AIMETA - XIII Convegno Italiano di Meccanica Computazionale - Brescia, 13-15 novembre 2000

IMPLEMENTAZIONE OBJECT-ORIENTED DELLE STRUTTURE DATI DI UN CODICE DI CALCOLO MULTIBODY GENERAL-PURPOSE

P. RIGHETTINI, A. TASORA

Politecnico di Milano, D.S.T.M.
P.za L. da Vinci 32, 20133 Milano, Italy
righettini@mech.polimi.it tasora@mech.polimi.it

Sommario

Nel presente lavoro si prendono in esame le problematiche e gli accorgimenti relativi ad un'implementazione razionale di un codice di calcolo per simulazioni meccaniche multicorpo, con particolare attenzione alla definizione delle strutture dati del programma. Tali scelte progettuali, inerenti la struttura intrinseca del software, si delineano nella fase iniziale dello sviluppo di un'applicazione e condizionano in modo sensibile il risultato dell'intero progetto. La realizzazione di un programma multibody con una vasta gamma di funzioni, in grado di soddisfare le esigenze dei progettisti alle prese con molteplici tipologie di simulazioni meccaniche, ha comportato lo studio di una particolare architettura software con caratteristiche di modularità, espandibilità ed elevata efficienza computazionale. Il kernel di simulazione così ottenuto è in grado di modellare e risolvere un'ampia casistica di problemi meccanici caratterizzati dalle peculiarità tipiche dei meccanismi reali, quali non linearità, collisioni, vincoli con giochi, attriti, motori ed attuatori con leggi di moto arbitrarie.

1 Introduzione

Durante l'ultimo decennio si è osservato nell'ambito della progettazione meccanica un crescente interesse per soluzioni software in grado di verificare il funzionamento e le prestazioni di meccanismi complessi per sistemi d'automazione prima della loro costruzione, con evidenti vantaggi che ciò comporta in termini di costi, tempi di progetto e cicli di prototipazione. Questo problema necessita della soluzione di due distinti problemi, da un lato è necessaria un'opportuna interfaccia per inserire in modo facile ed immediato i dati del problema, dall'altro si deve disporre di un *kernel* di simulazione, elemento centrale del sistema complessivo, in grado di risolvere un'ampia gamma di problemi *multibody* di interesse per l'automazione meccanica. La parte di interfaccia verso il codice multibody deve fornire metodi rapidi ed intuitivi per introdurre nel modello fisico tutte le entità che caratterizzano i meccanismi reali, ad esempio leggi non lineari, vincoli olonomi di vario genere, campi di forze, molle, motori, attuatori, collisioni fra superfici, vincoli monolateri o con giochi, ecc. L'introduzione e la descrizione di tali entità dovrebbe poter avvenire tanto con il tradizionale metodo di elaborazione batch di listati testuali, tanto quanto attraverso un'interfaccia grafica evoluta, con la quale sia possibile manipolare le parti meccaniche nello spazio tridimensionale con la facilità di un CAD o di un modellatore solido.

È auspicabile che il kernel di calcolo fornisca un elevato numero di funzioni di base come simulazione dinamica, cinematica, analisi statica, analisi di assemblaggio, cinematica interattiva, supporto di collisioni, registrazione dei risultati, ma è altrettanto necessario che in tali mansioni il kernel di simulazione dimostri un'elevata velocità di calcolo, ad esempio per ridurre i tempi morti qualora si faccia uso di un'interfaccia grafica interattiva, nonché per raggiungere, quando possibile, l'obiettivo della simulazione real time. Altrettanto opportuna sarebbe la possibilità di gestire modelli complessi mediante sotto-strutture organizzate in forma gerarchica (ad esempio un oggetto 'autoarticolato' può essere diviso negli oggetti 'motrice' e 'rimorchio', ecc.) Infine il kernel di calcolo dovrebbe essere impostato secondo i principi della modularità, in modo da facilitare miglioramenti futuri nonché per consentire l'aggiunta di funzioni specializzate da parte dell'utente.

A questo punto è bene osservare che per quanto concerne la maggior parte dei software multibody attualmente in commercio è difficile che tutte le prerogative fin qui esaminate coesistano contemporaneamente. Ad esempio fra i software in grado di rappresentare vaste tipologie di meccanismi generici (DADS^{TM}, ADAMS^{TM} ecc.) si osserva come la ricerca della massima astrazione possa talora incidere negativamente sull'efficienza di calcolo o sull'immediatezza d'uso, così come d'altro canto esistono programmi specializzati orientati alla robotica il cui impiego è veloce ed intuitivo, ma a discapito della versatilità e genericità. Solitamente tali compromessi sono conseguenze della scelta dei metodi risolutori, che possono condizionare l'orientamento dell'intero programma.

Infatti esistono diverse strategie per la risoluzione automatizzata dei problemi multicorpo, ed è noto che alcune si prestano meglio di altre alla soluzione di determinate classi di problemi [11]. In particolare, si osserva come i metodi più diffusi differiscano principalmente per la scelta delle coordinate di stato che descrivono il sistema fisico; ad esempio l'approccio *in coordinate cartesiane naturali* (detto anche *lagrangiano totale*) introduce un numero elevato di coordinate, proporzionalmente al numero di corpi rigidi, e considera tutti i vincoli mediante moltiplicatori di Lagrange, mentre all'estremo opposto vi è l'approccio *in coordinate ricorsive* (o relative, o vincolari) che introduce il minor numero possibile di coordinate per la descrizione del modello [5], [10]. Generalmente quest'ultima strategia comporta la soluzione di sistemi ODE (equazioni differenziali ordinarie) facilmente e velocemente risolvibili con metodi numerici tradizionali, mentre l'approccio in coordinate cartesiane implica la soluzione di problemi DAE (differenziali-algebrici, a causa dell'aggiunta delle equazioni di vincolo). Tuttavia, a fronte di questo vantaggio, i metodi in coordinate ricorsive richiedono analisi topologiche dei meccanismi, e spesso devono ricorrere anch'essi all'introduzione di moltiplicatori lagrangiani per trattare catene cinematiche chiuse [2].

In questo lavoro viene presentata la struttura software di un codice multibody che permette di far coesistere, per quanto possibile, le doti di compattezza, astrazione, genericità, facilità d'uso ed efficienza computazionale. Viene anche accennata la possibile implementazione nel contesto di un'interfaccia grafica avanzata.

I formalismi risolutori del simulatore presentato sono stati impostati secondo il metodo *lagrangiano totale*, ovvero in coordinate cartesiane naturali, ritenendo che questo metodo possa offrire migliori garanzie di applicabilità ad una classe vasta di problemi generali, soprattutto in vista di un'applicazione con interfaccia grafica dove l'utente possa inserire/cancellare sia corpi rigidi che vincoli durante il funzionamento del software, cambiando in continuazione la topologia dei meccanismi [6]. Inoltre il funzionamento mediante moltiplicatori lagrangiani garantisce il supporto di qualsiasi tipo di vincolo olonomo, mentre le formulazioni in coordinate ricorsive sono spesso limitate ad una ristretta varietà di giunti prismatici/rotoidali, tipicamente quelli usate in applicazioni di robotica. Infine, a giustificare la scelta dell'approccio lagrangiano, vi sono particolari accorgimenti e metodi numerici che rendono agevole la soluzione di problemi DAE [13], [15], [11], ad esempio traendo vantaggio dalla loro sparsità [14], o calcolando gli jacobiani delle equazioni di vincolo con metodi altamente efficienti [4].

Sono stati pertanto messi in evidenza alcuni requisiti che reputiamo fondamentali per l'archi-

tettura software di un simulatore multibody general-purpose completo ed innovativo, e che sono stati applicati durante la fase di sviluppo del codice di calcolo, denominato CHRONO.

