Universitá degli Studi di Trento

Dipartimento di Ingegneria Industriale

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA MECCATRONICA

Tesi di Laurea

Algoritmi per la Valutazione del Contatto tra Pneumatico e Strada in Soft Real Time

Laureando:

Davide Stocco

Relatore:

Prof. Enrico Bertolazzi

Anno Accademico 2019 · 2020

Sommario

This dissertation details …

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli acronimi

Introduzione 1

1.1 Obiettivi

Il presente lavoro di tesi ha preso avvio dalla collaborazione tra il Dipartimento di Ingegneria Industriale dell'Università di Trento e AnteMotion S.r.l., azienda specializzata in realtà virtuale e simulazione *multibody* per il campo *automotive*. In particolare, il modello di veicolo e pneumatico precedentemente studiati da Larcher in [4] saranno integrati nel simulatore di guida in tempo reale di AnteMotion. Pertanto, lo sviluppo dei modelli è stato finalizzato a minimizzare i tempi di compilazione massimizzando invece l'accuratezza. La necessità di sviluppare un algoritmo che calcoli i parametri dell'interazione tra terreno (rappresentato con una *mesh* triangolare) e pneumatico (rappresentato come un disco indeformabile) getta le basi per il lavoro svolto.

1.2 Il problema

La simulazione risolve alcuni dei problemi relativi al mondo della progettazione in modo sicuro ed efficiente, senza la necessità di costruire un prototipo dell'oggetto fisico. A differenza della modellazione fisica, che può coinvolgere il sistema reale o una copia in scala di esso, la simulazione è basata sulla tecnologia digitale e utilizza algoritmi ed equazioni per rappresentare il mondo reale al fine di imitare l'esperimento. Ciò comporta diversi vantaggi in termini di tempo, costi e sicurezza.

Infatti, il modello digitale può essere facilmente riconfigurato e analizzato, mentre questo è solitamente impossibile o troppo oneroso del punto di vista di tempi e/o costi da fare con il sistema reale [5].

Al giorno d'oggi esistono numerosi modelli di veicolo e pneumatico. Certamente, più semplice è il modello più veloce è la risoluzione delle equazioni costituenti, quindi, a seconda delle applicazioni, dev'essere scelto il modello con la giusta complessità. Per la maggior parte delle applicazioni di guida autonoma, un modello semplice è adeguato per caratterizzare con un livello di dettaglio sufficiente il comportamento del veicolo, e poiché queste analisi sono molto spesso fatte con l'ausilio di *Hardware in the Loop* (HIL), il modello dinamico del veicolo dev'essere risolto in tempo reale con tipico passo di tempo di un millisecondo. Il vincolo di esecuzione in tempo reale implica la scelta un modello di veicolo che sia velocemente risolvibile, ciò significa che i modelli semplici con pochi parametri, di solito modelli lineari a due ruote, sono particolarmente adatti per questo tipo di applicazioni. Tuttavia, ci sono alcune situazioni che richiedono modelli più dettagliati, come ad esempio l'azione prodotta da un *Advanced Driver-Assistance Systems* (ADAS), ovvero una manovra di sicurezza come l'elusione degli ostacoli o una frenata di emergenza, poiché il veicolo è spinto nella maggior parte dei casi al limite delle sue prestazioni [3]. In queste condizioni di guida si devono tenere conto di molti fattori come ad esempio il comportamento degli pneumatici che, spostandosi nella regione non lineare, fa sì che i fenomeni transitori non siano più trascurabili. Questo implica la necessità di utilizzare un modello più dettagliato di quello utilizzato per la guida in condizioni *standard*.

L'accuratezza dinamica del modello è di grande rilevanza per ricavare previsioni realistiche delle prestazioni del veicolo e del sistema di controllo. È importante notare che modellare in modo esaustivo tutti i sistemi di un'auto sarebbe un compito estremamente arduo e a volte anche impossibile. Esistono quindi modelli empirici come il modello della *Magic Formula* di Hans Pacejka, che cerca di imitare il reale comportamento del sistema. Il calcolo dei parametri di questo tipo di modelli richiede l'interpolazione di un insieme di dati di grandi dimensioni, e può quindi essere numericamente inefficiente o comunque troppo oneroso in termini di tempo.

Lo scopo di questo lavoro si collega a quello già svolto da Larcher in [4] in cui, grazie a un modello di veicolo completo con 14 gradi di libertà ha fornito un modello in grado di catturare con un livello di dettaglio appropriato il comportamento del veicolo quando viene spinto alle massime prestazioni. La necessità di calcolare

in tempo reale i parametri di input per il modello di ruota scelto da [4] definisce l'obiettivo di questo lavoro. In particolare lo scopo è quello di scrivere una libreria in linguaggio `C++` che con alcuni semplici parametri in *input* come la denominazione *European Tyre and Rim Technical Organisation* (ETRTO) dello pneumatico e la posizione nello spazio, calcola i dati relativi all'interazione pneumatico strada quali il punto di contatto virtuale e l'inclinazione locale del piano strada. Il tutto cercando di minimizzare i tempi di compilazione.

Il Pneumatico 2

2.1 Introduzione

Gli pneumatici sono probabilmente i componenti più complessi di un'auto in quanto combinano decine di componenti che devono essere formati, assemblati e combinati assieme. Il successo del prodotto finale dipende dalla loro capacità di fondere tutti i componenti separati in un prodotto dal materiale coeso che soddisfa le esigenze del conducente [8]. Gli pneumatici sono caratterizzati da un comportamento altamente non lineare con una dipendenza da diversi fattori costruttivi e ambientali.

2.2 Geometria dello Pneumatico secondo ETRTO

Quando si fa riferimento ai dati puramente geometrici, viene utilizzata una forma abbreviata della notazione completa prevista dall'ente di normazione ETRTO. Assumendo di avere un pneumatico generico la notazione che identificherà la geometria sarà del tipo $a/b\mathrm{R}c$. Dove:

- a rappresenta larghezza nominale del pneumatico nel punto più largo;
- b rappresenta percentuale dell'altezza della spalla dello pneumatico in relazione alla larghezza dello stesso;
- c rappresenta il diametro dei cerchi ai quali lo pneumatico si adatta.

Facendo un esempio, 195/55R16 significherebbe che la larghezza nominale del pneumatico è di circa 195 mm nel punto più largo, l'altezza della spalla dello pneumatico è il 55% della larghezza, ovvero 107 mm in questo caso, e che il pneumatico si adatta a dei cerchi di 16 pollici di diametro. Con questa notazione è possibile calcolare direttamente il diametro esterno teorico dello pneumatico tramite la seguente:

$$\phi_e = \frac{2ab}{25.4} + c \quad [\text{in}] \qquad \phi_e = 2ab + 25.4c \quad [\text{mm}] \tag{2.1}$$

Riprendendo l'esempio usato sopra, il diametro esterno risulterà dunque 24.44 in o 621 mm.

Meno comunemente usato negli Stati Uniti e in Europa (ma spesso in Giappone) è una notazione che indica l'intero diametro del pneumatico invece delle proporzioni dell'altezza della parete laterale, quindi non secondo ETRTO. Per fare lo stesso esempio, una ruota da 16 pollici avrebbe un diametro di 406 mm. L'aggiunta del doppio dell'altezza del pneumatico (2×107 mm) produce un diametro totale di 620 mm. Quindi, un pneumatico 195/55R16 potrebbe in alternativa essere etichettato 195/620R16.

Anche se questo è teoricamente ambiguo, in pratica queste due notazioni possono essere facilmente distinte perché l'altezza della parete laterale di un pneumatico automobilistico è in genere molto inferiore alla larghezza. Quindi, quando l'altezza è espressa come percentuale della larghezza, è quasi sempre inferiore al 100% (e certamente meno del 200%). Al contrario, i diametri degli pneumatici del veicolo sono sempre superiori a 200 mm. Pertanto, se il secondo numero è superiore a 200, allora è quasi certo che viene utilizzata la notazione giapponese, se è inferiore a 200 allora viene utilizzata la notazione USA/europea.

2.3 La Modellizzazione dello Pneumatico

Le forze di contatto tra la superficie stradale e lo pneumatico possono essere descritte completamente da un vettore di forza risultante applicato in un punto specifico dell'impronta di contatto e da una coppia risultante, come illustrato nella Figura 2.2.

Come componenti cruciali per la movimentazione dei veicoli e il comportamento di guida, le forze degli pneumatici richiedono particolare attenzione soprattutto quando, assieme al comportamento stazionario, anche il comportamento non stazionario dev'essere considerato.

FIGURA 2.1: Esempio di misure, secondo la notazione ETRTO, riportate sulla spalla dello pneumatico.

F_x	forza longitudinale
F_y	forza laterale
F_z	forza verticale
T_x	coppia di sovrasterzo
T_y	resistenza al rotolamento
T_z	coppia di autoallineamento

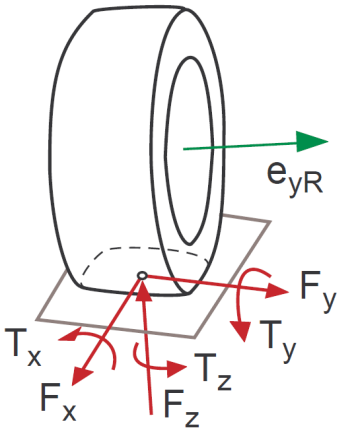


FIGURA 2.2: Forze e coppie generate dal contatto pneumatico-strada.

Attualmente, è possibile suddividere i modelli di pneumatico in tre gruppi:

- modelli matematici;
- modelli fisici;
- combinazione dei precedenti.

La prima tipologia di modello tenta di rappresentare le caratteristiche fisiche del pneumatico attraverso una descrizione puramente matematica. Pertanto questi tipi di modelli partono da un curve caratteristiche ricavate sperimentalmente e cercano

di derivare un comportamento approssimativo dall'interpolazione di un grande insieme di dati. Un esempio ben noto di questo approccio è il **modello di Pacejka** o ***Magic Formula*** [7]. Questo tipo di modellazione è adatta per la simulazione di guida dove il comportamento di interesse è per lo più la guidabilità del veicolo e le frequenze di uscita sono ben al di sotto delle frequenze di risonanza della cintura dello pneumatico. I modelli fisici o i modelli ad alta frequenza, come i modelli agli elementi finiti, sono in grado di rilevare fenomeni di risonanza a frequenza più elevata. Ciò permette di valutare la confortevolezza di guida di un veicolo. Dal punto di vista del calcolo, i modelli fisici complessi richiedono molto tempo al calcolatore per essere risolti, nonché di molti dati. Al contrario dei più veloci modelli matematici, che però richiedono un'accurata pre-elaborazione dei dati sperimentali. La terza tipologia di modelli consiste in un'estensione dei modelli matematici attraverso le leggi fisiche al fine di coprire una gamma di frequenza più ampia.

Il modello di pneumatico sviluppato nel modello di veicolo e il tipo di interfaccia di pneumatico/strada presentato da Larcher in [4] si basa sulla *Magic Formula* 6.2.

2.4 Il Modello della *Magic Formula*

2.4.1 La *Magic Formula*

Uno dei modelli di pneumatici più utilizzati è il cosiddetto modello *Magic Formula* sviluppato da Egbert Bakker e Pacejka in [1]. Questo modello è stato poi rivisto e l'ultima versione è riportata in [7]. Il modello *Magic Formula* consiste in una pura descrizione matematica del rapporto input-output del contatto pneumatico-strada. Questa formulazione collega le variabili di forza con lo slip rigido del corpo che vengono trattati nelle sezioni successive. La forma generale della funzione di descrizione può essere scritta come:

$$y(x) = D\sin\{C\arctan[B(x+S_h) - E(B(x+S_h) - \arctan(B(x+S_h)))]\} + S_v \tag{2.2}$$

dove:

- B rappresenta il fattore di rigidezza;
- C rappresenta il fattore di forma;
- D rappresenta il falore massimo della forza o coppia;
- E rappresenta il fattore di curvatura;

- S_v rappresenta lo spostamento in verticale della curva caratteristica;
- S_h rappresenta lo spostamento in orizzontale della curva caratteristica.

e dove $y(x)$ può essere la forza longitudinale F_x , la forza laterale F_y o la coppia di autoallineamento M_z, mentre x è la componente di slip corrispondente. In Figura 2.3 sono illustrate le curve caratteristiche generiche degli pneumatici derivate con il metodo della *Magic Formula*.

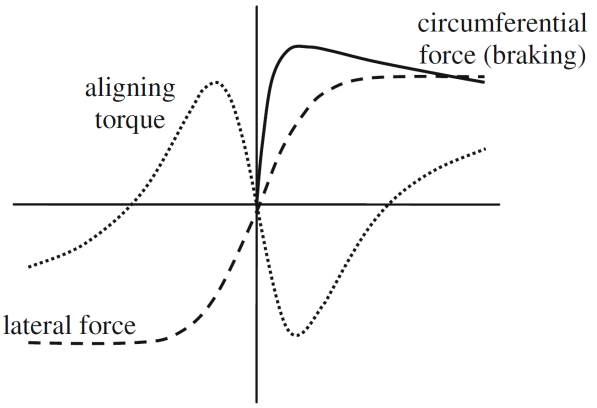


FIGURA 2.3: Curve caratteristiche generiche degli pneumatici derivate con il metodo della *Magic Formula*

2.4.2 Contatto con la Superficie Stradale

La posizione e l'orientamento della ruota in relazione al sistema fissato a terra sono dati dalla terna di riferimento del vettore ruota RF_{wh_i}, che viene calcolata istante per istante risolvendo le equazioni dinamiche del sistema ottenuto nel Capitolo 2 in [4]. Supponendo che il profilo stradale sia rappresentato da una funzione arbitraria a due coordinate spaziali del tipo:

$$z = z(x, y) \tag{2.3}$$

su una superficie irregolare, il punto di contatto P non può essere calcolato direttamente. Così, come prima approssimazione siamo in grado di identificare un punto $P^\star$, che è definito come una semplice traslazione del centro ruota M:

$$P^\star = M - R_0 \boldsymbol{e}_{zC} \begin{bmatrix} x^\star \\ y^\star \\ z^\star \end{bmatrix} \tag{2.4}$$

dove R_0 è il raggio dello pneumatico indeformato e $\boldsymbol{e}_{zC}$ è il vettore unitario che definisce l'asse z_c del sistema di riferimento del vettore ruota.

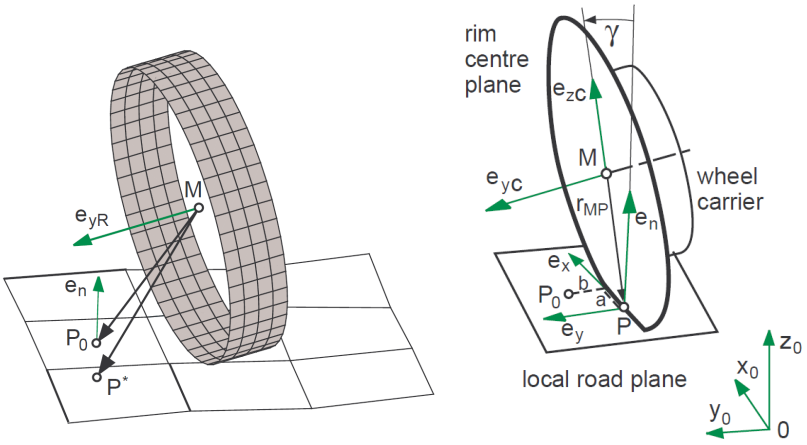


Figura 2.4: Geometria del contatto pneumatico-strada.

La prima stima del sistema di riferimento del punto di contatto $RF_{PC^\star}$ è una terna con origine in $P^\star$ e orientazione degli assi definiti dall'orientazione del sistema di riferimento della ruota.

$$RF_{PC^\star} = \left[\begin{array}{ccc|c} & [R_{RF_{wh}}] & & \begin{matrix} x^\star \\ y^\star \\ z^\star \end{matrix} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2.5}$$

Ora, i vettori di unità $\boldsymbol{e}_x$ ed $\boldsymbol{e}_y$, che descrivono il piano locale nel punto P, possono essere ottenuti dalle seguenti equazioni:

$$\boldsymbol{e}_x = \frac{\boldsymbol{e}_{yC} \times \boldsymbol{e}_n}{|\boldsymbol{e}_{yC} \times \boldsymbol{e}_n|} \qquad \boldsymbol{e}_y = \boldsymbol{e}_n \times \boldsymbol{e}_x \tag{2.6}$$

Al fine di ottenere una buona approssimazione del piano strada locale in termini di inclinazione longitudinale e laterale, sono stati utilizzati i quattro punti di campionamento $(Q_1^\star, Q_2^\star, Q_3^\star, Q_4^\star)$ che sono rappresentati graficamente in Figura 2.5. I punti di campionamento sono definiti sul sistema di riferimento temporaneo del punto di contatto $RF_{PC^\star}$ e lo spostamento longitudinale e laterale sono definiti dall'origine, ovvero lo stesso $P^\star$. I vettori di spostamento sono definiti come:

$$\begin{aligned} {}_{PC^\star}\boldsymbol{r}_{Q^\star_{1,2}} &= \pm\Delta x \\ {}_{PC^\star}\boldsymbol{r}_{Q^\star_{3,4}} &= \pm\Delta y \end{aligned} \tag{2.7}$$

e quindi, i quattro punti di campionamento sono:

$$
\begin{aligned}
{}^{P^\star}\boldsymbol{r}_{Q^\star_{1,2}} &= P^\star \pm \Delta x \boldsymbol{e}_{xPC^\star} \\
{}^{P^\star}\boldsymbol{r}_{Q^\star_{3,4}} &= P^\star \pm \Delta y \boldsymbol{e}_{yPC^\star}
\end{aligned}
\tag{2.8}
$$

Al fine di campionare l'impronta di contatto nel modo più efficiente possibile, le distanze di Δx e Δy, dell'equazione precedente, vengono regolate in base al raggio del pneumatico indeformato R_0 e alla larghezza del pneumatico B. I valori di queste due quantità possono essere trovate in letteratura e sono $\Delta x = 0.1R_0$ e $\Delta x = 0.3B$. Attraverso questa definizione, si può ottenere un comportamento realistico durante la simulazione.

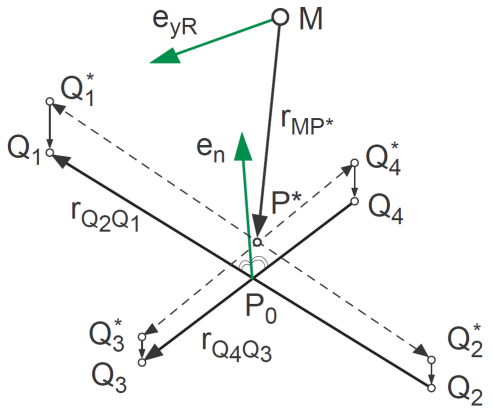


FIGURA 2.5: Punti campionati nel piano locale della superficie stradale.

