t)
*/
bool Quadtree::Scanmatcher2(float &xr, float &yr, float &theta,uint k, std::vector<float> &r,
std::vector<float> &angle)
{
 if (!(getScan(k,r,angle))) // where t=k*Ts      Ts=1/fs; frequency sampling
    return false;
 // --- DEBUG ---
 //cout <<  SEI ENTRATO NELL FUNZIONE SCANMATCHER PER TEST << endl;
 // ----------


 //pose iniziale
 float testxr=xr,
       testyr=yr,
       testtheta=theta;

 //calcolo la score
 float testscore=root->score_distance(testxr,testyr,testtheta,r,angle);

 const int ARRAYDIM=100;
 posewithscore array[ARRAYDIM];
 for (int i=0; i< ARRAYDIM; i++)
  {
    //inizializzo l'array
    array[i].x=0;
    array[i].y=0;
    array[i].theta=0;
    array[i].score=-20000; // in score_distance il massimo \'e 0.
  }
```

```
  //pongo la posizione test nel primo spazio dell'array
  array[0].x=testxr;
  array[0].y=testyr;
  array[0].theta=testtheta;
  array[0].score=testscore;
  //faccio i calcoli
  const int MAXITERATIONS=3;
  const int STARTING=20;
  const int GENERATE=ARRAYDIM/STARTING -1; //es 50/10 -1 = 4 x ogni pose
  int sigmacart=2; // per movimenti ortogonali agli assi cartesiani
  int sigmarot=2;  // per movimenti di rotazione.
  //inizio con generarne ARRAYDIM-1 dalla iniziale
 this->generatePoseandRank(array,ARRAYDIM, 1, ARRAYDIM-1, sigmacart*2,sigmarot*2,r,
angle );

  for (int divisor=1; divisor <= MAXITERATIONS; divisor++ )
    {
    this->generatePoseandRank(array,ARRAYDIM, STARTING, GENERATE,sigmacart/divisor
,sigmarot/divisor,r, angle );
    }
  // uso la migliore tra tutte
  xr=array[0].x;
  yr=array[0].y;
  theta=array[0].theta;
  // alla fine ritorno la migliore

  return true;
}

// Inserisce punto usando Bresenham (in ingresso i punti a coord cart assolute)
bool Quadtree::insertPt (const float &xr,const float &yr, const float &xpt, const float &ypt)
{
 // verifico che il punto non sia fuori dal quadrato...
 // se \'e all'interno chiamo l'insert del nodo, altrimenti devo creare nuovi nodi...e richiamare
questa!! funzione (potrebbe servire ancora di incrementare la dim)
bool limit=false;
if (outOfBorder(xr,yr,root,limit) || outOfBorder(xpt,ypt,root,limit))
{
 //sono fuori dal quadrato
```

```
 if (limit==true) // ho sforato la dim max...nn posso fare nulla
   {

   // --- DEBUG ---
   // std::cout << sono qui!!!!!!!!!! << std::endl;
   // ----------

   return false;
   }
 else // non ho sforato
   {
     // --- DEBUG ---
     // std::cout << sono passato per outofborder << std::endl;
     // ----------
     return this->insertPt(xr,yr,xpt,ypt); // richiamo questa stessa funzione..potrebbe riaccadere che esco ancora...servono magari 2 ingrandimenti
   }
}
else // caso normale
{

 /* IDEA
 1) Inserisci punto nuovo senza aggiornare visitati.(Crea i rami e i sottoalberi)
 2) Aggiorno i visitati usando Bresenham (flag per visitati)
 3) Riuso Bresenham e pulisco la flag dei visitati.
 N.B.: Questo procedimento serve per non aggiornare pi\'u volte il vis di un nodo.
 */

 // uso la insert del Node
 // [1]

 root->insertFinalPt(xpt,ypt);

 // --- DEBUG ---
 //std:: cout << Punto inserito: << xpt <<  <<ypt << std::endl;
 // ----------