In primo luogo è necessario scegliere formalismi efficienti, compatti, di validità generale, applicabili a meccanismi costruiti con qualsivoglia numero di corpi e con connessioni topologiche qualsiasi. Inoltre è di fondamentale importanza l'impiego di tecniche di programmazione che seguano l'approccio OOP (Object Oriented Programming), in particolare per garantire modularità, scalabilità e coerenza del codice. Tale obiettivo si è perseguito scegliendo il linguaggio C++ come strumento di programmazione. Altra prerogativa rilevante è il poter disporre di una base di dati modificabile dinamicamente, per consentire l'aggiunta/rimozione di corpi o di vincoli anche durante l'esecuzione della simulazione, pertanto caratterizzata dalla gestione run-time delle strutture dati che definiscono i corpi e i vincoli.

Per quanto concerne l'efficienza computazionale, si mette in luce la necessità di ottimizzare la velocità di esecuzione degli algoritmi, in modo da rendere competitiva la scelta del programma general-purpose rispetto a procedure numeriche specializzate. A questo proposito si rammentano i vantaggi dell'impiego dei quaternioni quali coordinate di rotazione: oltre ad evitare singolarità nelle trasformazioni, diversamente da qualsiasi set di tre angoli [8], [18], i quaternioni consentono l'adozione di un particolare formalismo da noi sviluppato [4] al fine di rendere efficiente il calcolo degli jacobiani.

Infine, per facilitare la definizione delle leggi di moto con cui i progettisti di macchine automatiche impongono il movimento agli attuatori (end-effector) che costituiscono la parte operativa della macchina, e più in generale per poter assegnare funzioni non lineari arbitrarie ai parametri che definiscono il modello multicorpo, si è data particolare importanza all'implementazione di strutture dati preposte alla descrizione di funzioni del tipo $x = f(t)$.

2 Impostazione dei formalismi risolutori

In virtù della scelta dell'approccio lagrangiano, si introduce il vettore di stato s contenente posizioni e velocità di tutti gli n corpi rigidi del sistema:

$$\vec{s} = \left\{ \begin{array}{c} \vec{q} \\ \dot{\vec{q}} \end{array} \right\} \ , \ \ \vec{s} = \left\{ \vec{q}_1, \ldots, \vec{q}_n, \dot{\vec{q}}_1, \ldots, \dot{\vec{q}}_n \right\}^t \tag{1}$$

laddove la posizione $\vec{q}$ dell'i-esimo corpo rigido è data da un vettore epta-dimensionale, unione di una parte tridimensionale $\vec{p}$ che esprime l'origine della terna mobile nello spazio cartesiano assoluto, ed una parte tetradimensionale $\vec{r}$ che esprime la rotazione assoluta sotto forma di quaternione unitario:

$$\vec{q}_n = \left\{ \begin{array}{c} \vec{p}_n \\ \vec{r}_n \end{array} \right\} \tag{2}$$

Si introducono anche i vincoli, sotto forma di un vettore di equazioni scalari olonome (eventualmente anche dipendenti dal tempo, se trattasi di vincoli reonomici come nel caso di attuatori), del tipo:

$$\vec{C} = \vec{C}(\vec{q}, t) \ , \ \ \vec{C} = \{C_1(\vec{q}, t), \ldots, C_n(\vec{q}, t)\}^t \tag{3}$$

Dall'equazione di Lagrange per la dinamica:

$$\left\{ \frac{d}{dt} \left(\frac{\partial E_c}{\partial \dot{\vec{q}}} \right) \right\}^t - \left\{ \left(\frac{\partial E_c}{\partial \vec{q}} \right) \right\}^t + [C_q]^t \vec{\lambda} = \vec{Q}_F \tag{4}$$

con opportuni passaggi [5] si ricavano i termini del sistema lineare che fornisce le accelerazioni e le reazioni incognite per un dato sistema, noto il suo stato:

$$\left\{ \begin{array}{lcl} [M_{qq}]^t \ddot{\vec{q}} + [C_q]^t \vec{\lambda} & = & \vec{Q}_F + \vec{Q}_M \\ [C_q]^t & = & \vec{Q}_C \end{array} \right. \tag{5}$$

anche:

$$\left\{ \begin{bmatrix} [M_{qq}] & [C_q]^t \\ [C_q] & [0] \end{bmatrix} \begin{Bmatrix} \ddot{\vec{q}} \\ \vec{\lambda} \end{Bmatrix} = \begin{Bmatrix} \vec{Q}_F + \vec{Q}_M \\ \vec{Q}_C \end{Bmatrix} \right. \tag{6}$$

Dal momento che si sono scelti i quaternioni come coordinate di rotazione, vanno aggiunti anche i vincoli scalari C_u sulla norma unitaria dei quaternioni [4], uno per ogni corpo rigido, pertanto il sistema si trasforma nel seguente:

$$\left\{ \begin{bmatrix} [M_{qq}] & [C_q]^t [C_u]^t \\ [C_q] & [0] \\ [C_u] & \end{bmatrix} \begin{Bmatrix} \ddot{\vec{q}} \\ \vec{\lambda} \\ \vec{\lambda}_u \end{Bmatrix} = \begin{Bmatrix} \vec{Q}_F + \vec{Q}_M \\ \vec{Q}_C \\ \vec{Q}_u \end{Bmatrix} \right. \tag{7}$$

I procedimenti che abbiamo elaborato per derivare i termini $[M_{qq}]$, $\vec{Q}_C$, $\vec{Q}_U$, $\vec{Q}_F$, $\vec{Q}_M$ $[C_u]$ e $[C_q]$ sono esposti approfonditamente in [4] e [6], laddove viene discusso in particolare l'originale formalismo che abbiamo sviluppato per calcolare analiticamente le matrici jacobiane dei vincoli $[C_q]$.

Ad esempio, nel caso di tre corpi rigidi, con due corpi vincolati fra loro da un giunto sferico ed un terzo corpo libero nello spazio, la matrice dei coefficienti assume l'aspetto mostrato in figura 1. Si osservi che in quest'esempio si ottiene una matrice di dimensione pari a

$$L = 7N_c + N_{vq} + N_{vs} = 8N_c + N_{vs} = 27 \ ,$$

avendo indicato con N_c il numero di corpi, con N_{vq} il numero di vincoli imposti dalla norma unitaria dei quaternioni e con N_{vs} il numero di vincoli scalari. Il numero di gradi di libertà risulta

$$N_{dof} = 7N_c - N_{vq} - N_v = 15 \ ,$$

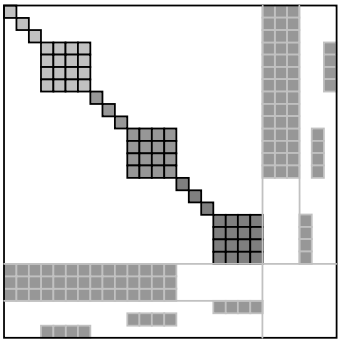


Figura 1: *esempio di matrice dei coefficienti del sistema 7*

come confermato dall'analisi cinematica classica.

Simili sistemi lineari presentano caratteristiche di elevata sparsità e di simmetria, quindi possono essere risolti con accorgimenti mirati alla massima velocità di esecuzione [14]. La soluzione del sistema 7, all'istante t, consente di ottenere la derivata dello stato:

$$\dot{\vec{s}} = \begin{Bmatrix} \dot{\vec{x}} \\ \ddot{\vec{x}} \end{Bmatrix} = f(\vec{s}, t) \tag{8}$$

che viene utilizzata dall'integratore numerico per procedere lungo l'asse temporale della simulazione dinamica (es. integrazione con metodo di Runge-Kutta, etc.). Pertanto, ai fini dell'analisi dinamica, il codice di calcolo dev'essere in grado di assemblare la matrice dei coefficienti del sistema 7, nonché il vettore dei termini noti, a partire da qualsiasi lista di vincoli, corpi rigidi e forze esterne.