Ora la componente z in corrispondenza dei quattro punti campione viene valutata attraverso la funzione $z(x, y)$ precedentemente definita. Quindi, aggiornando la terza coordinata dei punti di campionamento $Q^\star_i$, otteniamo i corrispondenti punti campione Q_i sulla superficie della pista locale. La linea fissata dai punti Q_1, Q_2 e rispettivamente Q_3, Q_4, può ora essere utilizzata per definire la normale al piano strada locale (Figura2.6). Pertanto, il vettore normale è definito come:

$$
\boldsymbol{e}_n = \frac{\boldsymbol{r}_{Q_1Q_2} \times \boldsymbol{r}_{Q_4Q_3}}{|\boldsymbol{r}_{Q_1Q_2} \times \boldsymbol{r}_{Q_4Q_3}|}
\tag{2.9}
$$

dove sono $\boldsymbol{r}_{Q_2Q_1}$ e $\boldsymbol{r}_{Q_4Q_3}$ sono i vettori che puntano rispettivamente da Q_1 a Q_2 e da Q_3 a Q_4. Applicando l'equazione 2.6 è ora possibile calcolare i vettori unitari $\boldsymbol{e}_x$ e $\boldsymbol{e}_y$ del piano di locale del punto di contatto. Il punto di contatto P si ottiene aggiornando le coordinate del primo punto di prova $P^\star$, con il valore medio delle

tre coordinate spaziali dei quattro punti campione.

$$P = \frac{1}{4} \begin{bmatrix} \sum_{i=1}^{4} x_i \\ \sum_{i=1}^{4} y_i \\ \sum_{i=1}^{4} z_i \end{bmatrix} \tag{2.10}$$

Infine possiamo mettere assieme tutte le componenti del piano di riferimento del punto di contatto finale ottenendo:

$$RF_{PC} = \left[\begin{array}{ccc|c} & & & x_P \\ [\boldsymbol{e}_x] & [\boldsymbol{e}_y] & [\boldsymbol{e}_z] & y_P \\ & & & z_P \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2.11}$$

Attraverso questo approccio, la normale del piano strada locale $\boldsymbol{e}_n$ insieme al punto di contatto locale P, sono in grado di rappresentare l'irregolarità della strada in modo soddisfacente. Come accade in realtà, bordi taglienti o discontinuità del manto stradale saranno smussate da questo approccio. Alcuni casi dimostrativi sono illustrati nella Figura 2.6.

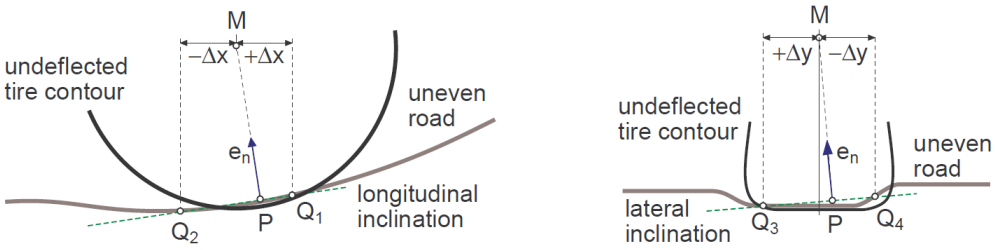


Figura 2.6: Inclinazione longitudinale e laterale del piano strada locale.

La Superficie Stradale 3

3.1 Introduzione

Oltre allo pnumatico, la superfice stradale rappresenta il secondo importante elemento che definisce il contatto. Perchè una superficie stradale possa essere facilmente utilizzata da un calcolatore deve essere prima discretizzata. La discretizzazione in questo caso avviene mediante la rappresentazione della superficie stessa in una *mesh* triangolare. La *mesh*, è contenuta in un file formato *Road Data File* (RDF), che contiene le posizione (x, y, z) di ogni vertice e i numeri di identificazione per ognuno dei tre vertici del triangolo, per ogni trangolo.

È importante notare che la discretizzazione del manto stradale è un passaggio molto importante in quando, se campionatonato troppo grossolanamente potrebbe influire negativamente sui risultati dei calcoli per l'estrazione del piano strada locale. In altre parole, una semplificazione troppo spinta, potrebbe causare degli errori tali da incorrere in risultati troppo approssimativi e non rispecchianti la realtà. Al contrario, una *mesh* troppo fitta, complica inutilmente i calcoli, dilatando sensibilmente i tempi di esecuzione. È bene quindi discretizzare più densamente in maniera oculata e solo dove occore realmente, ovvero in prossimità di cordoli, marciapiedi o qualsiasi tipo di ostacolo che potrebbe influire sulle performance della vettura.

3.2 Il Formato RDF

3.2.1 Superfici Semplici

Sfortunatamente, non esistono standard universalmente riconosciuti per il formato RDF. In linea di massima le superfici stradali sono definite nei *road data file* (`*.rdf`). Questa tipologia di file è composto da varie sezioni, indicate da parentesi quadre.

```
1  { Comments section }
2
3  [UNITS]
4  LENGTH = 'meter'
5  FORCE = 'newton'
6  ANGLE = 'degree'
7  MASS = 'kg'
8  TIME = 'sec'
9
10 [MODEL]
11 ROAD\_TYPE = '...'
12
13 [PARAMETERS]
14 ...
```

Nella sezione `[UNITS]`, vengono impostate le unità utilizzate nel file di dati stradali. La sezione `[MODEL]` viene invece utilizzata per specificare il tipo di strada, del tipo:

- `ROAD_TYPE = 'flat'`: come indica già il nome, si tratta di una superficie stradale piana.
- `ROAD_TYPE = 'plank'`: dove questa strada è composta da un singolo scalino o dosso orientato perpendicolarmente o obliquo rispetto all'asse X, con o senza bordi smussati.
- `ROAD_TYPE = 'poly_line'`: ovvero l'altezza della strada è in funzione della distanza percorsa.
- `ROAD_TYPE = 'sine'`: dove la superficie stradale è costituita da una o più onde sinusoidali con lunghezza d'onda costante.

La sezione `[PARAMETERS]` contiene parametri generali e parametri specifici del tipo di superficie stradale.

I parametri per ogni tipologia di superficie stradale sono elencati di seguito:

- Generali:
  - `MU`: è il fattore di correzione dell'attrito stradale (non il valore dell'attrito stesso), da moltiplicare con i fattori di ridimensionamento LMU del modello di pneumatico.
    Impostazione predefinita: `MU = 1.0`.
  - `OFFSET`: è l'offset verticale del terreno rispetto al sistema di riferimento inerziale.
  - `ROTATION_ANGLE_XY_PLANE`: è l'angolo di rotazione del piano XY attorno all'asse Z della strada, ovvero la definizione dell'asse X positivo della strada rispetto al sistema di riferimento inerziale.
- Strada con scalino:
  - `HEIGHT`: altezza dello scalino.
  - `START`: distanza lungo l'asse X della strada all'inizio dello scalino.
  - `LENGTH`: lunghezza dello scalino (escluso lo smusso) lungo l'asse X della strada.
  - `BEVEL_EDGE_LENGTH`: lunghezza del bordo smussato a $45°$ dello scalino.
  - `DIRECTION`: rotazione dello scalino attorno all'asse Z, rispetto all'asse Y della strada.
    Se lo scalino è posizionato trasversalmente, `DIRECTION = 0`. Se lo scalino è posto lungo l'asse X, `DIRECTION = 90`.
- Polilinea:
  Il blocco `[PARAMETERS]` deve avere un sottoblocco chiamato `(XZ_DATA)` e costituito da tre colonne di dati numerici:
  - La colonna 1 è un insieme di valori X in ordine crescente.
  - Le colonne 2 e 3 sono insiemi di rispettivi valori Z per la traccia sinistra e destra.

  Esempio:

```
1 [PARAMETERS]
2 MU = 1.0
3 OFFSET = 0.0
4 ROTATION_ANGLE_XY_PLANE = 0.0
5
```

```
{ X_road  Z_left  Z_right }
(XZ_DATA)
-1.0e04 0 0
0.0500  0 0
0.1000  0 0
0.1500  0 0
... ... ...
```

- Sinusoide:
  La strada a superficie sinusoidale è implementata come:

$$z(x) = \frac{H}{2}\left(1 - \cos\left(\frac{2\pi \cdot (x - x_i)}{L}\right)\right) \tag{3.1}$$

  dove

  - z: coordinata verticale della strada;
  - H: altezza;
  - x: posizione attuale;
  - x_i: inizio dell'onda sinusoidale;
  - L: semi-periodo dell'onda sinusoidale.

  I parametri sono:

  - `HEIGHT`: altezza dell'onda sinusoidale.
  - `START`: distanza lungo l'asse X della strada all'inizio dell'onda sinusoidale.
  - `LENGTH`: lunghezza dell'onda sinusoidale lungo l'asse X della strada.
  - `DIRECTION`: rotazione dell'onda sinusoidale attorno all'asse Z, rispetto all'asse Y della strada.
    Se l'onda sinusoidale è posizionata trasversalmente, `DIRECTION = 0`.
    Se l'onda sinusoidale è posta lungo l'asse X, `DIRECTION = 90`.

3.2.2 Superfici Complesse

Sfortunatamente, queste informazioni appena descritte permettono di costruire strade troppo semplicistiche e approssimative, che non rispecchiano la realtà. È quindi necessario inserire i risultati della discretizzazione della superficie stradale sopra citati.

Per descrivere una superficie stradale composta da una *mesh* di triangoli si userà la seguente formattazione del file.

- `[NODES]`: presenti nella prima sezione e dove vengono descritti sotto forma di una quartina (id, x, y, z) data dal numero di identificazione e dalle coordinate nello spazio.
- `[ELEMENTS]`: presenti nella seconda sezione e dove vengono descritti sotto forma di una quartina (n_1, n_2, n_3, μ) data dal numero di identificazione dei tre vertici componenti i-esimo triangolo e dal coefficente di attrito presente nella faccia.

Esempio:

```
1  [NODES]
2  { id x_coord y_coord z_coord }
3  0 2.64637 35.8522 -1.59419e-005
4  1 4.54089 33.7705 -1.60766e-005
5  2 4.52126 35.8761 -1.62482e-005
6  3 2.66601 33.7456 -1.57714e-005
7  4 0.771484 35.8282 -1.56367e-005
8  5 0.791126 33.7206 -1.5465e-005
9  ... ... ... ...
10
11 [ELEMENTS]
12 { n1 n2 n3 mu }
13 1 2 3 1.0
14 2 1 4 1.0
15 5 4 1 1.0
16 ... ... ... ...
```

Ulteriori parametri possono essere aggiunti prima della dichiarazaione dei nodi della *mesh*.

- `X_SCALE`: riscala i punti delle coordinate dei nodi lungo l'asse X.
- `Y_SCALE`: riscala i punti delle coordinate dei nodi lungo l'asse Y.
- `Z_SCALE`: riscala i punti delle coordinate dei nodi lungo l'asse Z.
- `ORIGIN`: definisce la posizione dell'origine della sistema di riferimento della superficie stradale.

- `UP`: definisce la direzione positiva dell'asse Z.
- `[ORIENTATION]`: ruota i punti delle coordinate dei nodi secondo la matrice definita.

Esempio:

```
1  X_SCALE
2  1000.0
3  Y_SCALE
4  1000.0
5  Z_SCALE
6  1000.0
7  ORIGIN
8  0 0 0
9  UP
10 0.0,0.0,1.0
11 ORIENTATION
12 1.0  0.0  0.0
13 0.0  1.0  0.0
14 0.0  0.0  1.0
```

Algoritmi 4

4.1 Parsificazione

4.1.1 Introduzione

La parsificazione o analisi sintattica è un processo che analizza un flusso continuo di dati in ingresso (letti per esempio da un file o una tastiera) in modo da determinare la correttezza della sua struttura grazie ad una data grammatica formale. Un *parser* è un programma che esegue questo compito. Nella maggior parte dei casi, l'analisi sintattica opera su una sequenza di *token* in cui l'analizzatore lessicale spezzetta l'input.

4.1.2 Parsificazione del formato RDF

Nel lavoro svolto è stato creato un algoritmo per pardificare i file di tipo RDF che descrivono superfici complesse. Purtroppo, come precedentemente detto, non esiste uno standard universalmente riconosciuto per questo formato. Creare dunque un *parser* o definire un generatore di parser è arduo. Si è quindi optato per la creazione di un *parser* che rilevi solo i nodi (`[NODES]`), li salvi temporaneamente e, dopo aver immagazzinato anche i dati relativi agli elementi (`[ELEMENTS]`), instanzi un oggetto *mesh*, composto dai nodi dichiarati nella sezione elementi. Gli altri parametri non sono stati considerati.

Come verrà richiamato nelle conclusioni, l'importanza di definire uno standard per il formato RDF è di cruciale importanza. In questo modo si potrà creare un generatore di parser con una grammatica e un lessico ben definiti, nonché aumentarne l'efficienza e la stabilità.

4.2 *Bounding Volume Hierarchy*

4.2.1 Introduzione

Una *Bounding Volume Hierarchy* (BVH) è una struttura ad albero su un insieme di oggetti geometrici. Tutti gli oggetti geometrici sono raccolti in volumi limite che formano i nodi fogliari dell'albero. Questi nodi vengono quindi raggruppati come piccoli insiemi e racchiusi in volumi di delimitazione più grandi. Questi, a loro volta, sono ancora raggruppati e racchiusi in altri volumi di delimitazione più grandi in modo ricorsivo, risultando infine in una struttura ad albero con un singolo volume di delimitazione nella parte superiore dell'albero. Le gerarchie di volumi limitanti vengono utilizzate per supportare in modo efficiente diverse operazioni su insiemi di oggetti geometrici, come ad esempio il rilevamento delle collisioni.

Sebbene il *wrapping* degli oggetti nei volumi di delimitazione e l'esecuzione di test di collisione su di essi prima del test della geometria dell'oggetto stesso semplifichino i test e possano comportare miglioramenti significativi delle prestazioni, è ancora in corso lo stesso numero di test a coppie tra volumi di delimitazione. Organizzando i volumi di delimitazione in una gerarchia di volumi di delimitazione, la complessità temporale (il numero di test eseguiti) può essere ridotta logaritmicamente nel numero di oggetti. Con una tale gerarchia in atto, durante i test di collisione, i volumi secondari non devono essere esaminati se i loro volumi principali non sono intersecati.

4.2.2 *Minimum Bounding Box*

In geometria, il rettangolo minimo o più piccolo (o *Minimum Bounding Box* (MBB)) per racchiudere un insieme di punti S in N dimensioni è l'rettangolo con la misura più piccola (area, volume o ipervolume in dimensioni superiori) all'interno del quale si trovano tutti i punti. Il termine "iper-rettangolo (o più semplicemente *box*) deriva dal suo utilizzo nel sistema di coordinate cartesiane, dove viene effettivamente

visualizzato come un rettangolo (caso bidimensionale), parallelepipedo rettangolare (caso tridimensionale), ecc. Nel caso bidimensionale viene chiamato rettangolo di delimitazione minimo.

4.2.2.1 *Axis Aligned Bounding Box*

Il MBB allineato agli'assi (*Axis Aligned Bounding Box* (AABB)) per un determinato set di punti è il rettangolo di delimitazione minimo soggetto al vincolo che i bordi del rettangolo sono paralleli agli assi cartesiani. È il prodotto cartesiano di N intervalli ciascuno dei quali è definito da un valore minimo e un valore massimo della coordinata corrispondente per i punti in S.

I rettangoli di delimitazione minimi allineati all'asse vengono utilizzati per determinare la posizione approssimativa di un oggetto e come descrittore molto semplice della sua forma. Ad esempio, nella geometria computazionale e nelle sue applicazioni quando è necessario trovare intersezioni nel set di oggetti, il controllo iniziale sono le intersezioni tra i loro MBB. Dato che di solito è un'operazione molto meno costosa del controllo dell'intersezione effettiva (perché richiede solo confronti di coordinate), consente di escludere rapidamente i controlli delle coppie che sono molto distanti.

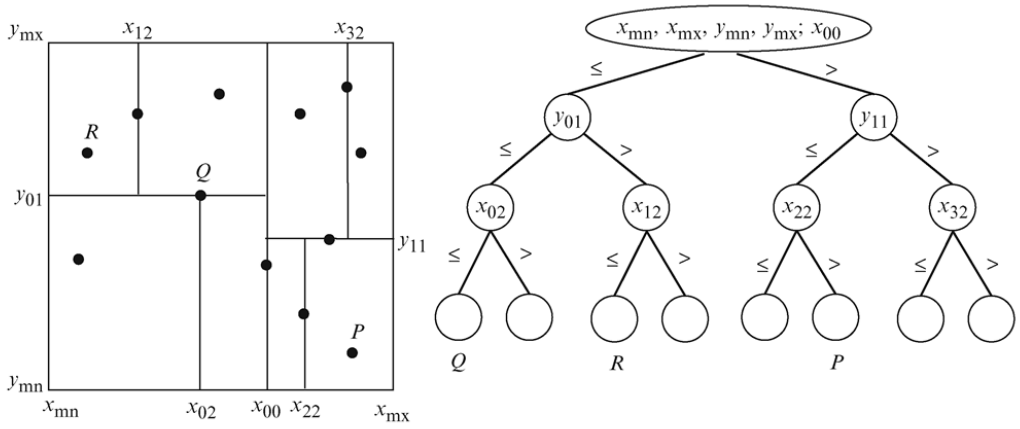


FIGURA 4.1: Esempio di albero di tipo AABB.

4.2.2.2 *Arbitrarily Oriented Bounding Box*

Il MBB orientato arbitrariamente (*Arbitrarily Oriented Bounding Box* (AOBB)) è il rettangolo di delimitazione minimo, calcolato senza vincoli per quanto riguarda

l'orientamento del risultato. Gli algoritmi del rettangolo di delimitazione minimo basati sul metodo dei calibri rotanti possono essere utilizzati per trovare l'area di delimitazione dell'area minima o del perimetro minimo di un poligono convesso bidimensionale in tempo lineare e di un punto bidimensionale impostato nel tempo impiegato costruire il suo scafo convesso seguito da un calcolo del tempo lineare. Un algoritmo di pinze rotanti tridimensionali può trovare il rettangolo di delimitazione orientato arbitrariamente sul volume minimo di un punto tridimensionale impostato in tempo cubo.

4.2.2.3 *Object Oriented Bounding Box*

Nel caso in cui un oggetto abbia un proprio sistema di coordinate locale, può essere utile memorizzare un rettangolo di selezione relativo a questi assi, che non richiede alcuna trasformazione quando cambia l'orientazione dell'oggetto stesso.

4.2.3 Intersezione tra Alberi AABB

Per il rilevamento delle collisioni tra oggetti in due dimensioni, l'intersezione tra alberi di tipo AABB, è l'algoritmo più veloce per determinare se le due entità di gioco si sovrappongono o meno, e in che parti. Nello specifico, ciò consiste nel controllare le posizioni delle *i*-esime *Bounding Box* (BB) nello spazio delle coordinate bidimensionali per vedere se si sovrappongono.

Il vincolo di allineamento dei rettangoli agli assi è presente per motivi di prestazioni, infatti, l'area di sovrapposizione tra due riquadri non ruotati può essere controllata solo con confronti logici. Mentre i riquadri ruotati richiedono ulteriori operazioni trigonometriche, che sono più lente da calcolare. Inoltre, se si hanno entità che possono ruotare, le dimensioni dei rettangoli e/o sotto-rettangoli dovranno modificarsi in modo da avvolgere ancora l'oggetto o si dovrà optare per un altro tipo di geometria di delimitazione, come le sfere (che sono invarianti alla rotazione).

Nel caso specifico, l'ombra dello pneumatico sarà rappresentata da un albero di tipo AABB con una sola foglia. Ovvero si andrà a rappresentare lo pneumatico con una BB avente lati uguali e rappresentanti il massimo ingombro che può avere nello spazio. Si andrà inoltre ad incrementare del 10% ognuno di questi lati in modo da tenere conto dell'angolo di camber, che portrebbe portare i punti di campionamento del terreno fuori dall'ombra. La strada, contrariamente al pneumatico, verrà tenuta come riferimento assoluto. In altre parole, una volta effettuato la parsificazione

del file RDF, verrà calcolato l'albero di tipo AABB. Lo pneumatico si muoverà all'interno della *mesh* e la sua ombra verrà ricalcolata e intersecata con l'albero AABB per ottenere tutti i triangoli in corrispondenza della stessa.