 // [2]
```

```
int x0,x1,y0,y1; // le variabili per bresenham sono intere
int temp; // valore temporaneo
/*
x0= (int)round(xr);
y0 = (int)round(yr);
x1 = (int)round(xpt);
y1 = (int)round(ypt);
*/
// Il round d\'a  problemi ad esempio se inserisci un punto a 17.5
x0= (int)floor(xr);
y0 = (int)floor(yr);
x1 = (int)floor(xpt);
y1 = (int)floor(ypt);
int xpttemp=(int)floor(xpt);
int ypttemp=(int)floor(ypt);
bool steep= (abs(y1 - y0) > abs(x1 - x0)); // pendenza

if (steep)
{
   // swap(x0, y0)   swap(x1, y1)
   temp=x0; x0=y0; y0=temp;
   temp=x1; x1=y1; y1=temp;

}
if (x0 > x1)
{
   //swap(x0, x1)    swap(y0, y1)
   temp=x0; x0=x1; x1=temp;
   temp=y0; y0=y1; y1=temp;
}

 int deltax = x1 - x0;
 int deltay = abs(y1 - y0);
 int error = -deltax / 2;
 int ystep;
 int y = y0;

  if (y0 < y1)
    ystep = 1;
  else ystep = -1;
```

```
  for (int x=x0; x<=x1; x++)
  {
    if (steep)
       {
         if (((y==xpttemp) &&( x==ypttemp)) || !( (y<=(xpttemp+1) && y>=(xpttemp-1))
&& (x<=(ypttemp+1) && x>=(ypttemp-1))))
       root->visit(y,x);
       }
    else
       {
         if (((x==xpttemp) &&( y==ypttemp)) || !( (x<=(xpttemp+1) && x>=(xpttemp-1))
&& (y<=(ypttemp+1) && y>=(ypttemp-1))))
        root->visit(x,y);}

    error = error + deltay;
    if (error > 0)
     {
         y = y + ystep;
         error = error - deltax;
     }
  }

 // [3]

 deltax = x1 - x0;
 deltay = abs(y1 - y0);
 error = -deltax / 2;
 y = y0;

  if (y0 < y1)
    ystep = 1;
  else ystep = -1;

  for (int x=x0; x<=x1; x++)
  {
    if (steep)
       {
         if (((y==xpttemp) &&( x==ypttemp)) || !( (y<=(xpttemp+1) && y>=(xpttemp-1))
&& (x<=(ypttemp+1) && x>=(ypttemp-1))))
```

```
        root->clean(y,x);
        }
    else
      {
        if (((x==xpttemp) &&( y==ypttemp)) || !( (x<=(xpttemp+1) && x>=(xpttemp-1))
&& (y<=(ypttemp+1) && y>=(ypttemp-1))))
        root->clean(x,y);
        }
    error = error + deltay;
    if (error > 0)
     {
        y = y + ystep;
        error = error - deltax;
     }
  }



 return true;
}

}

// Node *n \'e sempre uguale a root del quadtree
bool Quadtree::outOfBorder(const float &x,const float &y,Node *n, bool &limitReached)
{
if ((x < n->getCornX()) || (y < n->getCornY()) || (x >= (n->getCornX() + n->getDim())
) || (y >= (n->getCornY()+n->getDim()) ) )
 {
  // allora siamo fuori dai limiti
  //1) devo sapere dove cade il punto e avere il nuovo corner del padre
  uint dimroot= (n->getDim())*2;
  if (dimroot > MAXDIM)
    {
     limitReached=true; // non posso ingrandirmi...devo segnalarlo al quadtree
     return true; //controllo a livello sopra su limitreached
    }
  else
     {
      //caso tipico...
```

```
      // richiamo la funzione che crea la root nuova di dimensioni doppie
      fatherCorner(n,x,y);
      return true; // sono uscito dai limiti
     }

  }
else return false;  // return false...non \'e fuori dai limiti

}


void Quadtree:: fatherCorner(Node *n,float x, float y)
{
// childpos \'e la posizione 0,1,2,3 nell'array del padre del *n
// xcorn e ycorn sono i corner del padre

// posso uscire dal quadrato in alto in basso a sx o dx.