Si rammenta infine la possibilità di variare la dimensione dei termini e delle matrici anche nel corso della simulazione. Tale facoltà è particolarmente utile per contemplare gli effetti introdotti dalle discontinuità nel numero di vincoli o di gradi di libertà, ad esempio a causa dell'apertura della chiusura di vincoli monolateri.

3 Classi di dati e strutture

Ai fini di un'organizzazione razionale dei dati all'interno del codice di calcolo, la scelta della formulazione lagrangiana esposta nel capitolo precedente suggerisce la suddivisione delle informazioni sulla scorta degli oggetti fisici che compongono il modello multicorpo, in particolare corpi rigidi, sistemi di riferimenti ausiliari, forze applicate e vincoli. In virtù di tale considerazione l'utilizzo del linguaggio C++ il quale, aderendo alla filosofia OOP, consente la suddivisione di dati e funzioni in "classi" dotate di ereditarietà. In dettaglio, una classe C++ definisce dati e funzioni che rappresentano gli aspetti comuni di una determinata tipologia di oggetti: una delle conseguenze positive di quest'approccio è che se alcune classi mostrano caratteristiche in comune, queste ultime possono essere raccolte in una classe di base dalle quali si ereditano classi specializzate che implementano solamente le caratteristiche non condivisibili. Tale criterio comporta una strutturazione delle classi in forma di albero genealogico, con innegabili vantaggi in termini di unificazione dei concetti, razionalità e coerenza nella struttura del codice, modularità dei sorgenti, agevolazioni nelle fasi di debugging, manutenzione e comprensione del codice [3].

Questo tipo di approccio è facilmente applicabile alle entità che compongono un modello multibody, laddove è possibile ravvisare caratteristiche in comune fra i diversi oggetti. Ad esempio è possibile usare le medesime funzioni di trasformazione di coordinate per tutti gli oggetti (corpi rigidi, riferimenti ausiliari, ecc.) che presentano moti rigidi nello spazio tridimensionale.

A titolo esemplificativo si consideri il modello di figura 2, costituito dai due corpi rigidi **b_A** e **b_B** vincolati fra loro dalla cerniera **c_B**, a formare un doppio pendolo vincolato a terra dalla cerniera **c_A** che si muove orizzontalmente secondo una legge di moto imposta. Si è introdotta una molla torsionale in **c_A**, uno smorzatore con caratteristiche non lineari in **c_B**, ed infine si è applicato un vettore forza al corpo **b_B**. Ad ogni corpo rigido sono associati i sistemi di riferimento ausiliari (chiamati *marker*) utilizzati per definire i vincoli, ad esempio il corpo **b_B** comprende il marker **m_3** necessario alla costruzione della cerniera **c_B**. Si osservi che si è introdotto il corpo rigido **b_G** con lo scopo di fornire un sistema di riferimento assoluto, a cui appartiene il marker mobile **m_0** utilizzato per il vincolo a terra. Il corpo **b_G** è comunque immobile e non compare nel vettore di stato.

Dall'illustrazione si evince come sia possibile suddividere il modello nelle sue parti salienti, ovvero corpi rigidi, marker e vincoli, alle quali possono corrispondere analoghi oggetti C++. Fra l'altro, l'identificazione delle classi di dati costitutive del modello si può sviluppare fino al punto da rappresentare le funzioni sotto forma di oggetti autonomi, ad esempio la legge di moto $x = x(t)$ imposta al marker **m_O** può essere considerata a sua volta un oggetto di tipo "funzione", collegato all'oggetto "marker" mediante un puntatore, ed analogamente si può fare per le leggi non lineari che definiscono le proprietà della molla torsionale in **c_A** e dello smorzatore in **c_B**. Più in generale, si può dire che l'architettura del software CHRONO è fondata sul criterio per cui ogni oggetto, se possibile, dev'essere scomposto in più parti indipendenti, alle quali competono classi specifiche.

Seguono, per le classi più rilevanti, le descrizioni sintetiche dei dati e delle funzioni in esse implementati.

3.1 Classi per algebra, calcolo e funzioni

La classe di base **CH_function** rappresenta una generica funzione scalare del tipo $y = f(x)$.

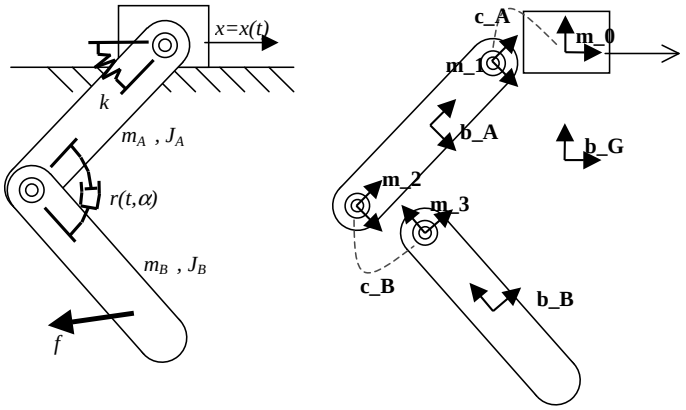
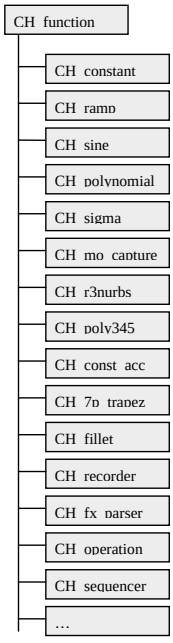


Figura 2: *esempio di sistema multicorpo: doppio pendolo*

Tale classe costituisce la base sulla quale si articolano, in via ereditaria, numerosi oggetti-funzione. Questi vengono ampiamente utilizzati all'interno del nostro codice, in quanto rappresentano un mezzo efficiente e modulare per la descrizione di generiche funzioni, ad esempio leggi di moto, comportamenti non lineari di molle e smorzatori, alzate di camme. La struttura della classe di base non comprende alcun dato, ma le classi ereditate implementano i dati di cui hanno bisogno (ad esempio la classe **CH_sine** implementa fase, frequenza, ampiezza, offset).



Fra le funzioni-membro salienti, citiamo quelle che permettono di ottenere $y = f(x)$ così come le derivate dy/dx e d^2y/dx^2, usando la differenziazione numerica di base. Le sotto-classi ereditano di default questo meccanismo per ottenere le derivate numeriche, ma in alternativa alcune di esse possono implementare funzioni specifiche per ottenere la propria derivata in forma analitica, quando questa sia nota. Altre funzioni di base consentono inoltre di effettuare integrazione numerica, quadratura, FFT, calcolo di coefficiente di accelerazione e di velocità, output su file e grafici. Fra le classi ereditate da questa classe, e qui non descritte singolarmente per motivi di spazio, citiamo di seguito le più interessanti. Innanzitutto vi sono funzioni algebriche elementari, ad esempio **CH_const** (funzione costante $y = C$), **CH_ramp** (funzione $y = A + Bx$), le funzioni trigonometriche come **CH_sine**, **CH_cosine**, **CH_tan**, la funzione polinomiale generica di grado N, ovvero **CH_poly**.