Volendo intersecare due semplici BB, quali $A = [\texttt{A.minX}, \texttt{A.maxX}; \texttt{A.minY}, \texttt{A.maxY}]$ e $B = [\texttt{B.minX}, \texttt{B.maxX}; \texttt{B.minY}, \texttt{B.maxY}]$, verrà usata la seguente funzione.

```
1 function intersect(A,B) {
2   return (A.minX <= B.maxX && A.maxX >= B.minX) &&
3          (A.minY <= B.maxY && A.maxY >= B.minY)
4 }
```

Volendo intersecare un albero di tipo AABB e una semplice BB, basterà ripetere a più step la funzione precedente lungo i rami dell'albero. Una volta arrivati a una o più foglia avremo tutti gli oggetti (o triangoli nel caso specifico) che sono posti in corrispondenza della BB (od ombra dello pneumatico nel caso specifico). Questi triangoli verranno poi usati per determinare il piano strada locale e il punto di contatto virtuale dello pneumatico.

È imporatante notare che il metodo appena visto, presenta numerosi vantaggi.

- Riduzione del numero di comparazioni da effettuare per ottenere l'intersezione BB-albero AABB. Infatti, la *mesh* può contenere decine di migliaia di trangoli, il metodo presentato consente di ridurre logarirmicamente il numero di comparazioni necessarie per ottenere il risultato.
- Riduzione del numero di trangoli da processare per ottenere il piano strada locale e il punto di contatto virtuale dello pneumatico. Infatti, vengono solamente processati quelli posti in corrispondenza del'ombra dello pneumatico.

4.3 Algoritmi Geometrici

4.3.1 Introduzione

La geometria computazionale è la branca dell'informatica che studia le strutture dati e gli algoritmi efficienti per la soluzione di problemi di natura geometrica e la loro implementazione al calcolatore. Storicamente, è considerato uno dei campi più antichi del calcolo, anche se la geometria computazionale moderna è uno sviluppo recente. La ragione principale per lo sviluppo della geometria computazionale è

stata dovuta ai progressi compiuti nella computer grafica, *Computer-Aided Design* (CAD), *Computer-Aided Manufacturing* (CAM) e nella visualizzazione matematica. Ad oggi, le applicazioni della geometria computazionale si trovano nella robotica, nella progettazione di circuiti integrati, nella visione artificiale, in *Computer-Aided Engineering* (CAE) e nel *Geographic Information Systems* (GIS). I rami principali della geometria computazionale sono:

- *Calcolo combinatorio* (o *geometria algoritmica*), che si occupa di oggetti geometrici come entità discrete. Ad esempio, può essere utilizzato per determinare il poliedro o il poligono più piccolo che contiene tutti i punti forniti, o più formalmente, dato un insieme di punti, si deve determinare il più piccolo insieme convesso che li contenga tutti (problema dell'inviluppo convesso).
- *Geometria di calcolo* numerica (o *Computer-Aided Geometric Design* (CAGD)), che si occupa principalmente di rappresentare oggetti del mondo reale in forme adatte per i calcoli informatici nei sistemi CAD e CAM. Questo ramo può essere visto come uno sviluppo della geometria descrittiva ed è spesso considerato un ramo della computer grafica o del CAD. Entità importanti di questo ramo sono superfici e curve parametriche, come ad esempio le *spline* e *curve di Bézier*.

In questo capitolo tutti gli algoritmi che verranno utilizzati in seguito durante l'analisi geometrica dell'intersezione tra pneumatico e superficie stradale saranno trattati. Questi algoritmi sono la soluzione di alcuni semplici ma molto importanti problemi, che devono essere risolti in modo efficiente. In particolare le intersezioni tra:

- punto e segmento (sul piano);
- punto e circonferenza (sul piano);
- raggio e circonferenza (sul piano);
- raggio e triangolo (sullo spazio);

saranno esaminati al fine di trovare la massima prestazione in termini di efficienza computazionale.

4.3.2 Intersezione tra Entità Geometriche

4.3.2.1 Punto-Segmento

Dato un punto $P = (x_p, y_p)$ e un segmento definito da due punti $A = (x_A, y_B)$ e $B = (x_B, y_B)$.

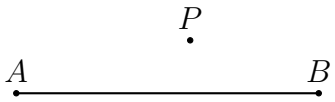


FIGURA 4.2: Schema grafico per l'intersezione punto-segmento

Per determinare se il punto P è intermo al segmento si eseguiranno i seguenti step.

1. Creazione di un vettore $\overrightarrow{AB}$ e di un vettore $\overrightarrow{AP}$.
2. Calcolo il prodotto vettoriale $\overrightarrow{P_1P_2} \times \overrightarrow{PP_1}$, se il modulo del vettore risultante è nullo allora il punto P appartiene al segmento considerato.
3. Calcolo il prodotto scalare tra $\overrightarrow{AB}$ e $\overrightarrow{AP}$. Se è nullo allora il punto P è coincidente a A, se è pari al modulo di $\overrightarrow{AB}$ allora il punto P è coincidente a B, se è compreso tra 0 il modulo di $\overrightarrow{AB}$, allora il punto P giace all'interno del segmento considerato.

Il codice che esegue questo tipo di test è riportato in Figura 4.4

(a) *Output* di tipo `integer` (b) *Output* di tipo `bool`

FIGURA 4.3: Schemi per l'*output* dell'intersezione punto-segmento.

4.3.2.2 Punto-Cerchio

Data una circonferenza con centro $C = (x_c, y_c)$ e raggio r, il problema consiste nel trovare se un punto generico $P = (x_p, y_p)$ è locato all'interno, all'esterno o sulla circonferenza. La soluzione al problema è semplice: la distanza tra il centro del cerchio C e il punto P è data dal teorema di Pitagora. In particolare:

$$d = \sqrt{(x_p - x_c)^2 + (y_p - y_c)^2} \tag{4.1}$$

Output di tipo `integer`

```
1  if ( AB.cross(AP) > epsilon )
2    { return 0; }
3  KAP = AB.dot(AP);
4  if ( KAP < -epsilon )
5    { return 0; }
6  if ( abs(KAP) < epsilon  )
7    { return 1; }
8  KAB = AB.dot(AB);
9  if ( KAP > KAB )
10   { return 0; }
11 if ( abs(KAP-KAB) < epsilon )
12   { return 2; }
13 return 3;
```

Output di tipo `bool`

```
1  if ( AB.cross(AP) > epsilon )
2    { return false; }
3  KAP = AB.dot(AP);
4  if ( KAP < -epsilon )
5    { return false; };
6  if ( abs(KAP) < epsilon )
7    { return true; }
8  KAB = AB.dot(AB);
9  if ( KAP > KAB )
10   { return false; }
11 if ( abs(KAP-KAB) < epsilon )
12   { return true; }
13 return true;
```

Figura 4.4: Schema del codice per l'intersezione punto-segmento.

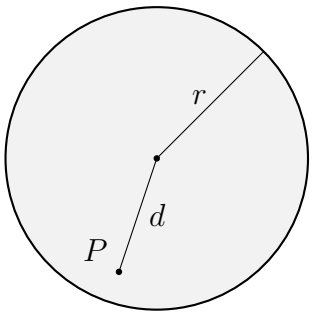


Figura 4.5: Schema del problema di intersezione punto-cerchio.

il punto P è dunque interno alla circonferenza se $d < r$, appartiene alla circonferenza se $d = r$ ed esterno alla circonferenza se $d > r$. In maniera analoga ma più efficace da punto di vista computazionale si può confrontare d^2 con r^2. Il punto P è dunque interno alla circonferenza se $d^2 < r^2$, appartiene alla circonferenza se $d^2 = r^2$ ed esterno alla circonferenza se $d^2 > r^2$. Pertanto, il confronto finale sarà tra il numero $(x_p - x_c)^2 + (y_p - y_c)^2$ e r^2.
Gli *inputs* dell'algoritmo per l'intersezione punto-cerchio sono:

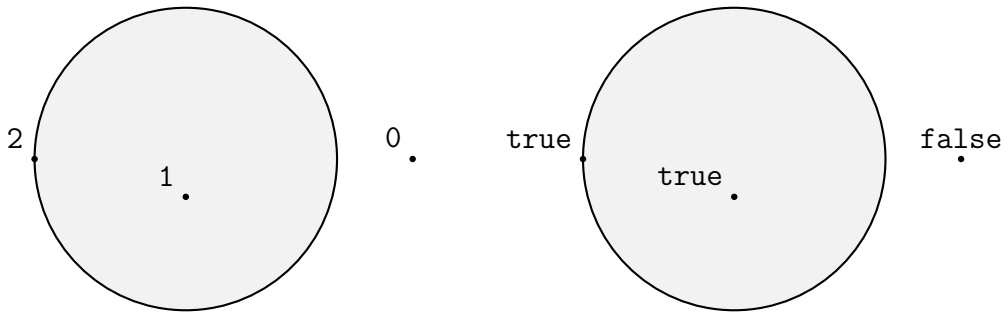
- il centro della circonferenza $C = (x_c, y_c)$;
- il raggio della circonferenza r;
- il punto generico da analizzare $P = (x_p, y_p)$.

L'*output* può essere un intero il cui valore può essere:

- `0` se il punto è esterno;
- `1` se il punto è interno;
- `2` se il punto appartiene alla circonferenza.

Il valore in *output* può essere anche una variabile booleana il cui valore è:

- `false` se il punto è esterno;
- `true` se il punto è interno o appartiene alla circonferenza.



(a) *Output* di tipo `integer` (b) *Output* di tipo `bool`

FIGURA 4.6: Schemi per l'*output* dell'intersezione punto-cerchio.

Output di tipo `integer`

```
1  d = (x_p-x_c)^2 + (y_p-y_c)^2;
2  if ( d > r^2 ){ return  0; }
3  else if ( d < r^2 ){ return 1; }
4  else { return 2; }
```

Output di tipo `bool`

```
1  d = (x_p-x_c)^2 + (y_p-y_c)^2;
2  if ( d > r^2 ){ return  true; }
3  else { return false; }
```

FIGURA 4.7: Schemi del codice per l'intersezione punto-cerchio.

4.3.2.3 Piano-Piano

Nello spazio delle coordinate tridimensionali, due piani P_1 e P_2 o sono paralleli o si intersecano creando una singola retta L. Sia P_i con $i = 1, 2$ descritto da un punto V_i e un vettore normale $\vec{n}_i$. L'equazione implicita del piano sarà dunque:

$$\vec{n}_i \cdot P + d_i = 0 \tag{4.2}$$

dove $P = (x, y, z)$. I piani P_1 e P_2 sono paralleli ogni volta che i loro normali vettori $\vec{n}_1$ e $\vec{n}_2$ sono paralleli. Questo equivale alla condizione che $\vec{n}_1 \times \vec{n}_2 = 0$.

Quando i piani non sono paralleli, $\vec{u} = \vec{n}_1 \times \vec{n}_2$ è il vettore di direzione della linea di intersezione L. Si noti che $\vec{u}$ è perpendicolare sia a $\vec{n}_1$ che a $\vec{n}_2$, e quindi è parallelo a entrambi i piani.

Dopo aver calcolato $\vec{n}_1 \times \vec{n}_2$, per determinare univocamente la linea di intersezione, è necessario trovare un punto di essa. Cioè, un punto $P_0 = (x_0, y_0, z_0)$ che si trova in entrambi i piani. Si può trovare una soluzione comune delle equazioni implicite per P_1 e P_2. Ma ci sono solo due equazioni nelle 3 incognite poiché il punto P_0 può trovarsi ovunque sulla linea monodimensionale L. Quindi è necessario aggiungere un altro vincolo da risolvere per un P_0 specifico. Esistono diversi modi per farlo, il più semplice è attraverso l'aggiunta dei un terzo piano P_3 avente equazione implicita $\vec{n}_3 \cdot P = 0$ dove $\vec{n}_3 = \vec{n}_1 \times \vec{n}_2$ e $d_3 = 0$ (ovvero passa attraverso l'origine). Questo metodo è funzionante poiché:

- L è perpendicolare a P_3 e quindi lo interseca;
- i vettori $\vec{n}_1$, $\vec{n}_2$ e $\vec{n}_3$ sono linearmente indipendenti.

Pertanto i piani P_1, P_2 e P_3 si intersecano in un unico punto P_0 che deve trovarsi su L.

Nello specifico, la formula per l'intersezione di 3 piani è:

$$P_0 = \frac{-d_1(\vec{n}_2 \times \vec{n}_3) - d_2(\vec{n}_3 \times \vec{n}_1) - d_3(\vec{n}_1 \times \vec{n}_2)}{\vec{n}_1 \cdot (\vec{n}_2 \times \vec{n}_3)} \tag{4.3}$$

e ponendo $d_3 = 0$ per P_3, si otteniene:

$$P_0 = \frac{-d_1(\vec{n}_2 \times \vec{n}_3) - d_2(\vec{n}_3 \times \vec{n}_1)}{\vec{n}_1 \cdot (\vec{n}_2 \times \vec{n}_3)} = \frac{(d_2\vec{n}_1 - d_1\vec{n}_2) \times \vec{n}_3)}{(\vec{n}_1 \times \vec{n}_2) \cdot \vec{n}_3} = \frac{(d_2\vec{n}_1 - d_1\vec{n}_2) \times \vec{u})}{|\vec{u}|^2} \tag{4.4}$$

e l'equazione parametrica per la retta L sarà:

$$L(s) = \frac{(d_2\vec{n}_1 - d_1\vec{n}_2) \times \vec{u})}{|\vec{u}|^2} + s\vec{u} \tag{4.5}$$

dove $\vec{u} = \vec{n}_1 \times \vec{n}_2$.

4.3.2.4 Piano-Segmento

4.3.2.5 Piano-Triangolo

4.3.2.6 Raggio-Triangolo

Dato un triangolo avente vertici (A, B, C) e un raggio R con origine R_O e direzione R_D, il problema consiste nel capire se il raggio colpisce o meno il triangolo e, in tal

caso, trovare il punto di intersezione P. Negli ultimi decenni, sono stati proposti

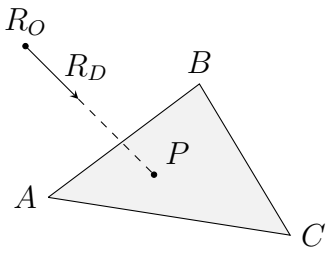


FIGURA 4.8: Rappresentazione del problema di intersezione raggio-triangolo.

numerosi algoritmi per risolvere questo problema, esistono quindi diverse soluzioni al problema di intersezione raggio-triangolo. Tre degli algoritmi più importanti sono:

- l'agoritmo di *Badouel*;
- l'agoritmo di *Segura*;
- l'agoritmo di *Möller* e *Trumbore*.

Come Jiménez, Segura e Feito afferma in [2], l'algoritmo di Möller-Trumbore's è il più veloce quando il piano normale e/o il piano di proiezione non sono stati precedentemente memorizzati, come nel caso specifico di questa tesi.

La teoria alla base di questo algoritmo è spiegata estensivamente in [6]. In particolare, l'algoritmo sfrutta la parametrizzazione di P, il punto di intersezione, in termini delle coordinate baricentriche, ovvero:

$$P = wA + uB + vC \tag{4.6}$$

Dato che $w = 1 - u - v$, si può quindi scrivere:

$$P = (1 - u - v)A + uB + vC \tag{4.7}$$

e sviluppando si ottiene:

$$P = A - uA - vA + uB + vC = A + u(B - A) + v(C - A) \tag{4.8}$$

Si noti che $(B-A)$ e $(C-A)$ sono i bordi AB e AC del triangolo ABC. L'intersezione P può anche essere scritta usando l'equazione parametrica del raggio:

$$P = R_O + tR_D \tag{4.9}$$

dove t è la distanza dall'origine del raggio all'intersezione P. Sostituendo P nell'equazione 4.8 con l'equazione del raggio si ottiene:

$$R_O + tR_D = A + u(B - A) + v(C - A) O - A = \quad -tD + u(B - A) + v(C - A) \tag{4.10}$$

Sul membro a sinistra si hanno le tre incognite (t, u, v) moltiplicate per tre termini noti $(B-A, C-A, D)$. Si può riorganizzare questi termini e presentare l'equazione 4.10 usando la seguente notazione:

$$\begin{bmatrix} -D & (B - A) & (C - A) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = R_O - A \tag{4.10}$$

Si immagini ora di avere un punto P all'interno del triangolo. Se si trasforma il triangolo in qualche modo (ad esempio traslandolo, ruotandolo o scalandolo), le coordinate del punto P espresse nel sistema di coordinate cartesiane tridimensionali (x, y, z) cambieranno. D'altra parte, se si esprime la posizione di P usando le coordinate baricentriche, le trasformazioni applicate al triangolo non influenzeranno le coordinate baricentriche del punto di intersezione. Se il triangolo viene ruotato, ridimensionato, allungato o traslato, le coordinate (u, v) che definiscono la posizione di P rispetto ai vertici (A, B, C) non cambieranno. L'algoritmo di Möller-Trumbore sfrutta proprio questa proprietà. Infatti, ciò che gli autori hanno fatto è definire un nuovo sistema di coordinate in cui le coordinate di P non sono definite in termini di (x, y, z) ma in termini di (u, v). La somma tra le coordinate baricentriche non può essere maggiore di 1 ($u + v \leq 1$), esprimono infatti le coordinate dei punti definiti all'interno di un triangolo unitario. Ovvero un triangolo definito nello spazio (u, v) dai vertici $(0, 0)$, $(1, 0)$, $(0, 1)$.

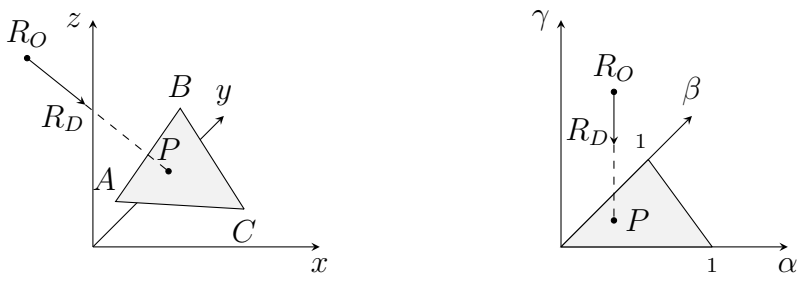


Figura 4.9: Transformation and base change of ray in Möller-Trumbore algorithm.

Geometricamente, si è appena chiarito il significato di u e v. Si consideri ora l'elemento t. Esso è il terzo asse del sistema di coordinate u e v appena introdotto. Si sà inoltre che t esprime la distanza dall'origine del raggio a P, il punto di intersezione, si è quindi creato un sistema di coordinate che consentirà di esprimere univocamente la posizione del punto d'intersezione P in termini di coordinate baricentriche e distanza dall'origine del raggio a quel punto sul triangolo.

Möller e Trumbore spiegano che la prima parte dell'equazione 4.10 (il termine $O - A$) può essere vista come una trasformazione che sposta il triangolo dalla sua posizione spaziale mondiale originale all'origine (il primo vertice del triangolo coincide con l'origine). L'altro lato dell'equazione ha l'effetto di trasformare il punto di intersezione dallo spazio (x, y, z) nello spazio (t, u, v) come spiegato precedentemente.