if (x < n->getCornX()) // punto cade a sinistra
  {
   if (y > (n->getCornY()+n->getDim())) // pt cade a sx e in alto
     {
      // il figlio va a finire in 4 cio\'e pos 3
      createNodesandPutN(n,3 );
     }
    else // se il punto cade a sinistra e in basso o sx e basta
      {
       //lo metto in childpos=1 (posizione 2)
       createNodesandPutN(n,1 );
      }
  }
else // cade a destra o \'e dentro
  {
   if (y > (n->getCornY()+n->getDim())) // pt cade a dx e in alto
     {
      // il figlio va a finire in 3 cio\'e pos 2
      createNodesandPutN(n,2 );
     }
    else // se il punto cade a dx e in basso o dx e basta
      {
```

```
        //lo metto in childpos=0
        createNodesandPutN(n,0 );
       }
  }

}


void Quadtree:: createNodesandPutN(Node *n,int childpos)
{
uint fatherdim=(n->getDim())*2;
Node *temproot;
switch (childpos)
{
case 0: temproot= new Node(false,0,0,fatherdim,n->getCornX(),(n->getCornY()-n->getDim()));
break;
case 1: temproot= new Node(false,0,0,fatherdim,(n->getCornX()-n->getDim()),(n->getCornY()-
n->getDim())); break;
case 2: temproot= new Node(false,0,0,fatherdim,n->getCornX() ,n->getCornY()); break;
case 3: temproot= new Node(false,0,0,fatherdim,(n->getCornX() -n->getDim()),n->getCornY());
break;
default : break;
};
//inoltre si devono creare 3 nodi nuovi e associare al posto giusto i nodi
int array[4][2];
temproot->newCorner( array);
// abbiamo i nuovi corner
Node *children[4];
for (int j=0; j<4; j++)
  {
   if (j!= childpos)
     children[j]=new Node(true,0,0,n->getDim(),array[j][0],array[j][1]);
   else
     children[j]=n; // associo n in pos giusta
  }
for (int j=0; j<4; j++)
  {
   temproot->setChild(j,children[j]); // li associo tutti
  }
```

```
// adesso \'e associato...dovrei mettere questa come root
root=temproot;
}
```

main.cpp

```
/***************************************************************************
 *   Copyright (C) 2007 by Mauro Brenna, Ivan Reguzzoni                    *
 *   malloblenne@gmail.com                                                 *
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 3 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 *   This program is distributed in the hope that it will be useful,       *
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of        *
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the         *
 *   GNU General Public License for more details.                          *
 *                                                                         *
 *   You should have received a copy of the GNU General Public License     *
 *   along with this program;If not, see <http://www.gnu.org/licenses/>.   *
 *                                                                         *
 ***************************************************************************/

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <iostream>
#include <cstdlib>
#include quadtree.h
#include <fstream>
#include <cstdio>
#include <cstdlib>
#include <vector>
#include <cmath>

using namespace std;



int main(int argc, char *argv[])
{
  //cout << Hello, world! << endl;
```

```
Quadtree *q;
fstream f;
uint maxdim;

maxdim= 65000;//(rangeof(int));          // limite dell'int

//cout << maxdim << endl;
q=new Quadtree(maxdim,512);                 //256cm il quadratone
//cout << *q << endl;

float xr=251,yr=251,theta=0,tempx,tempy;
//nel quadrato simulato sono   xr=47; yr=47; theta=0;
//float xr=47,yr=47,theta=0,tempx,tempy;

uint cnt=1;

vector<float> r, angle;
// il primo scan
 for(cnt=0; cnt<4;cnt++)
 {
  //primi 5 scan sono nella stessa posizione...non fare scanmatching(ROBUSTEZZA)
  q->getScan(cnt,r,angle);
  for (int k=0; (uint)k<r.size(); k++)
    {
      tempx=xr+r.at(k)*cos(theta+angle.at(k));
      tempy=yr+r.at(k)*sin(theta+angle.at(k)); // wrt absolute frame
      q->insertPt(xr,yr,tempx,tempy);
    }
 }

// Limite di 65k file !
int i = cnt;
while ( i ) //loop
 {
   r.clear();angle.clear();
   if(!q->Scanmatcher2(xr,yr,theta,i,r,angle)) break;


   // No ScanMatcher
   //if(!q->getScan(i,r,angle))
```

```
    //  break;