Sono state implementate specifiche funzioni per descrivere le leggi di azionamento di comune impiego nel settore dell'automazione: **CH_const_acc** è la legge di azionamento "ad accelerazione costante", **CH_poly345** è la legge di azionamento "polinomiale 3-4-5", **CH_7p_trapez**: rappresenta la legge "trapezioidale a 7 tratti" e **CH_cyclo** descrive la legge di azionamento "cicloidale". Alcune classi **CH_function** sono state sviluppate con l'obiettivo di gestire dati forniti da terze parti o comunque costituiti da sequenze di campionamenti da interpolare, come **CH_mo_capture** che contiene la registrazione di campionamenti di traiettorie (motion capture data) o **CH_recorder** che registra il valore assunto da una variabile durante una simulazione, per un uso in simulazioni successive o per un'uscita su file o su grafici. Si cita anche **CH_r3nurbs**, definita da una curva NURBS (non-uniform rational B-spline) che interpola ordinate arbitrarie con il massimo della flessibilità e generalità d'uso.

Al fine di offrire la massima espandibilità del codice di calcolo, si è poi implementata la classe

CH_fx_parser, la quale valuta generiche espressioni simboliche, macro e programmi introdotti dall'utente con sintassi C-like. Di particolare rilievo è la classe **CH_operation**. Questo tipo di funzione svolge operazioni algebriche sulle sotto-funzioni che essa contiene: ad esempio può moltiplicare una funzione **CH_sine** per una **CH_cosine** al fine di ottenere $y = sin(x)cos(x)$. Funzioni sempre più complesse si ottengono nidificando più livelli di **CH_operation**.

In ultimo si cita l'importante classe **CH_sequencer**, che consente di mettere in sequenza più funzioni semplici per ottenere, ad esempio, leggi di azionamento complesse composte da molteplici e diverse leggi di alzata. Fra l'altro, permette di imporre vincoli di continuità di grado C_n fra i diversi tratti. In tale contesto risultano utili le funzioni **CH_sigma**, che raccorda due tratti costanti mediante una legge cubica, e la funzione **CH_fillet** che rappresenta un raccordo fra due funzioni generiche, con continuità di grado C_n.

La necessità di svolgere le tipiche operazioni richieste dai formalismi multibody comporta l'implementazione di numerose funzioni per la manipolazione delle matrici, per l'algebra lineare e per il calcolo numerico. A tale proposito si introduce la classe **CH_matrix**, che descrive una generica matrice NxN e per la quale sono disponibili varie operazioni. Da questa classe viene ereditata la **CH_sparse_matrix**, che implementa dati e funzioni tipiche delle matrici sparse e che viene usata per migliorare l'efficienza computazionale nella soluzione di sistemi lineari con numerose incognite, ad esempio il sistema 7. La struttura dati comprende il puntatore all'indirizzo delle celle e il numero di colonne e righe. Le funzioni membro implementano l'algebra delle matrici, metodi numerici LU, QR, soluzione di sistemi lineari simmetrici, ecc..

La classe **CH_vector** rappresenta un vettore cartesiano tridimensionale $\vec{x} = \{x, y, z\}^t$, i suoi dati sono costituiti dalle tre componenti x,y,z in forma di numeri floating point, mentre le funzioni riguardano varie operazioni di calcolo vettoriale: somma, prodotto, normalizzazione, ecc..

La classe **CH_quaternion** definisce il quaternione quale numero ipercomplesso a quattro dimensioni, $\vec{r} = e_0 + ie_1 + je_2 + ke_3 = \{e_0, e_1, e_2, e_3\}^t$. Nel contesto di questo software i quaternioni vengono utilizzati per rappresentare rotazioni rigide nello spazio tridimensionale, senza incorrere in singolarità o blocchi cardanici. Opportune operazioni di algebra dei quaternioni vengono introdotte con lo scopo di facilitare la manipolazione di questi dati e la loro applicazione a problemi di meccanica razionale. La struttura dati è costituita dalle quattro componenti scalari. Le funzioni-membro implementano varie operazioni di algebra dei quaternioni: prodotti, somme, norma, coniugazione, ecc.

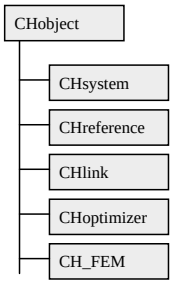
La classe **CH_coordsys** è una semplice unione delle classi vettore e quaternione, utile per rappresentare contemporaneamente la posizione e la rotazione di una terna cartesiana nello spazio tridimensionale, $\vec{q} = \{\vec{x}, \vec{r}\}^t$. La struttura dati è costituita da un oggetto CH_vector e da un oggetto CH_quaternion.

3.2 Oggetti Fisici

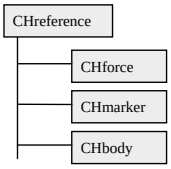
La classe di base **CH_object** accomuna tutti gli oggetti che compongono un modello fisico, i quali ne ereditano dati e funzioni. La sua struttura dati comprende i puntatori a CH_object precedenti e successivi nelle liste, nonché una serie di flag di impiego generico. Le funzioni membro implementano la gestione di flag, liste a puntatori, parsing dei simboli per la valutazione simbolica delle espressioni, input/output su file delle strutture dati in formato $ASCII$, input/output in formato binario con compatibilità inter-piattaforma.

CH_reference è la classe di base per tutti gli oggetti che possiedono una posizione nello spazio tridimensionale, in quanto rappresenta un sistema di riferimento cartesiano (terna mobile). La struttura dati comprende, oltre ai dati ereditati dalla classe CH_object, tre vettori CH_vector che rappresentano la posizione, la velocità e l'accelerazione "relative", ovvero dell'origine della terna rispetto al sistema di riferimento precedente. Seguono tre quaternioni CH_quaternion che rappresentano la rotazione, la velocità di rotazione e l'accelerazione relative della terna in forma

di numeri ipercomplessi e di loro derivate. Dal momento che in certe situazioni può risultare laborioso l'impiego di quaternioni e quaternioni derivati rispetto al tempo, in questa struttura dati compaiono anche numerosi dati ausiliari per una rappresentazione alternativa delle rotazioni (matrici di rotazione e loro derivate, vettori di velocità ed accelerazione angolare, ecc.) Altri dati ausiliari vengono introdotti al fine di ottimizzare la velocità di esecuzione del codice, ad esempio vi sono posizioni e rotazioni assolute che si potrebbero ricavare a partire da quelle relative, ma la frequenza con la quale si accede a tali informazioni è così elevata da giustificarne l'implementazione come dati della classe piuttosto che come valori di return di funzioni apposite. Le funzioni membro consentono di eseguire trasformazioni di coordinate attraverso questo sistema di riferimento. Vi sono inoltre specifiche funzioni in grado di operare conversioni fra rappresentazioni alternative: quaternioni, angoli di Eulero, angoli di Cardano, angoli HPB, parametri di Rodriguez, matrici.



La classe **CH_force**, derivata da CH_reference, rappresenta momenti o forze applicate ai corpi rigidi nello spazio. Il punto di applicazione è rappresentato dai dati ereditati dalla classe CH_reference. La struttura dati comprende inoltre il valore del modulo della forza ed un versore CH_vector che ne indica la direzione in coordinate assolute. Una serie di flag esprime diverse modalità di funzionamento della forza, ad esempio se la direzione ruota con il corpo cui è applicata, se il punto di applicazione si muove con il corpo, ecc. Un oggetto CH_function può essere usato per indicare una generica legge con la quale varia il modulo della forza nel tempo. Per la massima genericità d'uso, si possono anche usare tre distinti oggetti CH_function per esprimere la forza nelle sue tre componenti X,Y,Z. Fra le funzioni membro, citiamo quella che consente di ottenere la forza in coordinate lagrangiane.