Per risolvere l'equazione 4.10, Möller e Trumbore hanno usato una tecnica conosciuta in matematica come regola di Cramer. La regola di Cramer fornisce la soluzione a un sistema di equazioni lineari mediante il determinante. La regola afferma che se la moltiplicazione di una matrice M per un vettore colonna X è uguale a un vettore colonna C, allora è possibile trovare X_i (l'i-esimo elemento del vettore colonna X) dividendo il determinante di M_i per il determinante di M. Dove M_i è la matrice formata sostituendo la sua colonna di M con il vettore colonna C. Usando questa regola si ottiene;

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\begin{vmatrix} -D & E_1 & E_2 \end{vmatrix}} \begin{bmatrix} \begin{vmatrix} T & E_1 & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & T & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & E_1 & T \end{vmatrix} \end{bmatrix} \tag{4.11}$$

dove $T = O - A$, $E_1 = B - A$ ed $E_2 = C - A$. Il prossimo passo è trovare un valore per questi quattro determinanti. Il determinante (di una matrice 3×3) non è altro che un triplo prodotto scalare, quindi si può riscrivere l'equazione precedente come:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix} \tag{4.12}$$

dove $P = (D \times E_2)$ e $Q = (T \times E_1)$. Come si può vedere ora è facile trovare i valori t, u e v.

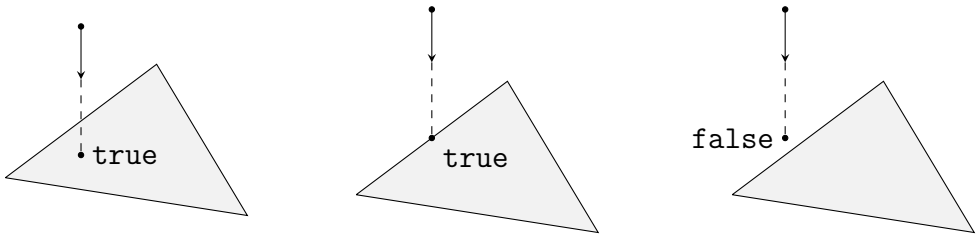


Figura 4.10: Schemi per l'*output* dell'intersezione punto-cerchio.

```
1  E_1 = B - A;
2  E_2 = C - A;
3  A = R_D × E_2;
4  D = A · E_1;
5  if ( D > epsilon ) {
6    T = R_0 - A;
7    u = A · T;
8    if ( u < 0.0 || u > D ) return false;
9    B = T × E_1;
10   v = B · R_D;
11   if (v < 0.0 || u + v > D) return false;
12 } else if ( D < -epsilon ) {
13   T = R_0 - A;
14   u = A · T;
15   if (u > 0.0 || u < D) return false;
16   B = T × E_1;
17   v = B · R_D;
18   if ( v > 0.0 || u + v < D ) return false;
19 } else {
20   return false;
21 }
22 t = ( B · E_2 ) / D;
23 if ( t > 0.0 ) {
24   P = Q + D * t;
25   return true;
26 } else {
27   return false;
28 }
```

Figura 4.11: Schema per del codice per l'intersezione raggio-triangolo con *back-face culling*.

La Libreria TireGround 5

5.1 Organizzazione

La libreria TireGround è stata organizzata in tre parti, definite dagli stessi *namespaces*. In seguito verranno riportate le informazioni di maggior rilievo per ognuna delle tre parti della libreria.

5.1.1 *Namespace* TireGround

In questo *namespace*, vengono raccolti i tipi dichiarati con typedef, comuni ai *namespaces* RDF e PatchTire

5.1.2 *Namespace* RDF

In questo *namespace* vengono raccolti alcuni tipi dichiarati con typedef presenti solo nel namespace RDF. Lo spazio dei nomi RDF contiene tutti le classi e la funzioni per gestire la *mesh* a partire dal file in formato RDF.

BBox2D Questa classe contiene tutte le informazioni per definire e manipolare una BB bidimensionale. Consiste nella descrizione geometrica dell'oggetto BB. I metodi più importanti di questa classe sono i seguenti.

- clear – Elimina il dominio della BB settando tutti i quattro valori su quietNaN.

- `updateBBox2D` – Aggiorna il dominio della BB settando i suoi valori secondo il massimo ingombro dato dai tre vertici nello spazio tridimensionale in *input*.

Tipo	Nome	*Getter*	*Setter*	Descrizione
`real_type`	`Xmin`	•	•	X_{min} della BB
`real_type`	`Ymin`	•	•	Y_{min} della BB
`real_type`	`Xmax`	•	•	X_{max} della BB
`real_type`	`Ymax`	•	•	Y_{max} della BB

Tabella 5.1: Attributi della classe `BBox2D`.

`Triangle3D` Questa classe contiene tutte le informazioni geometriche per definire e manipolare un triangolo con vertici nello spazio tridimensionale. Consiste nella descrizione geometrica dell'oggetto triangolo. I metodi più importanti di questa classe sono i seguenti.

- `Normal` – Calcola la normale alla faccia del triangolo.
- `intersectRay` – Interseca il triangolo con una data semiretta (detta anche raggio), definita da direzione e punto di partenza, e ne calcola il punto di intersezione.
- `intersectPlane` – Interseca il triangolo con un dato piano, definito da normale e punto noto, e ne calcola i punti di intersezione.

Tipo	Nome	*Getter*	*Setter*	Descrizione
`vec3`	`Vertices[3]`	•	•	Vertici del triangolo
`BBox2D`	`TriangleBBox`	•	•	BB del triangolo

Tabella 5.2: Attributi della classe `Triangle3D`.

`TriangleRoad` Questa classe contiene tutte le informazioni geometriche e non geometriche per definire e manipolare un triangolo con vertici nello spazio tridimensionale rappresentante la superficie stradale. È derivato dalla classe `Triangle3D` e ha inoltre un attributo che permetter di descrivere il coefficiente di attrito nella faccia (detto anche locale). I metodi più importanti sono ereditati dalla classe `Triangle3D`.

Tipo	Nome	*Getter*	*Setter*	Descrizione
`real_type`	`Friction`	•	•	Coefficiente di attrito μ

Tabella 5.3: Attributi della classe `TriangleRoad`.

`MeshSurface` Questa classe contiene il vettore di puntatori di tipo `std::shared_ptr` alle istanze della classe `TriangleRoad` che vengono create durante la parsificazione del file RDF. Inoltre contiene il vettore di puntatori alle BB di tipo `PtrBBox`, che è necessario per calcolare l'albero AABB. Quest'ultimo esiste come ulteriore attributo della classe sotto forma di puntatore `PtrAABB`. I metodi più importanti di questa classe sono i seguenti.

- `set` – Copia la mesh.
- `LoadFile` – Parsifica il file dato come *input* e crea le istanze `TriangleRoad` che costituiscono la *mesh*.
- `updateIntersection` – Interseca l'albero di tipo AABB della *mesh* con un altro albero esterno di tipo AABB e ne restituisce il vettore dei puntatori di tipo `std::shared_ptr` alle istanze della classe `TriangleRoad` che vengono intersecate.

Tipo	Nome	*Getter*	*Setter*	Descrizione
`TriangleRoad_list`	`Friction`	•		Vettore dei triangoli
`std::vector<PtrBBox>`	`PtrBBoxVec`	•		Vettore delle BB
`PtrAABB`	`PtrTree`	•		Albero di tipo AABB

Tabella 5.4: Attributi della classe `MeshSurface`.

5.1.3 *Namespace* `PatchTire`

In questo *namespace* vengono raccolti alcuni tipi dichiarati con `typedef` presenti solo nel namespace `PatchTire`. Lo spazio dei nomi `PatchTire` contiene inoltre tutti le classi e la funzioni per gestire l'intersezione tra lo pneumatico e la *mesh* a partire dalla conoscenza di quest'ultima, della geometria e della posizione dello pneumatico.

`Disk` Questa classe contiene tutte le informazioni geometriche per definire e manipolare un disco nello spazio tridimensionale. Consiste nella descrizione geometrica e nel posizionamento dello spazio delle coordinate tridimensionali dell'oggetto

disco (il disco viene rappresentato nel sistema di riferimento dello pneumatico). I metodi più importanti di questa classe sono i seguenti.

- `isPointInside` – Controlla se un punto generico nello spazio bidimensionale, definito dal piano in cui giace lo stesso disco, si trova all'interno o all'esterno della circonferenza.
- `intersectSegment` – Trova i punti di intersezione tra la circonferenza esterna del disco e un segmento bidimensionale, che dev'essere definito nel piano in cui giace lo stesso disco. L'intero di *output* fornisce il numero di punti di intersezione.
- `intersectPlane` – Interseca il disco con un piano definito da normale e punto noto. In *output* fornisce l'entità geometrica creata dall'intersezione sotto forma di punto noto e direzione della retta.
- `getPatchLength` – Funzione in *overloading* che consente, attraverso vari tipolgie in *input* di trovare la lunghezza del tratto interno al disco e che può essere creato da un piano, da dei triangoli, da un segmento bidimensionale o da una spezzata bidimensionale.

Tipo	Nome	*Getter*	*Setter*	Descrizione
`vec2`	`OriginXZ`	•	•	Coordinate XZ del disco
`real_type`	`OffsetY`	•	•	Coordinata Y del disco
`real_type`	`Radius`	•	•	Circonferenza del disco

Tabella 5.5: Attributi della classe `Disk`.

`ETRTO` Questa classe contiene tutte le informazioni necessarie per definire geometricamente uno pneumatico secondo la normativa ETRTO. Consiste nella descrizione geometrica dell'oggetto pneumatico in termini di larghezza totale e di diametro esterno indeformato. Come visto nel Capitolo 2 attraverso la nomenclatura ETRTO (e.g. 205/65R16) è infatti possibile risalire a tutte le informazioni geometriche che definiscono, anche se in maniera grossolana, lo pneumatico.

`ReferenceFrame` Questa classe contiene tutte le informazioni per definire e manipolare una terna di riferimento nello spazio tridimensionale. Consiste nel posizionamento dello spazio del sistema di riferimento. I metodi più importanti di questa classe sono i seguenti.

Tipo	Nome	*Getter*	*Setter*	Descrizione
`real_type`	`SectionWidth`	•	•	Larghezza dello pneumatico
`real_type`	`AspectRatio`	•	•	Rapporto percentuale H/W
`real_type`	`RimDiameter`	•	•	Diametro del cerchione
`real_type`	`SidewallHeight`	•		Altezza della spalla
`real_type`	`TireDiameter`	•		Diametro dello pneumatico

Tabella 5.6: Attributi della classe `BBox2D`.

- `setTotalTransformationMatrix` – Posiziona nello spazio il sistema di riferimento grazie alla matrice di trasformazione 4×4 fornita come *input*.
- `getEulerAngleX` – Ottiene l'angolo creato dalla rotazione attorno all'asse Y del sistema di riferimento locale rispetto a quello assoluto (lo stesso della *mesh*). L'angolo viene ottenuto in seguito alla fattorizzazione $R_z(\Omega)R_x(\gamma)R_y(\theta)$ e utilizzando il metodo di Eulero.
- `getEulerAngleY` – Come il metodo `getEulerAngleX`, ma usato per il ottenere l'angolo creato dalla rotazione attorno all'asse Y.
- `getEulerAngleZ` – Come il metodo `getEulerAngleX`, ma usato per il ottenere l'angolo creato dalla rotazione attorno all'asse Z.

Tipo	Nome	*Getter*	*Setter*	Descrizione
`vec3`	`Origin`	•	•	Origine della terna
`mat3`	`RotationMatrix`	•	•	Matrice di rotazione

Tabella 5.7: Attributi della classe `ReferenceFrame`.

`Shadow` Questa classe serve a rappresentare l'ombra dello pneumatico nello spazio bidimensionale. È molto simile alla `RDF::BBox2D` precedentemente presentata, ma a differenza di quest'ultima permette di calcolare anche l'albero per oggetti di tipo AABB a una sola foglia, relativo alla stessa ombra dello pneumatico. I metodi più importanti di questa classe sono i seguenti.

- `clear` – Elimina il dominio dell'ombra settando tutti i suoi valori su `quietNaN`.
- `update` – Aggiorna il dominio dell'ombra settando tutti i suoi valori secondo il massimo ingombro dato dalla geometria dello pneumatico e dalla sua posizione nello spazio.

Tipo	Nome	*Getter*	*Setter*	Descrizione
`real_type`	`Xmin`	•	•	X_{min} dell'ombra
`real_type`	`Ymin`	•	•	Y_{min} dell'ombra
`real_type`	`Xmax`	•	•	X_{max} dell'ombra
`real_type`	`Ymax`	•	•	Y_{max} dell'ombra
`std::vector<PtrBBox>`	`PtrBBoxVec`	•		BB dell'ombra
`PtrAABB`	`PtrTree`	•		Albero di tipo AABB

Tabella 5.8: Attributi della classe `Shadow`.

`Tire` Questa classe serve a rappresentare lo pneumatico nelle coordinate dello spazio tridimensionale. Consiste nel punto di giunzione tra la classe `ETRTO` che definisce la geometria dello pneumatico in condizione di riposo e la classe `ReferenceFrame` che ne definisce invece la posizione nello spazio.

Tipo	Nome	*Getter*	*Setter*	Descrizione
`ETRTO`	`TireGeometry`			Geometria
`ReferenceFrame`	`RF`	•	•	Posizione

Tabella 5.9: Attributi della classe `Tire`.

`MagicFormula` Questa classe calcola tutti i parametri necessari per valutare il contatto tra pneumatico e terreno attraverso la *Magic Formula*. I metodi più importanti di questa classe sono i seguenti.

- `setup` – Consente di riposizionare la ruota all'interno della *mesh*.
- `pointSampling` – Interseca i triangoli in corrispondenza dell'ombra dello pneumatico con un raggio, definito da direzione e punto di partenza, e ne calcola il punto di intersezione.
- `fourPointsSampling` – Effettua l'intersezione tra i triangoli in corrispondenza dell'ombra dello pneumatico e quattro reggi, il cui punto di partenza e direzione sono prestabiliti della geometria e posizione nello spazio. Con i quattro punti di intersezione e la normale è possibile stabilire il punto di contatto virtuale.
- `calculateLocalRoadPlane` – Calcola la normale del piano strada locale.
- `calculateRelativeCamber` – Calcola il camber relativo.
- `getContactDepth` – Calcola l'affondamento del disco nel piano strada locale.

- `getContactArea` – Funzione in *overloading* che calcola l'area dell'impronta contatto dello pneumatico (considerato come un cilindro) in via approssimata.
- `getContactVolume` – Calcola il volume di intersezione approssimato tra terreno e pneumatico (considerato come un cilindro)

Tipo	Nome	*Getter*	*Setter*	Descrizione
`Disk`	`SingleDisk`			Disco rigido
`vec3`	`ContactNormal`	•		Normale del piano strada
`vec3`	`ContactPoint`	•		Punto di contatto virtuale
`real_type`	`ContactFriction`	•		Coefficiente di attrito locale
`real_type`	`RelativeCamber`	•		Camber relativo

Tabella 5.10: Attributi della classe `MagicFormula`.

5.2 Librerie Esterne

Oltre al codice appena descritto sono state utilizzate anche altre due librerie esterne al fine di velocizzare il processo di sviluppo e al contempo di utilizzare una solida base per le operazione più complesse, ovvero le operazioni matriciali e vettoriali, nonché la creazione degli alberi per oggetti di tipo AABB e l'intersezione tra gli stessi.

5.2.1 `Eigen3`

`Eigen3` è una libreria `C++` di alto livello di *template headers* per operazioni di algebra lineare, vettoriali, matriciali, trasformazioni geometriche, *solver* numerici e algoritmi correlati.

Questa libreria è implementata usando la tecnica di *template metaprogramming*, che crea degli alberi di espressioni in fase di compilazione e genera un codice personalizzato per valutarli. Utilizzando i modelli di espressione e un modello di costo delle operazioni in virgola mobile, la libreria esegue il proprio srotolamento del loop e vettorializzazione.

5.2.2 `Clothoids`

Questa libreria nasce per il *fitting* dei polinomi di Hermite di tipo G^1 e G^2 con clotoidi, *spline* di clotoidi, archi circolari e *biarc*. In questo lavoro di tesi la libre-

ria `Clothoids` è stata usata per sfruttare l'implementazione dell'oggetto albero per oggetti di tipo AABB.

5.3 Utilizzo e Prestazioni

Conclusioni e Lavoro Futuro 6

Convenzioni e Notazioni A

A.0.1 Sistemi di Riferimento

La convenzione utilizzata per definire gli assi del sistema di riferimento della vettura è la *International Organization for Standardization* (ISO) 8855.

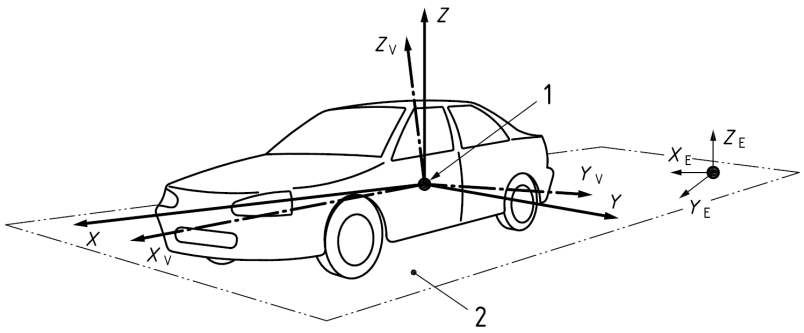


Figura A.1: Rappresentazione degli assi del sistema di riferimento della vettura secondo la convenzione ISO-V.

Il sistema di riferimento della ruota è conforme alla convenzione ISO-V, la cui disposizione degli assi è illustrata nella Figura A.2. L'origine del sistema di riferimento del vettore ruota è posta in corrispondenza del centro della ruota mentre posizione e orientamento relativi rispetto al sistema di riferimento del telaio sono definiti attraverso il modello della sospensione descritto in [4].

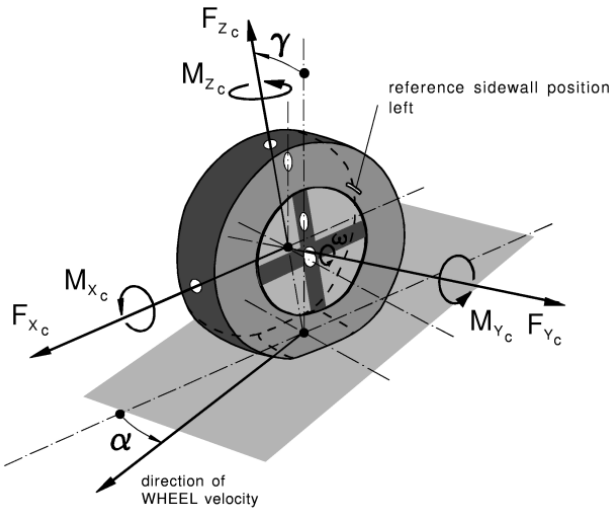


Figura A.2: Rappresentazione degli assi del sistema di riferimento dello pneumatico secondo la convenzione ISO-C.

A.0.2 Matrice di Trasformazione

Per descrivere sia l'orientamento che la posizione di un sistema di assi nello spazio, viene introdotta la matrice roto-traslazione, chiamata anche matrice di trasformazione. Questa notazione permette di impiegare le operazioni matrice-vettore per l'analisi di posizione, velocità e accelerazione. La forma generale di una matrice di trasformazione è del tipo:

$$T_m = \left[\begin{array}{ccc|c} & [R_m] & & \begin{matrix} O_{mx} \\ O_{my} \\ O_{mz} \end{matrix} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{A.1}$$

dove R_m è la matrice di rotazione 3×3 del sistema di riferimento in movimento e O_{mx}, O_{my} e O_{mz} sono le coordinate della sua origine nel sistema di riferimento assoluto o nativo.

L'introduzione dell'elemento fittizio 1 nel vettore della posizione di origine e la successiva spaziatura interna zero della matrice rende possibili le moltiplicazioni matrice-vettore, rendendo la matrice di trasformazione una notazione compatta e conveniente per la descrizione dei sistemi di riferimento. Si noti che per i vettori, le informazioni traslazionali vengono trascurate imponendo l'elemento fittizio pari a 0.