    // ---- DEBUG ----
    // cout << xr <<     << yr <<     <<theta << endl<<endl<<endl;
    // cout << r.size();
    // ----------

    for (int k=0; (uint)k<r.size(); k++)
      {
        tempx=xr+r.at(k)*cos(theta+angle.at(k));
        tempy=yr+r.at(k)*sin(theta+angle.at(k)); // wrt absolute frame
        q->insertPt(xr,yr,tempx,tempy);
      }

    i++;
  }



  // Disegna robot
  char buffer[30];
  sprintf(buffer,%s,argv[1]); //ogni ciclo 1 file
  string stringona= buffer;
  stringona= log/+stringona;
  f.open(stringona.c_str(), ios::out);
  f << *q;
  f << image.fillColor(\red\); ;
  f << image.draw( DrawableRectangle((<< xr << -x_o)*mul,( << yr << -y_o)*mul,( <<
xr << -x_o)*mul+mul,( << yr << -y_o)*mul+mul ) );;
  f.close();

  // delete q;

  return EXIT_SUCCESS;

}
```

Appendice B

Il manuale utente

Questa sezione può essere considerata un utile riferimento per un utilizzo corretto del software presentato.
Il software è stato testato sotto sistemi GNU/Linux[1] e l'uso di tale sistema operativo è consigliato, anche se non strettamente necessario, per sfruttare gli script di utilità scritti in bash.

Per il funzionamento del software sono necessari un compilatore `C++` (si consiglia `g++`) la libreria `GNU Scientific Library`[2] e la libreria `ImageMagick++`[3] per la creazione di immagini. Per testare il programma è possibile eseguire lo script **giratutto.sh** che utilizza tutti i dataset presenti nella cartella *logfiles* e restituisce le immagini della mappa risultante nella cartella *immagini*. Il programma fornito funziona di default nel modo seguente: utilizza i file presenti, in numero progressivo, nella cartella *logfiles* e, dopo aver costruito la mappa, restituisce un file log appropriato nella cartella *log*. Il file risultante è semplicemente una serie di istruzioni interpretabili dalla libreria `ImageMagick++`. Per la creazione di un'immagine di tipo bitmap è necessario usare lo script **creator.sh** che concatena i tre file **premessa log/nomecartella finale**, compila il programma e lo esegue.

È ovviamente possibile modificare parametri ed ottenere un comportamento e/o una mappa differente.
Nel file `node.h` è possibile modificare la **MAXRES**, cioè la grandezza minima della cella nella mappa. Per cambiare la dimensione in pixel nell'immagine ottenuta basta modificare **mul** nel file `premessa`. Modificando il

[1] Distribuzioni: Ubuntu (6.10, 7.04), Gentoo, Archilinux (Voodoo, Duke, Don't Panic!)
[2] utilizzata per le funzioni random e reperibile all'indirizzo: http://www.gnu.org/software/gsl/
[3] http://www.imagemagick.org/

main in modo opportuno si può scegliere se eseguire lo scan matching o inserire i dati in modo 'grezzo' utilizzando solo **getScan** e **InsertPt**. Nella cartella `altri_cpp_per_prove_etc` è possibile trovare differenti main di simulazione. Per ottenere file log di tipo corretto è utile sfruttare i driver originali del sensore per GNU/Linux sostituendo il file `singleScan.cpp` con quello fornito nel pacchetto software e successivamente eseguire lo script `cancellaprima.sh`. Per semplificare l'acquisizione delle scansioni è possibile utilizzare `hok.sh`. È possibile integrare l'acquisizione del sensore nel pacchetto software senza l'utilizzo di logfile e quindi privilegiando un uso in real time semplicemente facendo l'overridding della funzione membro pubblica **getScan**. Con lo stesso procedimento è possibile modificare la funzione di *scan matching*.
Si consiglia di mantenere la parte di creazione mappa tramite struttura dati quadtree indipendente dalla libreria `ImageMagick++` per non peggiorare i tempi di elaborazione, poiché la creazione di immagini richiede molto tempo.
Infine lo script `backup.sh` può essere considerato un utile strumento per salvare tutti i file prima di una modifica particolare e, in alcuni casi, potrebbe essere utilizzato insieme a cron per backup periodici.