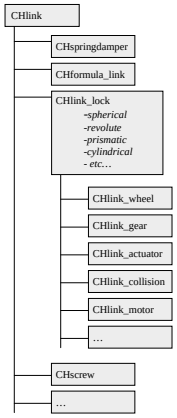
La classe **CH_marker**, derivata da CH_reference, rappresenta le terne mobili che vengono usate come riferimento per la costruzione di vincoli. A differenza dei semplici oggetti CH_reference, i marker possono essere dotati di specifiche leggi di moto rispetto al riferimento-padre. Infatti la struttura dati comprende -oltre ai dati ereditati da CH_reference- una serie di tre oggetti CH_function che vengono utilizzati per specificare il moto del marker nelle sue tre componenti x,y,z rispetto al riferimento padre, solitamente un corpo rigido.

La classe **CH_body**, derivata da CH_reference, rappresenta i corpi rigidi, ovvero le parti costitutive di un modello multibody. Si assume che la terna di riferimento coincida con il baricentro. La struttura dati comprende (oltre ai dati ereditati da CH_reference) informazioni relativamente al tensore d'inerzia ed il valore della massa complessiva. Di particolare importanza sono le liste degli oggetti **CH_marker** e **CH_force** che possono essere aggiunti al corpo, qualora si intenda vincolarlo o applicarvici una forza. Inoltre è possibile specificare un oggetto di tipo **CH_mesh** che viene usato per indicare la superficie del solido, secondo la topologia B-Rep. Tale mesh viene convertita in un albero binario OBB [17] per le verifiche di collisione in real-time, oppure viene usata per calcolare il tensore d'inerzia e la massa del corpo a partire dalla densità, con funzioni di quadratura. Altri dati di tipo booleano vengono impiegati per indicare se il corpo è vincolato a terra, se è visibile alle collisioni, se produce collisioni, se è disattivato, etc.

3.3 Vincoli

Essenzialmente si introduce l'oggetto di tipo "vincolo" ogni qual volta sia necessario vincolare fra loro due corpi rigidi. A partire da questa definizione, che accomuna tutti i vincoli di CHRONO, si è realizzato come sia possibile ereditare da una classe di base che implementa le caratteristiche generali di qualsiasi vincolo, le diverse altre classi "specializzate" che rappresentano vincoli di vario genere.

In particolare, fra le classi derivate, spicca per importanza una classe che accomuna tutti i vincoli per i quali è possibile usare il formalismo lock derivato [4], da noi sviluppato appositamente per consentire un efficiente calcolo analitico delle derivate delle equazioni di vincolo con evidenti benefici in termini di velocità di elaborazione.



Di fatto la classe di base **CH_link** viene utilizzata per rappresentare generici vincoli fra marker appartenenti a due corpi rigidi. Proprio da questa classe si ereditano numerosi vincoli con caratteristiche specifiche. Fra i principali dati appartenenti a questa classe di base, citiamo i puntatori ai due oggetti CH_marker vincolati (marker S e marker P il primo dei quali serve anche a definire il sistema di riferimento principale), i tre oggetti CH_coordsys con i quali si rappresentano la posizione, velocità ed accelerazione di P rispetto ad S, i tre vettori C=$\vec{C}$ C_dt=$\dot{\vec{C}}$ e C_dtdt=$\ddot{\vec{C}}$ che indicano le violazioni dell'equazione di vincolo e delle sue derivate, la matrice jacobiana Cq=$[C_q]$ ed i vettori Ct=$\vec{C}_t$ e Qc=$\vec{Q}_c$.

In particolare i termini $[C_q]$, $\vec{C}$, $\dot{\vec{C}}$ e $\ddot{\vec{C}}$, che vengono utilizzati dall'oggetto CH_system per le simulazioni cinematiche e statiche, devono poter variare di dimensioni a seconda del numero di equazioni scalari che compongono l'equazione del singolo vincolo. Un oggetto di tipo CH_link_mask contiene informazioni relativamente al numero di equazioni scalari utilizzate, sempre che alcune non vengano disattivate (ad esempio perché ridondanti o mal condizionate, informazioni sempre memorizzate in CH_link_mask).

Infine vi sono oggetti di tipo **CH_link_force** che possono essere usati per introdurre forze, momenti e cedevolezze elasto-plastiche non lineari fra i due marker, per i gradi di libertà residui. Le funzioni di questa classe svolgono essenzialmente due compiti: calcolano posizione, velocità ed accelerazione relative dei due marker, inoltre calcolano lo jacobiano $[C_q]$ ed i termini $\vec{C}_t$ e $\vec{Q}_c$ derivando numericamente qualsiasi equazione di vincolo del tipo $C_i(\vec{q}, t) = 0$.

Tuttavia le classi derivate possono operare un overloading di queste funzioni, per ottenere $[C_q]$, $\vec{C}_t$ e $\vec{Q}_c$ con procedimenti analitici quando sia desiderabile per ragioni di velocità, come nel caso della classe derivata CH_link_lock.

CH_screw, **CH_formula_link**, **CH_spring_damper**, ecc. sono esempi di classi derivate da CH_link. In particolare **CH_screw** rappresenta il vincolo vite-madrevite implementando la funzione che restituisce il residuo dell'equazione $C_i(\vec{q}, t)$, mentre il calcolo dello jacobiano $[C_q]$ e dei termini $\vec{C}_t$ e $\vec{Q}_c$ avviene ad opera delle funzioni di base ereditate da CH_link. **CH_formula_link** implementa un vincolo per il quale il legame cinematico fra i due marker è espresso da una generica formula simbolica scritta dall'utente. **CH_spring_damper** implementa un sistema molla-smorzatore che lega i due marker. Le caratteristiche di molla e smorzatore possono essere non lineari nel tempo, nella distanza e nella velocità di allontanamento, grazie all'impiego di specifici oggetti CH_function.

```
class CH_link  public CH_object     //cl.base vincoli
{
private:    // dati principali
    CH_flags  flags;    // stato del vincolo
    CH_link_mask*  mask;    // maschera attività eq.
    [..]
    CH_marker*  markerS;    // marker principale
    CH_marker*  markerP;    // marker secondario
    [..]
    Coordsys  relM; // posizione S-P
    Coordsys  relM_dt;  // velocità S-P
    Coordsys  relM_dtdt;    // accelerazione S-P
    [..]
    Matrix* C;  // residuo eq.vinc.
    Matrix* C_dt;   // d/dt residuo eq.vinc.
    Matrix* C_dtdt; // dd/dtdt res. eq.vinc.
    Matrix* Cq; // jacobiano vincolo
    Matrix* Ct; // der.parz. dC/dt
    Matrix* Qc; // vettore gamma
    [..]
    CH_link_force* forces;  // forze/coppie n.l.
    [..]
    [..]    // [..altri dati..]

public: // metodi
    [..]    // [..metodi..]
};
```

Figura 3: *principali dati della classe di base CH_link*

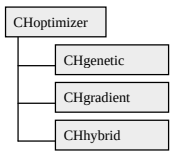
La classe **CH_link_lock**, derivata da CH_link, è in grado di rappresentare un elevato numero di vincoli olonomi di impiego comune (cerniere sferiche, cilindriche, guide prismatiche, giunti cardanici, giunti di Oldham, giunti Birfield, saldature, glifi, ecc.). Tale classe di vincoli fa uso della formulazione lock-derivata, che consente il calcolo degli jacobiani mediante un metodo analitico molto efficiente e compatto. Il procedimento di calcolo [4] si fonda sull'algebra dei quaternioni e costituisce una delle caratteristiche salienti dell'intero progetto software, in virtù dell'efficienza computazio-

nale e della facile applicabilità ad un'ampia categoria di vincoli. Si osservi che la medesima classe viene impiegata per rappresentare decine di vincoli differenti, in quanto i procedimenti di calcolo sono gli stessi. La struttura dati comprende l'indice che identifica il tipo di vincolo "lock-derivato" effettivamente voluto, alcune matrici e quaternioni ausiliari che vengono usati nel procedimento di calcolo, e una serie di oggetti CH_link_limit che specificano i limiti superiori/inferiori per i gradi di libertà del vincolo, se richiesti.