Codice della Libreria C++ B

B.1 TireGround.hh

```
/*!

\mainpage

Master's Thesis

Algorithm for Tire Contact Patch Evaluation in Soft Real-Time

Academic Year 2019 · 2020

Graduant:
--------

Davide Stocco\n
Department of Industrial Engineering\n
University of Trento\n
davide.stocco@studenti.unitn.it

Supervisor:
----------

Prof. Enrico Bertolazzi\n
Department of Industrial Engineering\n
University of Trento\n
enrico.bertolazzi@unitn.it

 */

///
/// file: TireGround.hh
///

#pragma once

#include <Eigen/Dense>  // Eigen linear algebra Library
#include <chrono>       // chrono - STD Time Measurement Library
```

```
37 #include <cmath>        // Math.h - STD math Library
38 #include <fstream>      // fStream - STD File I/O Library
39 #include <iostream>     // Iostream - STD I/O Library
40 #include <string>       // String - STD String Library
41 #include <vector>       // Vector - STD Vector/Array Library
42
43 namespace TireGround {
44
45   typedef double real_type; //!< Real number type
46   typedef int    int_type;  //!< Integer number type
47
48   typedef Eigen::Vector2d vec2; //!< 2D vector type
49   typedef Eigen::Vector3d vec3; //!< 3D vector type
50   typedef Eigen::Vector4d vec4; //!< 3D vector type
51   typedef Eigen::Matrix3d mat3; //!< 3x3 matrix type
52   typedef Eigen::Matrix4d mat4; //!< 4x4 matrix type
53
54   typedef Eigen::Matrix<vec2,1,Eigen::Dynamic>              row_vec2; //!< Row vector type of 2D vector
55   typedef Eigen::Matrix<vec2,Eigen::Dynamic,1>              col_vec2; //!< Column vector type of 2D vector
56   typedef Eigen::Matrix<vec2,Eigen::Dynamic,Eigen::Dynamic> mat_vec2; //!< Matrix type of 2D vector
57
58   typedef Eigen::Matrix<vec3,1,Eigen::Dynamic>              row_vec3; //!< Row vector type of 3D vector
59   typedef Eigen::Matrix<vec3,Eigen::Dynamic,1>              col_vec3; //!< Column vector type of 3D vector
60   typedef Eigen::Matrix<vec3,Eigen::Dynamic,Eigen::Dynamic> mat_vec3; //!< Matrix type of 3D vector
61
62   typedef std::basic_ostream<char> ostream_type; //!< Output stream type
63
64   real_type const epsilon = std::numeric_limits<real_type>::epsilon(); //!< Epsilon type
65
66 }  // namespace TireGround
67
68 ///
69 /// eof: TireGround.hh
70 ///
```

B.2 RoadRDF.hh

```
 1 ///
 2 /// file: MeshRDF.hh
 3 ///
 4
 5 #pragma once
 6
 7 #include <AABBtree.hh>
 8 #include "TireGround.hh"
 9
10 // Print progress to console while loading (large models)
11 #define RDF_CONSOLE_OUTPUT
12
13 #ifndef RDF_ERROR
14   #define RDF_ERROR(MSG) {                \
15     std::ostringstream ost; ost << MSG;    \
16     throw std::runtime_error( ost.str() ); \
17   }
18 #endif
19
20 #ifndef RDF_ASSERT
21   #define RDF_ASSERT(COND,MSG) \
22     if ( !(COND) ) RDF_ERROR( MSG )
23 #endif
24
25 //! RDF mesh computations routine
26 namespace RDF {
```

```
27
28  using namespace TireGround;
29
30  class TriangleRoad;
31
32  typedef std::shared_ptr<TriangleRoad> TriangleRoad_ptr;  //!< Shared pointer to TriangleRoad object
33  typedef std::vector<TriangleRoad_ptr> TriangleRoad_list; //!< Vector of shared pointers to TriangleRoad
        objects
34
35  /*\
36   |  ____  ____            ____  ____
37   | | __ )| __ )  _____  _|___ \|  _ \
38   | |  _ \|  _ \ / _ \ \/ / __) | | | |
39   | | |_) | |_) | (_) >  < / __/| |_| |
40   | |____/|____/ \___/_/\_\_____|____/
41   |
42  \*/
43
44  //! Class that handle BBox2D.
45  class BBox2D {
46  private:
47    real_type Xmin; //!< Xmin shadow domain point
48    real_type Ymin; //!< Ymin shadow domain point
49    real_type Xmax; //!< Xmax shadow domain point
50    real_type Ymax; //!< Ymax shadow domain point
51
52  public:
53
54    //! Default constructor for BBox2D object.
55    BBox2D() {}
56
57    //! Variable set constructor for BBox2D object.
58    BBox2D( vec3 const Vertices[3] ) {
59      updateBBox2D( Vertices );
60    }
61
62    //! Set Xmin shadow domain point.
63    void
64    setXmin(real_type const _Xmin) { Xmin = _Xmin; }
65
66    //! Set Ymin shadow domain point.
67    void
68    setYmin(real_type const _Ymin) { Ymin = _Ymin; }
69
70    //! Set Xmax shadow domain point.
71    void
72    setXmax(real_type const _Xmax) { Xmax = _Xmax; }
73
74    //! Set Ymax shadow domain point.
75    void
76    setYmax(real_type const _Ymax) { Ymax = _Ymax; }
77
78    //! Get Xmin shadow domain point.
79    real_type
80    getXmin(void) const { return Xmin; }
81
82    //! Get Ymin shadow domain point.
83    real_type
84    getYmin(void) const { return Ymin; }
85
86    //! Get Xmax shadow domain point.
87    real_type
88    getXmax(void) const { return Xmax; }
89
90    //! Get Ymax shadow domain point.
```

```
91    real_type
92    getYmax(void) const { return Ymax; }
93
94    //! Clear the bounding box domain.
95    void clear(void);
96
97    //! Print bounding box vertices.
98    void
99    print(ostream_type & stream) const {
100     stream
101       << "BBOX (xmin,ymin,xmax,ymax) = ( " << Xmin << ", " << Ymin
102       << ", " << Xmax << ", " << Ymax << " )\n";
103   }
104
105   //! Update the bounding box domain with multiple input triangles object.
106   void
107   updateBBox2D( vec3 const Vertices[3] );
108 };
109
110 /*\
111  |   _____     _                   _      _____ ____
112  |  |_   _| __(_) __ _ _ __   __ _| | ___|___ /|  _ \
113  |    | || '__| |/ _` | '_ \ / _` | |/ _ \ |_ \| | | |
114  |    | || |  | | (_| | | | | (_| | |  __/___) | |_| |
115  |    |_||_|  |_|\__,_|_| |_|\__, |_|\___|____/|____/
116  |                            |___/
117 \*/
118
119 //! Class for handling 3D triangles.
120 class Triangle3D {
121 protected:
122   vec3   Vertices[3];  //!< Vertices vector
123   BBox2D TriangleBBox; //!< Triangle bounding box
124
125   Triangle3D( Triangle3D const & ) = delete;               //!< Copy constructor
126   Triangle3D & operator = ( Triangle3D const & ) = delete; //!< Copy operator
127
128 public:
129   //! Variable set constructor for Triangle3D object.
130   Triangle3D() {
131     Vertices[0] = vec3(0,0,0);
132     Vertices[1] = vec3(0,0,0);
133     Vertices[2] = vec3(0,0,0);
134     TriangleBBox.updateBBox2D(Vertices);
135   }
136
137   //! Variable set constructor for Triangle3D object.
138   Triangle3D( vec3 const _Vertices[3] ) {
139     Vertices[0] = _Vertices[0];
140     Vertices[1] = _Vertices[1];
141     Vertices[2] = _Vertices[2];
142     TriangleBBox.updateBBox2D(Vertices);
143   }
144
145   //! Get triangle face normal versor.
146   vec3
147   Normal(void) const {
148     vec3 d1 = Vertices[1] - Vertices[0];
149     vec3 d2 = Vertices[2] - Vertices[0];
150     return d1.cross(d2).normalized();
151   }
152
153   //! Set vertices vector and update bounding box domain.
154   void
155   setVertices(
```

```
156       vec3 const _Vertices[3] //!< New vertices vector
157     ) {
158       Vertices[0] = _Vertices[0];
159       Vertices[1] = _Vertices[1];
160       Vertices[2] = _Vertices[2];
161       TriangleBBox.updateBBox2D(Vertices);
162     }
163
164     //! Set vertices vector and update bounding box domain.
165     void
166     setVertices(
167       vec3 const & Vertex_0,
168       vec3 const & Vertex_1,
169       vec3 const & Vertex_2
170     ) {
171       Vertices[0] = Vertex_0;
172       Vertices[1] = Vertex_1;
173       Vertices[2] = Vertex_2;
174       TriangleBBox.updateBBox2D(Vertices);
175     }
176
177     //! Get i-th vertex.
178     vec3 const &
179     getithVertex( unsigned i ) const { return Vertices[i]; }
180
181     //! Get triangle bonding box.
182     BBox2D const &
183     getBBox(void) const { return TriangleBBox; }
184
185     //! Print vertices information.
186     void
187     print( ostream_type & stream ) const {
188       stream
189         << "V0:\t" << Vertices[0].x() << ", " << Vertices[0].y() << ", " << Vertices[0].z() << "\n"
190         << "V0:\t" << Vertices[1].x() << ", " << Vertices[1].y() << ", " << Vertices[1].z() << "\n"
191         << "V0:\t" << Vertices[2].x() << ", " << Vertices[2].y() << ", " << Vertices[2].z() << "\n";
192     }
193
194     //! Check if a ray hits a triangle object through Möller-Trumbore
195     //! intersection algorithm.
196     bool
197     intersectRay(
198         vec3 const & RayOrigin,    //!< Ray origin position
199         vec3 const & RayDirection, //!< Ray direction vector
200         vec3       & IntPt         //! Intersection point
201     ) const;
202
203     //! Check if a side of the triangle hits a and find the intersection point.
204     bool
205     intersectSidePlane( vec3     const & planeN, //!< Plane normal vector
206                         vec3     const & planeP, //!< Plane known point
207                         int_type         Side,   //!< Triangle side number (1:3)
208                         vec3           & IntPt   //!< Intersection point
209     ) const;
210
211     //! Check if a plane intersects a triangle object and find the
212     //! intersection points.
213     bool
214     intersectPlane(
215         vec3              const & planeN, //!< Plane normal vector
216         vec3              const & planeP, //!< Plane known point
217         std::vector<vec3>       & IntPts  //!< Intersection points
218     ) const;
219
220   };
```

```
221
222  /*\
223   |   _____     _                    _      ____                  _
224   |  |_   _| __(_) __ _ _ __   __ _| | ___|  _ \ ___   __ _  __| |
225   |    | || '__| |/ _` | '_ \ / _` | |/ _ \ |_) / _ \ / _` |/ _` |
226   |    | || |  | | (_| | | | | (_| | |  __/  _ < (_) | (_| | (_| |
227   |    |_||_|  |_|\__,_|_| |_|\__, |_|\___|_| \_\___/ \__,_|\__,_|
228   |                           |___/
229  \*/
230
231  //! Class for handling Road triangles.
232  class TriangleRoad : public Triangle3D {
233  private:
234    real_type Friction;     //!< Face friction coefficient
235
236    TriangleRoad( TriangleRoad const & ) = delete;               //!< Copy constructor
237    TriangleRoad & operator = ( TriangleRoad const & ) = delete; //!< Copy operator
238
239  public:
240    //! Variable set constructor for TriangleRoad object.
241    TriangleRoad() : Triangle3D()
242    { Friction = 0.0;
243     }
244
245    //! Variable set constructor for TriangleRoad object.
246    TriangleRoad( vec3      const _Vertices[3],
247                  real_type       _Friction
248    ) : Triangle3D( _Vertices ) {
249      Friction    = _Friction;
250    }
251
252    //! Set friction.
253    void
254    setFriction( real_type _Friction ) { Friction = _Friction; }
255
256    //! Get friction coefficent on the face.
257    real_type
258    getFriction(void) const  { return Friction; }
259  };
260
261  /*\
262   |         _                  _ _   _
263   |    __ _| | __ _  ___  _ __(_) |_| |__  _ __ ___  ___
264   |   / _` | |/ _` |/ _ \| '__| | __| '_ \| '_ ` _ \/ __|
265   |  | (_| | | (_| | (_) | |  | | |_| | | | | | | | \__ \
266   |   \__,_|_|\__, |\___/|_|  |_|\__|_| |_|_| |_| |_|___/
267   |           |___/
268  \*/
269
270  //! Algorithms for RDF mesh computations routine.
271  namespace algorithms {
272
273    //! Split a string into a string array at a given token.
274    void
275    split(
276      std::string const        & in,   //!< Input string
277      std::vector<std::string> & out,  //!< Output string vector
278      std::string const        & token //!< Token
279    );
280
281    //! Get tail of string after first token and possibly following spaces.
282    std::string
283    tail( std::string const & in );
284
285    //! Get first token of string.
```

```
286   std::string
287   firstToken( std::string const & in );
288
289   //! Get element at given index position.
290   template<typename T>
291   T const &
292   getElement(
293     std::vector<T> const & elements, //!< Elements vector
294     std::string    const & index     //!< Index position
295   );
296 }  // namespace algorithms
297
298 /*\
299  |   __  __           _     ____              __
300  |  |  \/  | ___  ___| |__ / ___| _   _ _ __ / _| __ _  ___ ___
301  |  | |\/| |/ _ \/ __| '_ \\___ \| | | | '__| |_ / _` |/ __/ _ \
302  |  | |  | |  __/\__ \ | | |___) | |_| | |  |  _| (_| | (_|  __/
303  |  |_|  |_|\___||___/_| |_|____/ \__,_|_|  |_|  \__,_|\___\___|
304  |
305 \*/
306
307 //! Class for handling MeshSurface object.
308 class MeshSurface {
309 private:
310   TriangleRoad_list                PtrTriangleVec; //!< Triangles ptr vector list
311   std::vector<G2lib::BBox::PtrBBox> PtrBBoxVec;     //!< Bounding boxes pointers
312   G2lib::AABBtree::PtrAABB          PtrTree
313             = std::make_shared<G2lib::AABBtree>(); //!< Mesh tree pointer
314
315   MeshSurface( MeshSurface const & ) = delete;               //!< Copy constructor
316   MeshSurface & operator = ( MeshSurface const & ) = delete; //!< Copy operator
317
318 public:
319   //! Default set constructor for MeshSurface object.
320   MeshSurface() {};
321
322   //! Variable set constructor for MeshSurface object.
323   MeshSurface(
324     TriangleRoad_list const & _PtrTriangleVec //!< Triangles ptr vector list
325   ) {
326     this->PtrTriangleVec = _PtrTriangleVec;
327     updatePtrBBox();
328     PtrTree->build(PtrBBoxVec);
329   };
330
331   //! Variable set constructor for MeshSurface object.
332   MeshSurface(
333     std::string const & Path //!< Path to the RDF file
334   ) {
335     bool load = LoadFile(Path);
336     RDF_ASSERT( load, "Error while reading file" );
337   }
338
339   //! Get all triangles inside the mesh as a vector.
340   TriangleRoad_list const &
341   getTrianglesPtr(void) const
342   { return PtrTriangleVec; }
343
344   //! Get i-th triangle.
345   TriangleRoad_ptr const &
346   getithTrianglePtr( unsigned i ) const
347   { return PtrTriangleVec[i]; }
348
349   //! Get AABB tree.
350   G2lib::AABBtree::PtrAABB
```

```
351     getAABBPtr(void) const
352     { return PtrTree; }
353
354     //! Print data in file.
355     void printData( std::string const & FileName );
356
357     //! Get the mesh G2lib bounding boxes pointers vector.
358     std::vector<G2lib::BBox::PtrBBox> const &
359     getPtrBBox() const
360     { return PtrBBoxVec; }
361
362     //! Copy the mesh.
363     void
364     set( MeshSurface const & in ) {
365       this->PtrTriangleVec = in.PtrTriangleVec;
366       this->PtrBBoxVec     = in.PtrBBoxVec;
367       this->PtrTree        = in.PtrTree;
368     }
369
370     //! Load the RDF model and print information on a file.
371     //! If RDF model is properly loaded true value is returned.
372     bool
373     LoadFile(
374       std::string const & Path //!< Path to the RDF file
375     );
376
377     //! Update the local intersected triangles list.
378     RDF::TriangleRoad_list
379     updateIntersection(
380       G2lib::AABBtree::PtrAABB const & ExternalTreePtr //!< External AABB tree ptr
381     );
382
383   private:
384     //! Update the mesh G2lib bounding boxes pointers vector.
385     void updatePtrBBox(void);
386
387     //! Generate vertices from a list of positions face line.
388     void
389     GenVerticesFromRawRDF(
390       std::vector<vec3> const & iNodes,
391       std::string       const & icurline,
392       vec3                      oVerts[3]
393     );
394   };
395
396 }  // namespace RDF
397
398 ///
399 /// eof: MeshRDF.hh
400 ///
```

B.3 RoadRDF.cc

```
 1 #include "RoadRDF.hh"  // RDF file extention Loader
 2
 3
 4 //! RDF mesh computations routine
 5 namespace RDF {
 6
 7   /*\
 8    |  ____  ____            ____  ____
 9    | | __ )| __ )  _____  _|___ \|  _ \
10    | |  _ \|  _ \ / _ \ \/ / __) | | | |
```

```
11   | | |_) | |_) | (_) >  < / __/| |_| |
12   | |____/|____/ \___/_/\_\_____|____/
13   |
14  \*/
15
16  //! Clear the bounding box domain.
17  void
18  BBox2D::clear(void) {
19    Xmin = std::numeric_limits<real_type>::quiet_NaN();;
20    Ymin = std::numeric_limits<real_type>::quiet_NaN();;
21    Xmax = std::numeric_limits<real_type>::quiet_NaN();;
22    Ymax = std::numeric_limits<real_type>::quiet_NaN();;
23  }
24
25  //! Update the bounding box domain with multiple input triangles object.
26  void
27  BBox2D::updateBBox2D( vec3 const Vertices[3] ) {
28    G2lib::minmax3( Vertices[0].x(), Vertices[1].x(), Vertices[2].x(), Xmin, Xmax );
29    G2lib::minmax3( Vertices[0].y(), Vertices[1].y(), Vertices[2].y(), Ymin, Ymax );
30  }
31
32  /*\
33   |   _____     _                    _      _____ ____
34   |  |_   _| __(_) __ _ _ __   __ _| | ___|___ /|  _ \
35   |    | || '__| |/ _` | '_ \ / _` | |/ _ \ |_ \| | | |
36   |    | || |  | | (_| | | | | (_| | |  __/___) | |_| |
37   |    |_||_|  |_|\__,_|_| |_|\__, |_|\___|____/|____/
38   |                            |___/
39  \*/
40
41  //! Check if a ray hits a triangle object through Möller-Trumbore
42  //! intersection algorithm.
43  bool
44  Triangle3D::intersectRay(
45      vec3 const & RayOrigin,    //!< Ray origin position
46      vec3 const & RayDirection, //!< Ray direction vector
47      vec3       & IntPt         //! Intersection point
48  ) const {
49    vec3 E1(Vertices[1] - Vertices[0]);
50    vec3 E2(Vertices[2] - Vertices[0]);
51    vec3 A(RayDirection.cross(E2));
52    real_type det = A.dot(E1);
53    real_type t_param;
54
55    if (det > epsilon) {
56      vec3 T(RayOrigin - Vertices[0]);
57      real_type u = A.dot(T);
58      if (u < 0.0 || u > det)
59        return false;
60      vec3 B(T.cross(E1));
61      real_type v = B.dot(RayDirection);
62      if (v < 0.0 || u + v > det)
63        return false;
64      t_param = (B.dot(E2)) / det;
65    } else if (det < -epsilon) {
66      vec3 T(RayOrigin - Vertices[0]);
67      real_type u = A.dot(T);
68      if (u > 0.0 || u < det)
69        return false;
70      vec3 B(T.cross(E1));
71      real_type v = B.dot(RayDirection);
72      if (v > 0.0 || u + v < det)
73        return false;
74      t_param = (B.dot(E2)) / det;
75    } else {
```

```
76      return false;
77    }
78    // At this stage we can compute t to find out where the intersection
79    // point is on the line
80    if (t_param >= 0) {  // ray intersection
81      IntPt = RayOrigin + RayDirection * t_param;
82      return true;
83    } else {
84      // This means that there is a line intersection on negative side
85      return false;
86    }
87  }
88
89  //! Check if a side of the triangle hits a and find the intersection point.
90  bool
91  Triangle3D::intersectSidePlane(vec3    const & planeN, //!< Plane normal vector
92                                 vec3    const & planeP, //!< Plane known point
93                                 int_type        Side,   //!< Triangle side number (1:3)
94                                 vec3          & IntPt   //!< Intersection point
95  ) const {
96    vec3 PointA, PointB;
97    // Check that side number is between 1 and 3
98    RDF_ASSERT( (Side >=1 && Side <= 3) , "Side number must be between 1 and 3.")
99    if ( Side == 1) {
100     PointA = Vertices[0];
101     PointB = Vertices[1];
102   } else if ( Side == 2 ) {
103     PointA = Vertices[1];
104     PointB = Vertices[2];
105   } else if ( Side == 3 ) {
106     PointA = Vertices[2];
107     PointB = Vertices[0];
108   }
109
110   vec3 Direction(PointB - PointA);
111   real_type d = planeP.dot(-planeN);
112   real_type t = -(PointA.dot(planeN) + d) / (Direction.dot(planeN));
113   if (t >= 0 && t <= 1) {
114     IntPt = PointA + Direction * t;
115     return true;
116   } else {
117     return false;
118   }
119 }
120
121 //! Check if a plane intersects a triangle object and find the
122 //! intersection points.
123 bool
124 Triangle3D::intersectPlane(
125     vec3              const & planeN, //!< Plane normal vector
126     vec3              const & planeP, //!< Plane known point
127     std::vector<vec3>       & IntPts  //!< Intersection points
128 ) const {
129   // Clear intersection points vector
130   IntPts.clear();
131
132   // Fill intersection points vector
133   vec3 IntPt1, IntPt2, IntPt3;
134   if ( intersectSidePlane( planeN, planeP, 1, IntPt1 ) )
135     IntPts.push_back(IntPt1);
136   if ( intersectSidePlane( planeN, planeP, 2, IntPt2 ) )
137     IntPts.push_back(IntPt2);
138   if ( intersectSidePlane( planeN, planeP, 3, IntPt3 ) )
139     IntPts.push_back(IntPt3);
140   // Check the results (there must be only 2 intersection points)
```

```
141     if ( IntPts.size() == 2 ) { return true; }
142     else if ( IntPts.size() == 0 ) { return false; }
143     else { RDF_ERROR( "\nPlane-Triangle intersection not handled. There are " << IntPts.size() << "
            intersections."); }
144   }
145
146   /*\
147    |         _                    _ _    _
148    |    __ _| | __ _  ___  _ __(_) |_| |__  _ __ ___  ___
149    |   / _` | |/ _` |/ _ \| '__| | __| '_ \| '_ ` _ \/ __|
150    |  | (_| | | (_| | (_) | |  | | |_| | | | | | | | \__ \
151    |   \__,_|_|\__, |\___/|_|  |_|\__|_| |_|_| |_| |_|___/
152    |           |___/
153   \*/
154
155   //! Holds all of the algorithms needed for the mesh processing
156   namespace algorithms {
157
158     //! Split a string into a string array at a given token.
159     void
160     split(
161       std::string const        & in,   //!< Input string
162       std::vector<std::string> & out,  //!< Output string vector
163       std::string const        & token //!< Token
164     ) {
165       out.clear();
166       std::string temp;
167
168       for ( int i = 0; i < int(in.size()); ++i ) {
169         std::string test = in.substr(i, token.size());
170         if (test == token) {
171           if (!temp.empty()) {
172             out.push_back(temp);
173             temp.clear();
174             i += (int)token.size() - 1;
175           } else {
176             out.push_back("");
177           }
178         } else if (i + token.size() >= in.size()) {
179           temp += in.substr(i, token.size());
180           out.push_back(temp);
181           break;
182         } else {
183           temp += in[i];
184         }
185       }
186     }
187
188     //! Get tail of string after first token and possibly following spaces.
189     std::string
190     tail(std::string const & in ) {
191       size_t token_start = in.find_first_not_of(" \t");
192       size_t space_start = in.find_first_of(" \t", token_start);
193       size_t tail_start  = in.find_first_not_of(" \t", space_start);
194       size_t tail_end    = in.find_last_not_of(" \t");
195       if (tail_start != std::string::npos && tail_end != std::string::npos) {
196         return in.substr(tail_start, tail_end - tail_start + 1);
197       } else if (tail_start != std::string::npos) {
198         return in.substr(tail_start);
199       }
200       return "";
201     }
202
203     //! Get first token of string.
204     std::string
```

```
205   firstToken( std::string const & in ) {
206     if (!in.empty()) {
207       size_t token_start = in.find_first_not_of(" \t\r\n");
208       if (token_start != std::string::npos) {
209         size_t token_end = in.find_first_of(" \t\r\n", token_start);
210         if (token_end != std::string::npos) {
211           return in.substr(token_start, token_end - token_start);
212         } else {
213           return in.substr(token_start);
214         }
215       }
216     }
217     return "";
218   }
219
220   //! Get element at given index position.
221   template<typename T>
222   T const &
223   getElement(
224     std::vector<T> const & elements,  //!< Elements vector
225     std::string    const & index         //!< Index position
226   ) {
227     // std::cout << "Index: " << index << std::endl;
228     int_type id = std::stoi(index);
229     if ( id < 0 ) std::cerr << "ELEMENTS indexes cannot be negative\n";
230     return elements[id - 1];
231   }
232 }  // namespace algorithms
233
234 /*\
235  |   __  __           _     ____              __
236  |  |  \/  | ___  ___| |__ / ___| _   _ _ __ / _| __ _  ___ ___
237  |  | |\/| |/ _ \/ __| '_ \\___ \| | | | '__| |_ / _` |/ __/ _ \
238  |  | |  | |  __/\__ \ | | |___) | |_| | |  |  _| (_| | (_|  __/
239  |  |_|  |_|\___||___/_| |_|____/ \__,_|_|  |_|  \__,_|\___\___|
240  |
241 \*/
242
243 //! Print data in file.
244 void
245 MeshSurface::printData( std::string const & FileName ) {
246   // Create/Open Out.txt
247   std::ofstream file(FileName);
248
249   // Print introduction
250   file
251     << "LOADED RDF MESH DATA\n\n"
252     << "Legend:\n"
253     << "\tVi: i-th vertex\n"
254     << "\t N: normal to the face\n"
255     << "\t F: friction coefficient\n\n";
256
257   for ( unsigned i = 0; i < PtrTriangleVec.size(); ++i ) {
258     TriangleRoad const & Ti = *PtrTriangleVec[i];
259     vec3 const & V0( Ti.getithVertex(0) );
260     vec3 const & V1( Ti.getithVertex(1) );
261     vec3 const & V2( Ti.getithVertex(2) );
262     vec3          N = Ti.Normal();
263
264     // Print vertices, normal and friction
265     file
266       << "TRIANGLE " << i
267       << "\n\tV0:\t" << V0.x() << ", " << V0.y() << ", " << V0.z()
268       << "\n\tV1:\t" << V1.x() << ", " << V1.y() << ", " << V1.z()
269       << "\n\tV2:\t" << V2.x() << ", " << V2.y() << ", " << V2.z()
```

```
270         << "\n\t N:\t" << N.x()  << ", " << N.y()  << ", " << N.z()
271         << "\n\t F:\t" << Ti.getFriction()
272         << "\n\n";
273     }
274     // Close File
275     file.close();
276   }
277
278   //! Update the mesh G2lib bounding boxes pointers vector
279   void
280   MeshSurface::updatePtrBBox(void) {
281     PtrBBoxVec.clear();
282     RDF::BBox2D iBBox;
283     for (unsigned id = 0; id < PtrTriangleVec.size(); ++id) {
284       iBBox = (*PtrTriangleVec[id]).getBBox();
285       PtrBBoxVec.push_back(G2lib::BBox::PtrBBox(
286           new G2lib::BBox(iBBox.getXmin(), iBBox.getYmin(), iBBox.getXmax(),
287                           iBBox.getYmax(), id, 0)));
288       iBBox.clear();
289     }
290   }
291
292   //! Load the RDF model and print information on a file
293   bool
294   MeshSurface::LoadFile( std::string const & Path ) {
295     // Check if the file is an ".rdf" file, if not return false
296     if (Path.substr(Path.size() - 4, 4) != ".rdf") {
297       std::cerr << "Not a RDF file\n";
298       return false;
299     }
300
301      // Check if the file had been correctly open, if not return false
302     std::ifstream file(Path);
303     if (!file.is_open()) {
304       std::cerr << "RDF file not opened\n";
305       return false;
306     }
307
308     // Vector for nodes coordinates
309     std::vector<vec3> Nodes;
310
311     bool nodes_parse    = false;
312     bool elements_parse = false;
313
314 #ifdef RDF_CONSOLE_OUTPUT
315     int_type const outputEveryNth = 5000;
316     int_type outputIndicator      = outputEveryNth;
317 #endif
318
319     std::string curline;
320     while (std::getline(file, curline)) {
321 #ifdef RDF_CONSOLE_OUTPUT
322       if ((outputIndicator = ((outputIndicator + 1) % outputEveryNth)) == 1) {
323         std::cout
324           << "\r- "
325           << "Loading mesh..."
326           << "\t triangles > "
327           << PtrTriangleVec.size() << std::endl;
328       }
329 #endif
330
331       std::string token = algorithms::firstToken(curline);
332       if ( token == "[NODES]" || token == "NODES" ) {
333         nodes_parse    = true;
334         elements_parse = false;
```

```
335         continue;
336       } else if (token == "[ELEMENTS]" || token == "ELEMENTS") {
337         nodes_parse    = false;
338         elements_parse = true;
339         continue;
340       } else if (token[0] == '{') {
341         // commento multiriga, continua a leggere fino a che trovo '}'
342         continue;
343       } else if (token[0] == '%' || token[0] == '#' || token[0] == '\r') {
344         // Check comments or empty lines
345         continue;
346       }
347
348       // Generate a vertex position
349       if (nodes_parse) {
350         std::vector<std::string> spos;
351         vec3 vpos;
352
353         algorithms::split(algorithms::tail(curline), spos, " ");
354
355         vpos[0] = std::stod(spos[0]);
356         vpos[1] = std::stod(spos[1]);
357         vpos[2] = std::stod(spos[2]);
358         Nodes.push_back(vpos);
359       }
360
361       // Generate a face (vertices & indices)
362       if (elements_parse) {
363         // Generate the triangle vertices from the elements
364         vec3 iVerts[3];
365         GenVerticesFromRawRDF( Nodes, curline, iVerts );
366
367         // Get the triangle friction from current line
368         std::vector<std::string> curlinevec;
369         algorithms::split(curline, curlinevec, " ");
370         real_type iFriction = std::stod(curlinevec[4]);
371
372         // Create a shared pointer for the last triangle and push it in the pointer vector
373         PtrTriangleVec.push_back(TriangleRoad_ptr(new TriangleRoad(iVerts,iFriction)));
374       }
375     }
376
377 #ifdef RDF_CONSOLE_OUTPUT
378     std::cout << std::endl;
379 #endif
380
381     file.close();
382
383     if (PtrTriangleVec.empty()) {
384       perror("Loaded mesh is empty");
385       return false;
386     } else {
387       // Update the local intersected triangles list
388 #ifdef RDF_CONSOLE_OUTPUT
389     std::cout << std::endl << "Building AABB tree... ";
390 #endif
391       updatePtrBBox();
392       PtrTree->build(PtrBBoxVec);
393 #ifdef RDF_CONSOLE_OUTPUT
394     std::cout << "Done" << std::endl << std::endl;
395 #endif
396
397       return true;
398     }
399   }
```

```
400
401   // Generate vertices from a list of positions face line.
402   void
403   MeshSurface::GenVerticesFromRawRDF(
404     std::vector<vec3> const & iNodes,
405     std::string       const & icurline,
406     vec3                      oVerts[3]
407   ) {
408     std::vector<std::string> svert;
409     vec3                     vVert;
410     algorithms::split( icurline, svert, " " );
411
412     int_type control_size = int(svert.size() - 4);
413     for ( int i = 1; i < int(svert.size() - control_size); ++i ) {
414       // Calculate and store the vertex
415       vVert = algorithms::getElement(iNodes, svert[i]);
416       oVerts[i-1] = vVert; // CONTROLLARE i <= 3!!!!!!!!!!!!!!!!!!!!!!!!
417     }
418   }
419
420   //! Update the local intersected triangles list.
421   RDF::TriangleRoad_list
422   MeshSurface::updateIntersection(
423     G2lib::AABBtree::PtrAABB const & ExternalTreePtr
424   ) {
425     RDF::TriangleRoad_list intersectionTriPtr;
426     G2lib::AABBtree::VecPairPtrBBox intersectionList;
427     (*PtrTree).intersect(*ExternalTreePtr, intersectionList);
428     for ( unsigned i = 0; i < intersectionList.size(); ++i ) {
429       intersectionTriPtr.push_back(
430         getithTrianglePtr((*intersectionList[i].first).Id()));
431     }
432     return intersectionTriPtr;
433   }
434
435 }  // namespace RDF
436
437 ///
438 /// eof: MeshRDF.hh
439 ///
```