CH_link_actuator, **CH_link_gear**, **CH_link_wheel**, **CH_link_motor**, etc. sono esempi di classi derivate da CH_link_lock: tutti questi vincoli sfruttano il formalismo "lock-derivato" ereditato dalla classe genitrice, dato che questo si presta bene a descrivere vincoli olonomi e reonomici di vario genere [4], ma implementano dati e funzioni aggiuntive. Ad esempio, **CH_link_actuator** rappresenta un attuatore lineare con legge di azionamento imposta $s = s(t)$, pertanto tale legge corrisponde ad un dato CH_function. (Il vincolo CH_actuator comprende anche altri dati, fra i quali un flag che specifica se l'attuatore è in modalità di apprendimento) **CH_link_gear** rappresenta il vincolo fra due ruote dentate, sia interne che esterne, con vari parametri definibili dall'utente (inclinazione della retta d'azione, etc.) **CH_link_wheel** definisce il vincolo ruota-terreno, con un'ampia collezione di parametri configurabili mediante oggetti CH_function, ad esempio dipendenza non lineare del coefficiente di aderenza in funzione della pressione, ecc. Il vincolo **CH_link_motor**, fra i vari parametri caratteristici, introduce oggetti CH_function per definire motori con curve di coppia arbitrarie.

3.4 Altre Classi

Un oggetto della classe CH_system contiene l'intera descrizione del sistema fisico, ed è responsabile dell'esecuzione delle simulazioni statiche, cinematiche, dinamiche. La struttura dati comprende innanzitutto le liste dei corpi rigidi CH_body e dei vincoli CH_link, essenziali per la descrizione del modello multibody.



Fra i dati salienti citiamo inoltre la presenza di un oggetto del tipo CH_integrator che viene utilizzato per eseguire la simulazione dinamica, nonché di un oggetto CH_optimizer, utilizzato per sintesi/ottimizzazione di meccanismi. Seguono vari oggetti matematici di utilizzo globale (ad esempio la matrice di stato, il vettore di stato, il vettore dei moltiplicatori lagrangiani) e numerosi parametri di simulazione (tempo corrente, step di integrazione, tolleranze lineari ed angolari, ecc.). Le principali funzioni membro implementano analisi statica, dinamica, cinematica e di assemblaggio. Vi sono inoltre specifiche funzioni preposte alla gestione delle liste di corpi e vincoli.

La classe di base **CH_optimizer** raccoglie procedure matematiche orientate all'ottimizzazione di funzioni. Da questa classe si ereditano **CH_genetic**, per ottimizzazione globale con metodi genetici evolutivi, **CH_gradient** per l'ottimizzazione locale e **CH_hybrid** per un particolare tipo di ottimizzazione avanzata, orientata alla sintesi di meccanismi articolati [7]. Quest'ultimo metodo fa uso di operatori genetici specializzati.

La classe **CH_integrator** definisce l'interfaccia di base per il funzionamento delle procedure di integrazione numerica, ad esempio la funzione che ricava lo stato dopo un passo di integrazione, ovvero $s(t + dt) = f(s(t), ds(t)/dt)$. Segue la necessità di predisporre nell'oggetto CH_system la funzione adibita al calcolo della derivata dello stato rispetto al tempo, ovvero $ds(t)/dt$, ricavabile mediante la formula 7.

Le specifiche funzioni di integrazione numerica vengono implementate dalle classi ereditate, ad esempio **CH_int_Eulero**, che utilizza il metodo di Eulero, o **CH_int_Runge** che impiega il metodo di Runge-Kutta del quart'ordine, con stima dell'errore locale. Sono stati implementati anche altri metodi (Runge Kutta con coefficienti di Cash-Karp, metodo di Heun, di Eulero modificato). Si sono previsti opportuni algoritmi per la variazione del passo di integrazione (entro limiti fissati

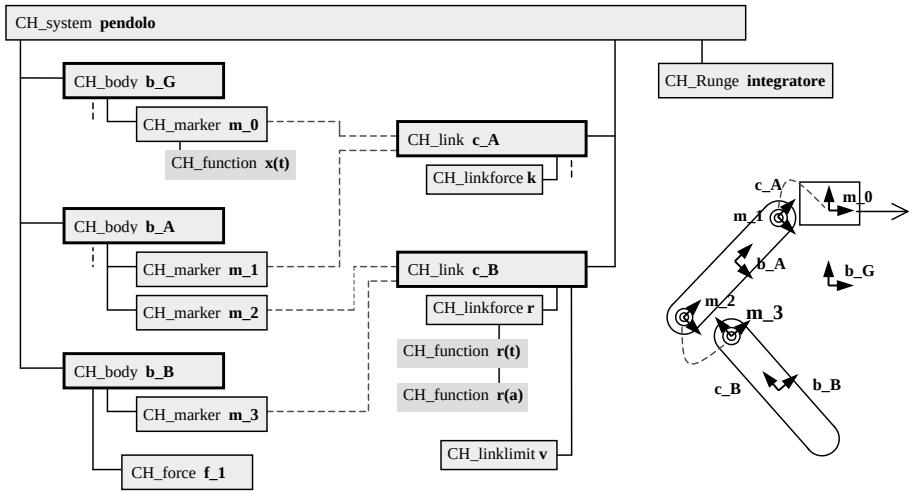
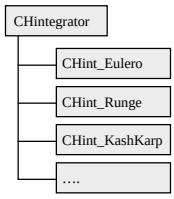


Figura 4: *base di dati corrispondente al sistema "doppio pendolo"*

dall'utente) in modo da affrontare problemi "stiff" in maniera efficiente sotto il profilo dello sfruttamento delle risorse di calcolo. Oltre alla stima dell'errore locale, vi sono altri eventi che possono portare l'integratore a ripetere il passo di integrazione, ad esempio il verificarsi di eventi discreti (collisioni), la divergenza dello stato istantaneo rispetto al valore predetto da un algoritmo che usa l'estrapolazione di Lagrange, infine un'eccessiva violazione delle equazioni algebriche di vincolo (mentre violazioni al di sotto di una certa tolleranza vengono corrette ad ogni step mediante algoritmi iterativi [13] o mediante stabilizzatori di Baumgarte [15]).



Le classi **CH_link_limit**, **CH_link_force**, **CH_link_mask**, vengono usate per rappresentare strutture di dati relative ai vincoli. In dettaglio, ogni vincolo può essere fornito di più oggetti **CH_link_force**, ognuno dei quali ha il compito di definire coppie e forze agenti lungo uno specifico grado di libertà. Ad esempio si può associare un oggetto CH_link_force al grado di libertà di traslazione di una guida prismatica, per rappresentare un effetto di attrito viscoso. La struttura CH_link_force è composta da vari oggetti CH_function che possono descrivere forzanti, molle e smorzatori agenti lungo il grado di libertà del giunto, anche con caratteristiche non lineari. Gli oggetti della classe **CH_link_limit** definiscono limiti superiori e/o inferiori per le coordinate di rotazione o traslazione di un giunto. Il superamento del limiti monolateri può essere gestito in due modi: sia rappresentando un urto dotato di uno specifico coefficiente di restituzione, sia introducendo forze di campo non lineari che descrivano l'effetto di deformazione del materiale che impedisce il moto (in tal caso le caratteristiche elasto-plastiche del materiale vengono descritte tramite oggetti CH_function). La classe **CH_link_mask** gestisce in maniera efficiente le informazioni relative allo stato delle equazioni scalari di vincolo utilizzate in un singolo oggetto CH_link. Infatti ognuna di queste condizioni può essere: attiva, disattivata, temporaneamente disattivata, ridondante, mal condizionata, ecc.