B.4 PatchTire.hh

```
 1 ///
 2 /// file: PatchTire.hh
 3 ///
 4
 5 #pragma once
 6
 7 #include "RoadRDF.hh"  // RDF file extention Loader
 8
 9 //! Tire computations routine
10 namespace PatchTire {
11
12   using namespace TireGround;
13
14   static real_type quiteNaN =
15       std::numeric_limits<real_type>::quiet_NaN(); //!< Not-a-Number type
16
17   static mat3 mat3_NaN = mat3::Constant(quiteNaN); //!< Not-a-Number 3x3 matrix type
18   static vec3 vec3_NaN = vec3::Constant(quiteNaN); //!< Not-a-Number 3D vector type
19
20   real_type const epsilon = std::numeric_limits<real_type>::epsilon(); //!< Epsilon (small number)
```

```
21
22  /*\
23   |   ____  _     _
24   |  |  _ \(_)___| | __
25   |  | | | | / __| |/ /
26   |  | |_| | \__ \   <
27   |  |____/|_|___/_|\_\
28   |
29   |  ^ Z
30   |  |     -----
31   |  |   /       \
32   |  |  |    O    |  O = OriginXZ
33   |  |   \       /
34   |  |     -----
35   |  Y------------------> X
36   |
37   |  ISO Reference Frame!
38  \*/
39
40  class ReferenceFrame;
41  class ETRTO;
42
43  //! Class for handling tire disks.
44  class Disk {
45  private:
46    vec2      OriginXZ; //!< X0,Z0 origin vector
47    real_type OffsetY;  //!< Y0 (= D) origin coordinate or offset from center
48    real_type Radius;   //!< Disk radius
49
50    Disk( Disk const & ) = delete;                      //!< Copy constructor
51    Disk const & operator = ( Disk const & ) = delete; //!< Copy operator
52
53  public:
54
55    Disk( Disk && ) = default; //!< Enable && operator
56
57    //! Default constructor for Disk object.
58    Disk()
59    : OriginXZ( vec2(quiteNaN, quiteNaN) )
60    , OffsetY( quiteNaN )
61    , Radius( quiteNaN )
62    {}
63
64    //! Variable set constructor for orientation object.
65    Disk(
66      vec2 const & _OriginXZ, //!< X0,Z0 origin coordinate
67      real_type    _OffsetY,  //!< Y0 (= D) origin coordinate
68      real_type    _Radius    //!< Disk radius
69    ) {
70      OriginXZ = _OriginXZ;
71      OffsetY  = _OffsetY;
72      Radius   = _Radius;
73    }
74
75    //! Copy class Disk.
76    void
77    set( Disk const & in ) {
78      this->OriginXZ = in.OriginXZ;
79      this->OffsetY  = in.OffsetY;
80      this->Radius   = in.Radius;
81    }
82
83    //! Set origin vector.
84    void
85    setOriginXZ( vec2 const & _OriginXZ )
```

```
86    { OriginXZ = _OriginXZ; }
87
88    //! Get origin vector XZ-axes coordinates.
89    vec2 const & getOriginXZ(void) const { return OriginXZ; }
90
91    //! Get origin vector XYZ-axes coordinates.
92    vec3 getOriginXYZ(void) const
93    { return vec3(OriginXZ.x(), OffsetY, OriginXZ.y()); }
94
95    //! Get origin Y-axis coordinate.
96    real_type getOffsetY(void) const { return OffsetY; }
97
98    //! Get disk radius.
99    real_type getRadius(void) const { return Radius; }
100
101   //! Check if a point lays inside or outside a circumfererence.
102   //! If output bool is true the point is inside the circumfererence,
103   //! otherwise it is outside.
104   bool
105   isPointInside(
106     vec2      const & Point   //!< Query point
107   ) const;
108
109   //! Find the intersection points between a circle and a line segment.
110   //! Output integer gives the number of intersection points.
111   int_type
112   intersectSegment(
113     vec2      const & Point_1,     //!< Line segment point 1
114     vec2      const & Point_2,     //!< Line segment point 2
115     vec2            & Intersect_1, //!< Intersection point 2
116     vec2            & Intersect_2  //!< Intersection point 2
117   ) const;
118
119   //! Check if two plane intersects and find the intersecting rect.
120   bool
121   intersectPlane(
122     vec3 const & Plane_Normal,
123     vec3 const & Plane_Point,
124     vec3       & Line_Direction,
125     vec3       & Line_Point
126   ) const;
127
128   //! Get the contact patch length inside the single disk of a segment described by
129   //! the intersection of triangles on XZ plane.
130   real_type
131   getPatchLength(
132     RDF::TriangleRoad_list const & intersectionTriPtr, //!< Local intersected Triangle3D vector
133     ReferenceFrame         const & RF                  //!< Tire ReferenceFrame object
134   ) const;
135
136   //! Get the contact patch length inside the single disk [m] given
137   //! a plane (normal and point).
138   real_type
139   getPatchLength(
140     vec3           const & Plane_Normal, //!< Plane normal in Disk RF
141     vec3           const & Plane_Point,  //!< Plane point in Disk RF
142     ReferenceFrame const & RF            //!< Tire ReferenceFrame object
143   ) const;
144
145   //! Get the contact patch length inside the single disk [m] given two
146   //! points in Disk RF.
147   real_type
148   getPatchLength(
149     vec2           const & PointXZ_1, //!< Point1 in Disk RF
150     vec2           const & PointXZ_2, //!< Point2 in Disk RF
```

```
151       ReferenceFrame const & RF         //!< ReferenceFrame object
152     ) const;
153
154     //! Get the contact patch length inside the single disk [m] give a sequence
155     //! points in Disk RF.
156     real_type
157     getPatchLength(
158       col_vec2       const & XZ_sequence, //!< Points sequence in Disk RF
159       ReferenceFrame const & RF           //!< ReferenceFrame object
160     ) const;
161
162   };
163
164   /*\
165    |   _____ _____ ____ _____ ___
166    |  | ____|_   _|  _ \_   _/ _ \
167    |  |  _|   | | | |_) || || | | |
168    |  | |___  | | |  _ < | || |_| |
169    |  |_____| |_| |_| \_\|_| \___/
170   \*/
171
172   //! Class for handling ETRTO tire data.
173   class ETRTO {
174   private:
175     real_type SectionWidth;   //!< Tire section width[mm]
176     real_type AspectRatio;    //!< Tire aspect ratio [%]
177     real_type RimDiameter;    //!< Rim diameter [in]
178     real_type SidewallHeight; //!< Sidewall height [m]
179     real_type TireDiameter;   //!< External diameter [m]
180
181     ETRTO( ETRTO const & ); //!< Copy constructor
182
183   public:
184     //! Default constructor for ETRTO object.
185     ETRTO() {}
186
187     //! Variable set constructor for ETRTO object (for Magic formula).
188     ETRTO(
189       real_type _SectionWidth, //!< Tire section width[mm]
190       real_type _AspectRatio,  //!< Tire aspect ratio [%]
191       real_type _RimDiameter   //!< Rim diameter [in]
192     ) {
193       SectionWidth = _SectionWidth;
194       AspectRatio  = _AspectRatio;
195       RimDiameter  = _RimDiameter;
196       calcSidewallHeight();
197       calcTireDiameter();
198     }
199
200     //! Operator = for ETRTO object.
201     ETRTO const &
202     operator = ( ETRTO const & in ) {
203       this->SectionWidth = in.SectionWidth;
204       this->AspectRatio  = in.AspectRatio;
205       this->RimDiameter  = in.RimDiameter;
206       calcSidewallHeight();
207       calcTireDiameter();
208       return *this;
209     }
210
211     //! Get sidewall height [m].
212     real_type getSidewallHeight(void) const { return SidewallHeight; }
213
214     //! Get external tire diameter [m].
215     real_type getTireDiameter(void) const { return TireDiameter; }
```

```
216
217   //! Get external tire radius [m].
218   real_type getTireRadius(void) const { return TireDiameter / 2.0; }
219
220   //! Get section width [m].
221   real_type getSectionWidth(void) const { return SectionWidth / 1000.0; }
222
223   //! Display tire data on stream.
224   void
225   print( ostream_type & stream ) const {
226     stream
227       << "Current Tire Data:\n"
228       << "\tSection width = " << SectionWidth << " mm\n"
229       << "\tAspect ratio  = " << AspectRatio << " %\n"
230       << "\tRim diameter  = " << RimDiameter << " in\n"
231       << "\tS.wall Height = " << getSidewallHeight() * 1000 << " mm\n"
232       << "\tTire diameter = " << getTireDiameter() * 1000 << " mm\n\n";
233   }
234
235 private:
236   //! Calculate sidewall height [m].
237   void
238   calcSidewallHeight(void)
239   { SidewallHeight = SectionWidth / 1000.0 * AspectRatio / 100; }
240
241   //! Calculate external tire diameter [m].
242   void
243   calcTireDiameter(void)
244   { TireDiameter = RimDiameter * 0.0254 + getSidewallHeight() * 2.0; }
245 };
246
247 /*\
248  |   ____       __                                  _____
249  |  |  _ \ ___ / _| ___ _ __ ___ _ __   ___ ___|  ___| __ __ _ _ __ ___   ___
250  |  | |_) / _ \ |_ / _ \ '__/ _ \ '_ \ / __/ _ \ |_  | '__/ _` | '_ ` _ \ / _ \
251  |  |  _ <  __/  _|  __/ | |  __/ | | | (_|  __/  _|| | | (_| | | | | | |  __/
252  |  |_| \_\___|_|  \___|_|  \___|_| |_|\___\___|_|  |_|  \__,_|_| |_| |_|\___|
253  |
254 \*/
255
256 //! Class for handling tire reference frame.
257 class ReferenceFrame {
258 private:
259   vec3 Origin         = vec3_NaN; //!< Origin position (default NaN)
260   mat3 RotationMatrix = mat3_NaN; //!< 3x3 rotation matrix (default NaN)
261
262   ReferenceFrame( ReferenceFrame const & ) = delete;               //!< Copy constructor
263   ReferenceFrame & operator = ( ReferenceFrame const & ) = delete; //!< Copy operator
264
265 public:
266   //! Default constructor for ReferenceFrame object.
267   ReferenceFrame() {}
268
269   //! Variable set constructor for ReferenceFrame object.
270   ReferenceFrame(
271     vec3 const & _Origin,        //!< Origin position
272     mat3 const & _RotationMatrix //!< Rotation matrix
273   ) {
274     Origin         = _Origin;
275     RotationMatrix = _RotationMatrix;
276   }
277
278   //! Check if ReferenceFrame object is empty or not (true == empty).
279   bool
280   isEmpty(void) {
```

```
281     if ( Origin != Origin || RotationMatrix != RotationMatrix){
282       return true;
283     } else {
284       return false;
285     }
286   }
287
288   //! Get current rotation matrix.
289   mat3 const & getRotationMatrix(void) const { return RotationMatrix; }
290
291   //! Get current rotation matrix inverse with LU decomposition.
292   vec3 getRotationMatrixInverse(vec3 const & Point) const {
293     // DA INDAGARE https://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html
294     Eigen::PartialPivLU<RDF::mat3> RF_LU(RotationMatrix);
295     return RF_LU.solve(Point);
296   }
297
298   //! Get current X-axis versor.
299   vec3 getX(void) const { return RotationMatrix.col(0); }
300
301   //! Get current Y-axis versor.
302   vec3 getY(void) const { return RotationMatrix.col(1); }
303
304   //! Get current Z-axis versor.
305   vec3 getZ(void) const { return RotationMatrix.col(2); }
306
307   //! Get origin position.
308   vec3 const & getOrigin(void) const { return Origin; }
309
310   //! Set origin position.
311   void setOrigin( vec3 const & _Origin ) { Origin = _Origin; }
312
313   //! Set rotation matrix.
314   void setRotationMatrix( mat3 const & _RotationMatrix ) { RotationMatrix = _RotationMatrix; }
315
316   //! Set total transformation matrix.
317   void
318   setTotalTransformationMatrix( mat4 const & TM ) {
319     Origin         = TM.block<3,1>(0,3);
320     RotationMatrix = TM.block<3,3>(0,0);
321   }
322
323   //! Get total transformation matrix.
324   mat4
325   getTotalTransformationMatrix(void) {
326     mat4 out;
327     out << RotationMatrix, Origin, vec4(0.0, 0.0, 0.0, 1.0).transpose();
328     return out;
329   }
330
331   //! Copy class ReferenceFrame.
332   void
333   set( ReferenceFrame const & in ) {
334     this->Origin         = in.Origin;
335     this->RotationMatrix = in.RotationMatrix;
336   }
337
338   //! Get current Euler angles [rad] for X-axis.
339   //! Warning: Factor as Rz*Rx*Ry!
340   // https://www.geometrictools.com/Documentation/EulerAngles.pdf
341   real_type getEulerAngleX(void) const;
342
343   //! Get current Euler angles [rad] for Y-axis.
344   //! Warning: Factor as Rz*Rx*Ry!
345   // https://www.geometrictools.com/Documentation/EulerAngles.pdf
```

```
346   real_type getEulerAngleY(void) const;
347
348   //! Get current Euler angles [rad] for Z-axis.
349   //! Warning: Factor as Rz*Rx*Ry!
350   // https://www.geometrictools.com/Documentation/EulerAngles.pdf
351   real_type getEulerAngleZ(void) const;
352
353 };
354
355 /*\
356  |   ____  _                 _
357  |  / ___|| |__   __ _  __| | _____      __
358  |  \___ \| '_ \ / _` |/ _` |/ _ \ \ /\ / /
359  |   ___) | | | | (_| | (_| | (_) \ V  V /
360  |  |____/|_| |_|\__,_|\__,_|\___/ \_/\_/
361 \*/
362
363 //! Class for tire shadow bounding box
364 class Shadow {
365 private:
366   real_type Xmin;                                //!< Xmin shadow domain point
367   real_type Ymin;                                //!< Ymin shadow domain point
368   real_type Xmax;                                //!< Xmax shadow domain point
369   real_type Ymax;                                //!< Ymax shadow domain point
370   std::vector<G2lib::BBox::PtrBBox> PtrBBoxVec; //!< Bounding boxes pointers
371   G2lib::AABBtree::PtrAABB PtrTree =
372       std::make_shared<G2lib::AABBtree>();      //!< Mesh tree pointer
373
374   Shadow( Shadow const & );                      //!< Copy constructor
375   Shadow const & operator = ( Shadow const & ); //!< Copy operator
376
377 public:
378   //! Default constructor for Shadow object.
379   Shadow() {}
380
381   //! Variable set constructor for Shadow object.
382   Shadow(
383     ETRTO          const & TireGeometry, //!< ETRTO tire denomination object
384     ReferenceFrame const & RF            //!< ReferenceFrame object
385   ) {
386     update( TireGeometry, RF );
387   }
388
389   //! Get tire shadow AABB tree.
390   G2lib::AABBtree::PtrAABB
391   getAABBPtr(void) const {
392     std::vector<G2lib::BBox::PtrBBox> PtrBBoxVec;
393     G2lib::AABBtree::PtrAABB PtrTree =
394       std::make_shared<G2lib::AABBtree>(); // Mesh tree pointer
395     updatePtrBBox( PtrBBoxVec );
396     PtrTree->build(PtrBBoxVec);
397     return PtrTree;
398   }
399
400   //! Get Xmin shadow domain point.
401   real_type getXmin(void) const { return Xmin; }
402
403   //! Get Ymin shadow domain point.
404   real_type getYmin(void) const { return Ymin; }
405
406   //! Get Xmax shadow domain point.
407   real_type getXmax(void) const { return Xmax; }
408
409   //! Get Ymax shadow domain point.
410   real_type getYmax(void) const { return Ymax; }
```

```
411
412   //! Clear the tire shadow domain.
413   void
414   clear(void) {
415     Xmin = quiteNaN;
416     Ymin = quiteNaN;
417     Xmax = quiteNaN;
418     Ymax = quiteNaN;
419   }
420
421   //! Print tire shadow bounding box vertices.
422   void
423   print( ostream_type & stream ) const {
424     stream
425       << "BBOX (xmin,ymin,xmax,ymax) = ( " << Xmin << ", " << Ymin
426       << ", " << Xmax << ", " << Ymax << " )\n";
427   }
428
429   //! Update the rectangular shadow domain of the tire in X and Y-axis
430   void
431   update(
432     ETRTO          const & TireGeometry, //!< ETRTO tire denomination object
433     ReferenceFrame const & RF            //!< ReferenceFrame object
434   );
435
436 private:
437   //! Update the mesh G2lib bounding boxes pointers vector.
438   void
439   updatePtrBBox(
440     std::vector<G2lib::BBox::PtrBBox> & PtrBBoxVec //!< Bounding boxes pointers
441     ) const {
442     PtrBBoxVec.resize(1);
443     PtrBBoxVec[0] = G2lib::BBox::PtrBBox(
444       new G2lib::BBox(Xmin, Ymin, Xmax, Ymax, 0, 0)
445     );
446   }
447 };
448
449 /*\
450  |         _                  _ _   _
451  |    __ _| | __ _  ___  _ __(_) |_| |__  _ __ ___  ___
452  |   / _` | |/ _` |/ _ \| '__| | __| '_ \| '_ ` _ \/ __|
453  |  | (_| | | (_| | (_) | |  | | |_| | | | | | | | \__ \
454  |   \__,_|_|\__, |\___/|_|  |_|\__|_| |_|_| |_| |_|___/
455  |           |___/
456 \*/
457
458 //! Algorithms for tire computations routine.
459 namespace algorithms {
460
461   //! Check if a point lays inside or outside a line segment.
462   //! Warning: The point query point must be on the same rect of the line
463   //! segment.
464   bool
465   intersectPointSegment(
466     vec2 const & Point_1, //!< Line segment point 1
467     vec2 const & Point_2, //!< Line segment point 2
468     vec2 const & Point    //!< Query point
469   );
470
471 }  // namespace algorithms
472
473 /*\
474  |   _____ _
475  |  |_   _(_)_ __ ___
```

```
476   |    | | | | '__/ _ \
477   |    | | | | | |  __/
478   |    |_| |_|_|  \___|
479   |
480   |          ^ Z
481   |          |
482   |        --|--
483   |      /   |   \
484   |     |  O=Y----|---> X
485   |      \       /
486   |        -----
487   |
488   |          ^ Y
489   |          |
490   |      ____|____
491   |     |    |    |
492   |     |  O=Z----|---> X
493   |     |_________|
494   |
495   |  O = OriginXZ
496   |  ISO Reference Frame!
497   \*/
498
499   //! Class for tire geometry and reference frame.
500   class Tire {
501   protected:
502     ETRTO          TireGeometry; //!< ETRTO tire denomination object
503     ReferenceFrame RF;           //!< ReferenceFrame object
504
505     Tire( Tire const & );                     //!< Copy constructor
506     Tire const & operator = ( Tire const & ); //!< Copy operator
507
508   public:
509     //! Variable set constructor for Tire object.
510     Tire(
511       ETRTO const & _TireGeometry
512     ) {
513       TireGeometry = _TireGeometry;
514     }
515
516     //! Set the Tire orientation.
517     void
518     setReferenceFrame( ReferenceFrame const & _RF )
519     { RF.set(_RF); }
520
521     //! Get Tire ReferenceFrame.
522     ReferenceFrame const &
523     getReferenceFrame(void) const
524     { return RF; }
525
526     //! Set camber angle.
527     void
528     setOrigin( vec3 const & Origin )
529     { RF.setOrigin(Origin); }
530
531     //! Set rotation matrix.
532     void
533     setRotationMatrix( mat3 const & RotationMatrix )
534     { RF.setRotationMatrix(RotationMatrix); }
535
536     //! Set total transformation matrix.
537     void
538     setTotalTransformationMatrix( mat4 const & TM )
539     { RF.setTotalTransformationMatrix(TM); }
540
```

```
541   //! Get current Euler angles [rad] for X-axis.
542   //! Warning: Factor as Rz*Rx*Ry!
543   real_type getEulerAngleX(void) const { return RF.getEulerAngleX(); };
544
545   //! Get current Euler angles [rad] for Y-axis.
546   //! Warning: Factor as Rz*Rx*Ry!
547   real_type getEulerAngleY(void) const { return RF.getEulerAngleY(); };
548
549   //! Get current Euler angles [rad] for Z-axis.
550   //! Warning: Factor as Rz*Rx*Ry!
551   real_type getEulerAngleZ(void) const { return RF.getEulerAngleZ(); };
552 };
553
554 /*\
555  |   __  __            _      _____                          _
556  |  |  \/  | __ _  __ _(_) ___|  ___|__  _ __ _ __ ___  _   _| | __ _
557  |  | |\/| |/ _` |/ _` | |/ __| |_ / _ \| '__| '_ ` _ \| | | | |/ _` |
558  |  | |  | | (_| | (_| | | (__|  _| (_) | |  | | | | | | |_| | | (_| |
559  |  |_|  |_|\__,_|\__, |_|\___|_|  \___/|_|  |_| |_| |_|\__,_|_|\__,_|
560  |               |___/
561 \*/
562
563 //! Class for evaluating the contact patch for Magic Formla.
564 class MagicFormula : public Tire {
565 protected:
566   Disk      SingleDisk;      //!< Single Disk
567   vec3      ContactNormal;   //!< Contact normal direction
568   vec3      ContactPoint;    //!< Contact point
569   real_type ContactFriction; //!< Contact point friction
570   real_type RelativeCamber;  //!< Relative camber angle
571
572   MagicFormula( MagicFormula const & );                     //!< Copy constructor
573   MagicFormula const & operator = ( MagicFormula const & ); //!< Copy operator
574
575 public:
576   //! Variable set constructor for MagicFormula object.
577   MagicFormula(
578     ETRTO const & _TireGeometry
579   ) : Tire( _TireGeometry ) {
580     SingleDisk.set( Disk( vec2(0.0, 0.0), 0.0, TireGeometry.getTireRadius() ) );
581   }
582
583   //! Get contact point normal direction.
584   vec3 const &
585   getContactNormal(void) const
586   { return ContactNormal; }
587
588   //! Get contact point.
589   vec3 const &
590   getContactPoint(void) const
591   { return ContactPoint; }
592
593       //! Get contact point.
594   real_type
595   getContactFriction(void) const
596   { return ContactFriction; }
597
598   //! Get contact point reference frame with 4x4 transformation matrix.
599   mat4
600   getContactPointRF(void) const;
601
602   //! Get relative camber angle [rad].
603   real_type
604   getRelativeCamber(void) const
605   { return RelativeCamber; }
```

```
606
607   //! Get contact depth at center point (if negative the tire
608   //! does not touch the ground) [m].
609   real_type
610   getContactDepth(void) const
611   { return TireGeometry.getTireRadius()-(RF.getOrigin()-ContactPoint).norm(); }
612
613   //! Get approximed contact area [m^2] with triangles intersection.
614   real_type
615   getContactArea(
616     RDF::TriangleRoad_list const & intersectionTriPtr //!< Local intersected triangles vector
617   ) const
618   {
619     return SingleDisk.getPatchLength( intersectionTriPtr, RF ) * TireGeometry.getSectionWidth();
620   }
621
622   //! Get approximed contact area [m^2].
623   real_type
624   getContactArea(void) const
625   {
626     return SingleDisk.getPatchLength( ContactNormal, ContactPoint, RF ) * TireGeometry.getSectionWidth();
627   }
628
629   //! Get approximed contact volume [m^3] with triangles intersection.
630   real_type
631   getContactVolume(
632     RDF::TriangleRoad_list const & intersectionTriPtr // Local intersected triangles vector
633   ) const;
634
635   //! Get approximed contact volume [m^3].
636   real_type
637   getContactVolume(void) const;
638
639   //! Update the current position of the tire and find parameters
640   //! for the contact with precomputed local plane.
641   void
642   setup(
643     vec3 const & _ContactNormal, //!< Contact point normal direction
644     vec3 const & _ContactPoint,  //!< Contact point
645     mat4 const & _TM             //!< Trasnformation matrix
646   ) {
647     // Set the new reference frame
648     RF.setTotalTransformationMatrix(_TM);
649     // Update class members
650     ContactNormal = _ContactNormal;
651     ContactPoint  = _ContactPoint;
652   }
653
654   //! Update the current position of the tire and find all parameters
655   //! for the contact.
656   void
657   setup(
658     RDF::MeshSurface       & Mesh, //!< MeshSurface object
659     mat4             const & TM,   //!< Trasnformation matrix
660     bool                     print = false
661   );
662
663 protected:
664   //! Perform one point sampling (return intersected triangle friction).
665   real_type
666   pointSampling(
667   RDF::TriangleRoad_list const & intersectionTriPtr, //!< Intersected triangles vector
668   vec3                   const & RayOrigin,          //!< Ray origin
669   vec3                   const & RayDirection,       //!< Ray direction
670   vec3                         & SampledPt           //!< Intersection point
```

```
671     ) const;
672
673     //! Perform triangles sampling on 4 points at 0.1*R0 along X and 0.3*W along Y.
674     void
675     fourPointsSampling(
676     RDF::TriangleRoad_list const & intersectionTriPtr, //!< Intersected triangles vector
677     row_vec3                   & SampledPtsVec       //!< Intersection points vector
678     );
679
680     //! Calculate the normal vector of the local track plane and local contact point.
681     void
682     calculateLocalRoadPlane(
683       RDF::TriangleRoad_list const & intersectionTriPtr //!< Intersected triangles vector
684     );
685
686     //! Calculate the relative camber angle [rad].
687     void
688     calculateRelativeCamber(
689       row_vec3 const & Qvec //!< Intersection points vector
690     );
691   };
692
693 }  // namespace PatchTire
694
695 ///
696 /// eof: PatchTire.hh
697 ///
```