4 Struttura della base di dati

La definizione di un modello da sottoporre all'analisi dinamica implica che il software sia in grado di creare le istanze delle strutture dati del sistema multicorpo e che possa organizzarle in un database a puntatori. Innanzitutto vi è un oggetto **CH_system** che, come già illustrato, contiene

informazioni relative alle impostazioni dell'integratore, parametri globali, vettori e matrici di stato. Tale oggetto "sistema" comprende fondamentalmente due liste a puntatori, una per i corpi rigidi (oggetti **CH_body**) ed una per i vincoli (oggetti ereditati da **CH_link**). Si rammenta la possibilità, già citata nell'introduzione, di poter togliere ed aggiungere oggetti da tali liste anche durante il corso della simulazione, ottenendo così un "database dinamico". Ogni corpo rigido della classe **CH_body** a sua volta può contenere un numero illimitato di oggetti della classe **CH_force** e della classe **CH_marker**, organizzati in due liste a puntatori. Ogni oggetto **CH_link**, rappresentando un vincolo fra due sistemi di riferimento, possiede due puntatori che indicano i due marker vincolati; tali puntatori sono perciò responsabili della topologia del meccanismo. Ovviamente la rimozione di un marker o di un intero corpo rigido può comportare la disattivazione dei vincoli che vi facevano riferimento, pertanto la gestione dei puntatori dai vincoli ai marker va effettuata con particolare cautela. Con riferimento al grafo del database, si osservi come il sistema di figura 2 (un semplice doppio pendolo con forzanti, molle e smorzatori non lineari) si traduca nella corrispondente base di dati, composta da oggetti collegati da puntatori. Caratteristica saliente di questo modello di database è la possibilità di rappresentare meccanismi con un numero qualsiasi di elementi (corpi, vincoli) e con qualsiasi topologia (catene cinematiche multiple, aperte, chiuse), semplicemente grazie all'uso accorto di liste e puntatori.

Si accenna infine al metodo con il quale si svolge l'aggiornamento automatico della condizione degli oggetti fisici in funzione del vettore di stato $\vec{s}$: con una sola operazione **CH_system**::**Update()** si aggiornano in cascata tutte le variabili dei corpi rigidi e dei vincoli appartenenti al sistema (ad esempio ogni vincolo calcola, in funzione del nuovo stato $\vec{s}$ del sistema, il residuo $C(\vec{q},t)$, lo jacobiano $[C_q(\vec{q},t)]$, ecc). A sua volta ogni corpo rigido, dopo aver aggiornato la propria posizione, aggiorna anche i marker e le forze ad esso associate. In pratica ogni oggetto derivato da CH_object implementa una funzione del tipo *Update()* che aggiorna le proprie variabili in funzione dello stato dell'oggetto chiamante, e che propaga gli aggiornamenti alle ramificazioni dell'albero gerarchico del database chiamando le funzioni *Update()* degli oggetti figli.

5 Implementazione dell'interfaccia

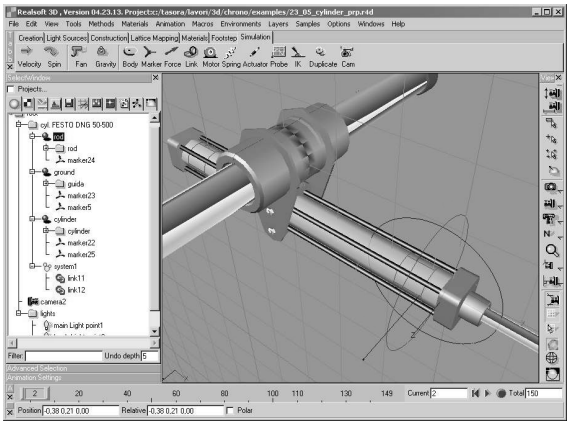


Figura 5: *editor grafico interattivo*

Vista la flessibilità e la modularità del kernel sviluppato secondo l'architettura esposta nei paragrafi precedenti, è stato possibile trasferire il codice di calcolo all'interno di un modellatore solido sviluppato da terze parti [19]. In questo modo gli strumenti di simulazione multibody diventano un modulo aggiuntivo per il modellatore, che pertanto può coniugare la propria capacità di modellare parti meccaniche con la possibilità, offerta dal nostro software, di simulare il funzionamento dei meccanismi così progettati.

In particolare si sottolinea la possibilità di aggiungere e rimuovere corpi rigidi e vincoli durante le fasi di progetto e simulazione: ciò è reso possibile dalla particolare struttura del database dinamico visto in precedenza. In particolare il modellatore gestisce i propri oggetti mediante una gerarchia ad albero [19]: in tale contesto i corpi rigidi, i marker e gli altri oggetti del nostro software sono implementati come "directory" dell'albero, che possono contenere altri oggetti (ad esempio un corpo rigido può contenere i marker

ed altri oggetti che non riguardano la simulazione multicorpo, ad esempio le superfici Nurbs che il modellatore usa per rappresentare i solidi).

Tale organizzazione ad albero dei modelli, quasi una metafora di un file-sistem dotato di directory, è di fondamentale importanza in quanto consente di organizzare modelli molto complessi in più sotto-meccanismi che possono essere collaudati, salvati e manipolati separatamente.

Nella figura 5, che rappresenta un cilindro pneumatico montato su un manicotto, si può osservare l'albero degli oggetti a sinistra della vista prospettica: si notino i corpi rigidi, i marker, i due vincoli e vari oggetti geometrici introdotti con gli strumenti standard del modellatore. Il nostro software trae beneficio dalle avanzate funzioni dell'interfaccia grafica del modellatore, ad esempio è possibile modificare le coordinate di vincoli e corpi rigidi semplicemente muovendoli sullo schermo, usufruendo della vista prospettica con *real-time shading*. Fra l'altro abbiamo sfruttato le funzioni di cinematica sottovincolata [6] fornite dal nostro codice multibody per poter verificare la cinematica dei meccanismi appena assemblati, muovendone interattivamente le parti con il mouse. Apposite finestre asincrone non-modali consentono l'accesso a tutte le informazioni relative ai parametri di simulazione ed alle proprietà di ogni oggetto introdotto nel modello, come in figura 6. Le funzionalità delle classi sono state incrementate per trarre profitto dall'interazione con il modellatore, ad esempio sono possibili le operazioni di undo, redo, cut and paste, salvataggio in binario su file assieme agli oggetti geometrici, drag and drop, ecc. Si rammenta infine la possibilità di salvare i risultati delle simulazioni sia sotto forma di grafici, sia come animazioni tridimensionali dal realismo fotografico, grazie ad avanzati algoritmi di ray-tracing.

Figura 6: *finestra per l'editing dei principali parametri dei vincoli*

6 Esempio

A dimostrazione delle caratteristiche salienti del software CHRONO, citiamo un'applicazione relativa allo studio delle proprietà cinematiche e dinamiche di un particolare robot parallelo a tre gradi di libertà, con azionamenti pneumatici, in cui il parallelismo fra end-effector e telaio è garantito dall'impiego di sei giunti cardanici. Il modello multicorpo di tale robot, realizzato con l'ausilio dell'interfaccia grafica, consiste essenzialmente di 14 corpi rigidi (3 alberi, 6 crociere cardaniche, 3 carrelli, 1 end-effector, 1 telaio) collegati fra loro da 15 vincoli (12 cerniere dei giunti cardanici, 3 guide prismatiche). Il sistema è definito da 91 coordinate generalizzate ed 88 equazioni scalari di vincolo: di fatto i rimanenti tre gradi di libertà diventano nulli se, al fine di studiare la cinematica del robot, si introduce un ulteriore vincolo del tipo "punto su traiettoria" applicato all'end-effector. In tal modo la simulazione cinematica fornisce i grafici delle coordinate dei giunti in funzione delle coordinate cartesiane dell'end-effector (cinematica inversa). E' possibile anche la cinematica diretta, in cui si impongono gli spostamenti agli attuatori e si verificano le posizioni dell'end-effector nello spazio cartesiano.

Al fine di analizzare le proprietà dinamiche del robot, si sono introdotti tre vincoli del tipo "attuatore lineare" ai carrelli, ai quali sono state assegnate le leggi di moto ricavate in precedenza con la simulazione della cinematica inversa. L'analisi dinamica ha fornito l'andamento delle reazioni e delle accelerazioni nei vincoli durante sei secondi di simulazione, nei quali il robot ha compiuto alcune operazioni standard. Sulla scorta dei risultati delle simulazioni dinamiche e cinematiche sono stati dimensionati i componenti del robot. La realizzazione materiale del robot ha infine confermato la validità delle previsioni del nostro software multicorpo.

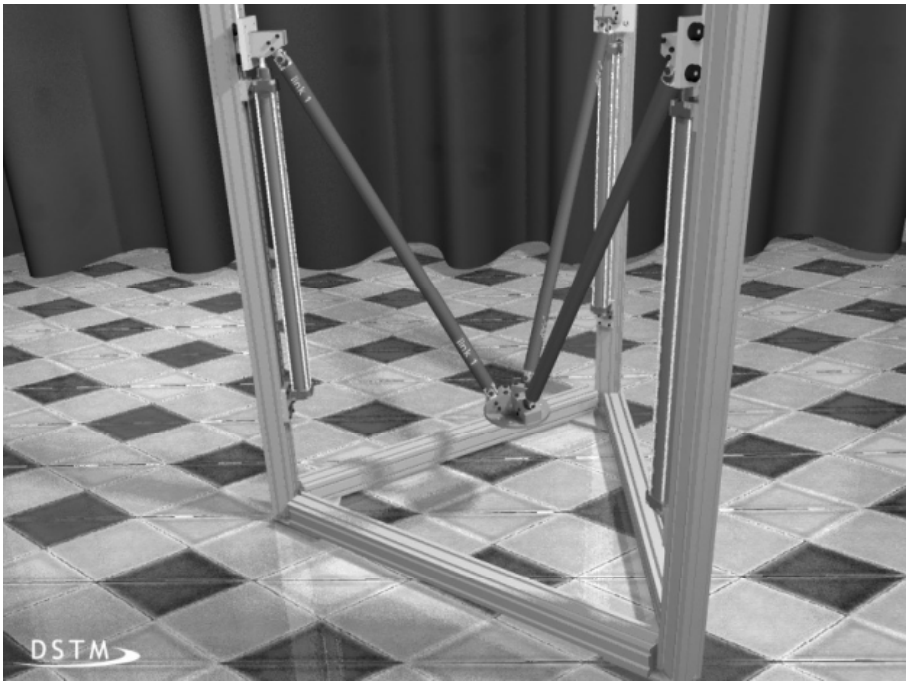


Figura 7: *simulazione ed animazione di un robot parallelo*

7 Conclusioni

Nel contesto della simulazione di sistemi dinamici multicorpo si è sviluppata un'architettura software particolarmente efficiente, laddove la strutturazione dei dati mediante l'approccio della programmazione orientata agli oggetti ha consentito di raggiungere un elevato grado di modularità e di razionalizzazione del codice. Il codice di calcolo è stato sviluppato in vista di una sua applicazione alla soluzione di problemi ingegneristici di vario genere, grazie alla possibilità di introdurre un'ampia gamma di entità fisiche (giunti, forze, motori) ed alla capacità di assegnarvi proprietà avanzate grazie a particolari oggetti preposti alla descrizione di funzioni non lineari. Il software è stato dotato di un'interfaccia grafica che sfrutta opportunamente la capacità di gestire basi di dati dinamiche strutturate in modalità gerarchica, da cui segue la facoltà di aggiungere, rimuovere o manipolare parti di meccanismi anche nel corso delle simulazioni. Le prestazioni raggiunte dal programma CHRONO, che risultano elevate anche in virtù dell'impiego di particolari formalismi risolutori, suggeriscono futuri sviluppi in vista di nuove funzioni di simulazione impostate sulla medesima architettura software.

Riferimenti bibliografici

[1] V.Enderlein, H.Hermsdorf, A.Keil, *On the description of Multibody System Models, Advances in multibody computational dynamics*, EUROMECH 404, Lisboa 1999.

[2] D.Bae, E.Haug, *A recursive formulation for constrained mechanical system, Center for Computer Aided Design*, University of Iowa, 1986.

[3] B.Stroustrup, *The C++ programming language, 3rd edition*, A.Wesley 2000.

[4] A.Tasora, P.Righettini, *Application Of Quaternion Algebra To The Efficient Computation Of Jacobians For Holonomic-Rheonomic Constraints, Advances in multibody computational dynamics*, EUROMECH 404, Lisboa 1999.

[5] A.Shabana, *Multibody systems*, John Wiley & Sons, New York 1989.

[6] A.Tasora, *Simulazione multibody mediante algebra dei quaternioni*, Politecnico di Milano, DSTM, 1997.

[7] P.Righettini, A.Tasora, *Un algoritmo genetico ibrido orientato all'ottimizzazione di sistemi multicorpo: applicazione alla sintesi di esalateri*, AIMETA 1999, Como, Italy.

[8] R.Dejo, *A Vehicle Dynamics Primer, section 3, Driving Simulation Project*, Evans & Sutherland, 1987.

[9] J.Post, F.Pfeiffer, *Object oriented description of multibody systems based on substructures, Advances in multibody computational dynamics*, EUROMECH 404, Lisboa 1999.

[10] R.Roberson, R.Schwertassek, *Dynamics of multibody systems*, Springer Verlag, Berlin.

[11] J.Cuadrado, J.Cardenal, E.Bayo, *Modeling and solution methods for efficient real-time simulation of multibody dynamics*, Multibody System Dynamics vol.1 n.3, Kluwer, 1997.

[12] D.Baraff, *Rigid body simulation*, Siggraph workshop, 1994.

[13] W.Blajer, *Elimination of constraint violation and accuracy improvement in numerical simulation of multibody systems, Advances in multibody computational dynamics*, EUROMECH 404, Lisboa 1999.

[14] K.S.Kundert, *Sparse matrix techniques and their application to circuit simulation, Circuit analysis, simulation and design*, North Holland.

[15] D.Bae, S.Yang, *A stabilization method for kinematic and kinetic constraint equations, Real-time integration methods for mechanical system simulation*, NATO advanced research workshop, Utah, ed. Springer Verlag 1991.

[16] A.Watt, M.Watt, *Advanced animation and rendering techniques*, Addison Wesley, 1995.

[17] S.Gottschalk, M.C.Lin, D.Manocha, *OBB tree: a hierarchical structure for rapid interference detection*, Proc. of ACM Siggraph'96, 1996, 171-180.

[18] J.Funda, R.Taylor, R.Paul, *On homogeneous transforms, quaternions, and computational efficiency*, IEEE Transactions on Robotics and Automation, June 1990.

[19] *Realsoft3D SDK documentation, Realsoft OY*, Finland 2000.