B.5 PatchTire.cc

```
 1 #include "PatchTire.hh"        // RDF file extention Loader
 2 // #include "PolynomialRoots.hh"  // Quartic roots Flocke library
 3
 4 //! Tire computations routine
 5 namespace PatchTire {
 6
 7   using namespace TireGround;
 8
 9   /*\
10    |   ____  _     _
11    |  |  _ \(_)___| | __
12    |  | | | | / __| |/ /
13    |  | |_| | \__ \   <
14    |  |____/|_|___/_|\_\
15   \*/
16
17   //! Check if a point lays inside or outside a circumfererence.
18   //! If output bool is true the point is inside the circumfererence,
19   //! otherwise it is outside.
20   bool
21   Disk::isPointInside(
22     vec2 const & Point
23   ) const {
24     // Compare radius with distance of disk center from given point
25     vec2 PO( Point - OriginXZ );
26     return PO.dot(PO) <= Radius*Radius;
27   }
28
29   //! Find the intersection points between a circle and a line segment.
30   //! Output integer gives the number of intersection points.
31   int_type
32   Disk::intersectSegment(
33     vec2      const & Point_1,     //!< Line segment point 1
```

```
34    vec2      const & Point_2,     //!< Line segment point 2
35    vec2            & Intersect_1, //!< Intersection point 2
36    vec2            & Intersect_2  //!< Intersection point 2
37  ) const {
38    real_type t_param;
39    vec2      d( Point_2 - Point_1 );
40    vec2      P10( Point_1 - OriginXZ );
41    real_type A   = d.dot(d);
42    real_type B   = 2 * d.dot(P10);
43    real_type C   = P10.dot(P10) - Radius*Radius;
44    real_type det = B*B - 4 * A * C;
45    if ( A <= epsilon || det < 0 ) {
46      // No real solutions
47      Intersect_1 = vec2(quiteNaN, quiteNaN);
48      Intersect_2 = vec2(quiteNaN, quiteNaN);
49      return 0;
50    } else if ( det == 0.0 ) {
51      // One solution
52      t_param     = -B / (2*A);
53      Intersect_1 = Point_1 + t_param * d;
54      Intersect_2 = vec2(quiteNaN, quiteNaN);
55      return 1;
56    } else {
57      // Two solutions
58      t_param = (-B + std::sqrt(det)) / (2 * A);
59      Intersect_1 = Point_1 + t_param * d;
60      t_param = (-B - std::sqrt(det)) / (2 * A);
61      Intersect_2 = Point_1 + t_param * d;
62      return 2;
63    }
64  }
65
66  //! Check if two plane intersects and find the intersecting rect.
67  bool
68  Disk::intersectPlane(
69    vec3 const & Plane_Normal,
70    vec3 const & Plane_Point,
71    vec3       & Line_Direction,
72    vec3       & Line_Point
73  ) const {
74    // Plane(Point,Normal) and Disk intersection -> Parametric rect
75    vec3 Disk_Point( getOriginXYZ() );
76    vec3 Disk_Normal( 0.0, 1.0, 0.0 );
77    // Rect direction
78    Line_Direction = Plane_Normal.cross(Disk_Normal);
79    // If the two plane are parallel they do not intersects
80    if ( Line_Direction.norm() > epsilon) {
81      // Given the plane ax+by+cz=d
82      real_type d_Disk = Disk_Point.dot(-Disk_Normal);
83      real_type d_Plane = Plane_Point.dot(-Plane_Normal);
84      // Find a point on the line, which is also on both planes
85      // choose simplest plane where d=0: ax + by + cz = 0
86      vec3 u1(  d_Disk * Plane_Normal );
87      vec3 u2( -d_Plane * Disk_Normal );
88      Line_Point = (u1 + u2).cross(Line_Direction) / Line_Direction.dot(Line_Direction); // In absolute
            reference frame return
89      return true;
90    } else {
91      return false;
92    }
93  }
94
95  //! Get the contact patch length inside the single disk of a segment described by
96  //! the intersection of triangles on XZ plane.
97  real_type
```

```
 98 Disk::getPatchLength(
 99   RDF::TriangleRoad_list const & intersectionTriPtr, //!< Local intersected triangles
100   ReferenceFrame         const & RF                   //!< ReferenceFrame object
101 ) const {
102   // Disk point and vector in absolute reference frame
103   vec3 Disk_Point( RF.getOrigin() + RF.getRotationMatrix()*getOriginXYZ() );
104   vec3 Disk_Normal( RF.getY() );
105   real_type PatchLength = 0.0;
106   std::vector<vec3> IntersectionPts;
107   for ( unsigned i = 0; i < intersectionTriPtr.size(); ++i ) {
108     if( (*intersectionTriPtr[i]).intersectPlane(Disk_Normal, Disk_Point, IntersectionPts) ) {
109       // Transform in disk relative reference frame
110       vec3 P1_rel( RF.getRotationMatrixInverse(IntersectionPts[0] - RF.getOrigin()) );
111       vec3 P2_rel( RF.getRotationMatrixInverse(IntersectionPts[1] - RF.getOrigin()) );
112
113       // Transfer only the XZ part (Y part must be = to OffsetY, so useless)
114       PatchLength += getPatchLength( vec2(P1_rel.x(),P1_rel.z()), vec2(P2_rel.x(),P2_rel.z()), RF);
115     }
116   }
117   return PatchLength;
118 }
119
120 //! Get the contact patch length inside the single disk of a segment described by
121 //! points PointXZ_1 and PointXZ_2 on XZ plane.
122 // Plane (P-P0).N = 0 con N rivolto verso alto
123 // http://www.songho.ca/math/plane/plane.html
124
125 real_type
126 Disk::getPatchLength(
127     vec3           const & Plane_Normal,
128     vec3           const & Plane_Point,
129     ReferenceFrame const & RF           //!< New ReferenceFrame object
130 ) const {
131   // Change reference frame for local road plane
132   vec3 Plane_Normal_rel( RF.getRotationMatrixInverse(Plane_Normal) ) ;
133   vec3 Plane_Point_rel( RF.getRotationMatrixInverse(Plane_Point - RF.getOrigin()) );
134   //! Check if two plane intersects and find the intersecting rect.
135   vec3 P, T;
136   if(intersectPlane( Plane_Normal_rel, Plane_Point_rel, T, P)){
137     // Make a segment on the intersection (on relative Disk rerence frame)
138     vec3 P1( P - 200.0*Radius*T );
139     vec3 P2( P + 200.0*Radius*T );
140
141     return getPatchLength(vec2(P1.x(),P1.z()), vec2(P2.x(),P2.z()), RF);
142   } else {
143     RDF_ERROR("Cannot handle planes intersection");
144     return quiteNaN;
145   }
146 }
147
148 //! Get the contact patch length inside the single disk of a segment described by
149 //! points PointXZ_1 and PointXZ_2 on XZ plane.
150 real_type
151 Disk::getPatchLength(
152   vec2           const & PointXZ_1,
153   vec2           const & PointXZ_2,
154   ReferenceFrame const & RF           //!< New ReferenceFrame object
155 ) const {
156   vec2 Intersection_1, Intersection_2;
157   int_type Type = this->intersectSegment(
158     PointXZ_1, PointXZ_2, Intersection_1, Intersection_2
159   );
160
161   if ( Type == 0 ) {
162     // No contact points, the line segment is not into the Disk
```

```
163       return 0.0;
164     } else if (Type == 1) {
165       // Tangent, no length added
166       return 0.0;
167     } else if (Type == 2) {
168       // Check whether the two segment points are into the circle
169       bool Pose_pt1 = this->isPointInside( PointXZ_1 );
170       bool Pose_pt2 = this->isPointInside( PointXZ_2 );
171       // Check whether the two intersection points are onto the line segment
172       bool Pose_int1 = algorithms::intersectPointSegment(PointXZ_1, PointXZ_2, Intersection_1);
173       bool Pose_int2 = algorithms::intersectPointSegment(PointXZ_1, PointXZ_2, Intersection_2);
174
175       // Cases
176       // Line segment PointXZ_1 and line segment PointXZ_2 outside the circle,
177       // intersection points oustide line segment
178       if ( !Pose_pt1 && !Pose_pt2 && !Pose_int1 && !Pose_int2 ) {
179         return 0.0;
180       }
181       // Line segment PointXZ_1 and line segment PointXZ_2 into the circle,
182       // intersection points oustide line segment
183       if ( Pose_pt1 && Pose_pt2 && !Pose_int1 && !Pose_int2 ) {
184         return (PointXZ_2 - PointXZ_1).norm();
185       }
186       // Intersection points into the line segment and line segment points
187       // outside the circle
188       else if ( !Pose_pt1 && !Pose_pt2 && Pose_int1 && Pose_int2 ) {
189         return (Intersection_2 - Intersection_1).norm();
190       }
191       // Line segment Point_1 outside the circle, line segment Point_2
192       // inside the circle
193       else if ( !Pose_pt1 && Pose_pt2 ) {
194         if ( Pose_int1 && !Pose_int2 ) {
195           // Add length from Intersection_1 to Point_2
196           return (Intersection_1 - PointXZ_2).norm();
197         } else if ( Pose_int2 && !Pose_int1 ) {
198           // Add length from Intersection_2 to Point_2
199           return (Intersection_2 - PointXZ_2).norm();
200         }
201       }
202       // Line segment Point_1 inside the circle, line segment Point_2
203       // outside the circle
204       else if ( Pose_pt1 && !Pose_pt2 ) {
205         if ( Pose_int1 && !Pose_int2 ) {
206           // Add length from Intersection_1 to Point_1
207           return (Intersection_1 - PointXZ_1).norm();
208         } else if ( !Pose_int1 && Pose_int2 ) {
209           // Add length from Intersection_2 to Point_1
210           return (Intersection_2 - PointXZ_1).norm();
211         }
212       }
213     }
214     RDF_ERROR("Invalid intersection on Disk::getPatchLength");
215     return quiteNaN;
216   }
217
218   /*\
219    |   ____      __                                 _____
220    |  |  _ \ ___ / _| ___ _ __ ___ _ __   ___ ___|  ___| __ __ _ _ __ ___   ___
221    |  | |_) / _ \ |_ / _ \ '__/ _ \ '_ \ / __/ _ \ |_  | '__/ _` | '_ ` _ \ / _ \
222    |  |  _ <  __/  _|  __/ | |  __/ | | | (_|  __/  _| | | | (_| | | | | | |  __/
223    |  |_| \_\___|_|  \___|_|  \___|_| |_|\___\___|_|   |_|  \__,_|_| |_| |_|\___|
224    |
225   \*/
226
227   //! Get current Euler angles [rad] for X-axis.
```

```
228 //! Warning: Factor as Rz*Rx*Ry!
229 // https://www.geometrictools.com/Documentation/EulerAngles.pdf
230 real_type ReferenceFrame::getEulerAngleX(void) const {
231   real_type r21 = RotationMatrix(2, 1);
232   if (r21 < 1.0) {
233     if (r21 > -1.0) {
234       return std::asin(r21);
235     } else {  // r21 == -1.0
236       // Not a unique solution : thetaY - thetaZ = atan2( r02 , r00 )
237       return -G2lib::m_pi / 2.0;
238     }
239   } else {  // r21 == 1.0
240     // Not a unique solution : thetaY + thetaZ = atan2( r02 , r00 )
241     return G2lib::m_pi / 2.0;
242   }
243 }
244
245 //! Get current Euler angles [rad] for Y-axis.
246 //! Warning: Factor as Rz*Rx*Ry!
247 // https://www.geometrictools.com/Documentation/EulerAngles.pdf
248 real_type ReferenceFrame::getEulerAngleY(void) const {
249   real_type r20 = RotationMatrix(2, 0);
250   real_type r21 = RotationMatrix(2, 1);
251   real_type r22 = RotationMatrix(2, 2);
252   if (r21 < 1.0) {
253     if (r21 > -1.0) {
254       return std::atan2(-r20, r22);
255     } else {  // r21 == -1.0
256       // Not a unique solution : thetaY - thetaZ = atan2( r02 , r00 )
257       return 0.0;
258     }
259   } else {  // r21 == 1.0
260     // Not a unique solution : thetaY + thetaZ = atan2( r02 , r00 )
261     return 0.0;
262   }
263 }
264
265 //! Get current Euler angles [rad] for Z-axis.
266 //! Warning: Factor as Rz*Rx*Ry!
267 // https://www.geometrictools.com/Documentation/EulerAngles.pdf
268 real_type ReferenceFrame::getEulerAngleZ(void) const {
269   real_type r00 = RotationMatrix(0, 0);
270   real_type r01 = RotationMatrix(0, 1);
271   real_type r02 = RotationMatrix(0, 2);
272   real_type r11 = RotationMatrix(1, 1);
273   real_type r21 = RotationMatrix(2, 1);
274   if (r21 < 1.0) {
275     if (r21 > -1.0) {
276       return std::atan2(-r01, r11);
277     } else {  // r21 == -1.0
278       // Not a unique solution : thetaY - thetaZ = atan2( r02 , r00 )
279       return -std::atan2(r02, r00);
280     }
281   } else {  // r21 == 1.0
282     // Not a unique solution : thetaY + thetaZ = atan2( r02 , r00 )
283     return std::atan2(r02, r00);
284   }
285 }
286
287 /*\
288 |   ____  _               _
289 |  / ___|| |__   __ _  __| | _____      __
290 |  \___ \| '_ \ / _` |/ _` |/ _ \ \ /\ / /
291 |   ___) | | | | (_| | (_| | (_) \ V  V /
292 |  |____/|_| |_|\__,_|\__,_|\___/ \_/\_/
```

```
293 \*/
294
295  //! Update the rectangular shadow domain of the tire in X and Y-axis
296  void Shadow::update(
297      ETRTO const & TireGeometry,  //!< ETRTO tire denomination object
298      ReferenceFrame const & RF    //!< ReferenceFrame object
299  ) {
300    // Calculate maximum covered space
301    real_type diagonal =
302        hypot(TireGeometry.getSectionWidth(), TireGeometry.getTireDiameter()) /
303        2;
304
305    // Increment shadow to take in account camber angle
306    real_type inc = 1.1;
307
308    // Set new tire shadow domain
309    this->Xmax = RF.getOrigin()[0] + inc * diagonal;
310    this->Ymax = RF.getOrigin()[1] + inc * diagonal;
311    this->Xmin = RF.getOrigin()[0] - inc * diagonal;
312    this->Ymin = RF.getOrigin()[1] - inc * diagonal;
313  }
314
315  /*\
316   |         _                  _ _   _
317   |    __ _| | __ _  ___  _ __(_) |_| |__  _ __ ___  ___
318   |   / _` | |/ _` |/ _ \| '__| | __| '_ \| '_ ` _ \/ __|
319   |  | (_| | | (_| | (_) | |  | | |_| | | | | | | | \__ \
320   |   \__,_|_|\__, |\___/|_|  |_|\__|_| |_|_| |_| |_|___/
321   |           |___/
322  \*/
323
324  //! Algorithms for tire computations routine.
325  namespace algorithms {
326
327    //! Check if a point lays inside or outside a line segment.
328    //! Warning: The point query point must be on the same rect of the line
329    //! segment.
330    bool
331    intersectPointSegment(
332      vec2 const & Point_1,
333      vec2 const & Point_2,
334      vec2 const & Point
335    ) {
336      // A and B are the extremities of the current segment C is the point to
337      // check
338
339      // Create the vector AB
340      vec2 AB( Point_2 - Point_1 );
341      // Create the vector AC
342      vec2 AC( Point - Point_1 );
343
344      // Compute the cross product of AB and AC
345      // Check if the three points are aligned (cross product is null)
346      real_type z = AB.x() * AC.y() - AC.x() * AB.y();
347      if (z*z > epsilon) {
348        RDF_ERROR("Cannot handle point-segment intersection!");
349        return false;
350      }
351
352      // Compute the dot product of vectors
353      real_type KAC = AB.dot(AC);
354      if ( KAC      < -epsilon ) return false;
355      if ( abs(KAC) < epsilon  ) return true;
356
357      // Compute the square of the segment lenght
```

```
358     real_type KAB = AB.dot(AB);
359     if ( KAC > KAB ) return false;
360     if ( abs(KAC - KAB) < epsilon ) return true;
361
362     // The point is on the segment
363     return true;
364   }
365
366 }  // namespace algorithms
367
368 /*\
369  |   __  __             _      _____                          _
370  |  |  \/  | __ _  __ _(_) ___|  ___|__  _ __ _ __ ___  _   _| | __ _
371  |  | |\/| |/ _` |/ _` | |/ __| |_ / _ \| '__| '_ ` _ \| | | | |/ _` |
372  |  | |  | | (_| | (_| | | (__|  _| (_) | |  | | | | | | |_| | | (_| |
373  |  |_|  |_|\__,_|\__, |_|\___|_|  \___/|_|  |_| |_| |_|\__,_|_|\__,_|
374  |                |___/
375 \*/
376
377 //! Get contact point reference frame with 4x4 transformation matrix.
378 mat4
379 MagicFormula::getContactPointRF(void) const {
380   vec3 X_versor = RF.getY().cross(ContactNormal).normalized();
381   vec3 Y_versor = ContactNormal.cross(X_versor);
382   mat4 TM;
383   TM << X_versor, Y_versor, ContactNormal, ContactPoint,
384       vec4(0.0, 0.0, 0.0, 1.0).transpose();
385   return TM;
386 }
387
388 //! Get approximed contact volume [m^3].
389 real_type
390 MagicFormula::getContactVolume(void)
391 const {
392   real_type R = TireGeometry.getTireRadius();
393   real_type c = SingleDisk.getPatchLength( ContactNormal, ContactPoint, RF );
394   real_type c_2R = c / (2*R);
395   return R*R * ( std::asin(c_2R) - c_2R*std::sqrt(1-c_2R*c_2R)) * TireGeometry.getSectionWidth();
396 }
397
398 //! Update the current position of the tire and find all parameters
399 //! for the contact.
400 void
401 MagicFormula::setup(
402   RDF::MeshSurface       & Mesh, //!< MeshSurface object
403   mat4             const & TM,   //!< Trasnformation matrix
404   bool                     print
405 ) {
406   // Set the new reference frame
407   RF.setTotalTransformationMatrix(TM);
408   // Shadow bounding box object
409   Shadow TireShadow(TireGeometry, RF);
410   // Local intersected triangles vector
411   RDF::TriangleRoad_list intersectionTriPtr =
412       Mesh.updateIntersection(TireShadow.getAABBPtr());
413   // Update the local road plane (normal, point and realtive camber)
414   calculateLocalRoadPlane(intersectionTriPtr);
415
416   // Print the results
417   if (print) {
418     std::cout << "MAGIC FORMULA CONTACT PARAMETERS\n"
419               << "\tYaw angle\n"
420               << "\ t = " << getEulerAngleZ() / G2lib::m_pi << "pi rad\n"
421               << "\tRotation angle\n"
422               << "\tß = " << getEulerAngleY() / G2lib::m_pi << "pi rad\n"
```

```
423               << "\tCamber angle\n"
424               << "\Γt = " << getEulerAngleX() / G2lib::m_pi << "pi rad\n"
425               << "\tNormal contact point vector of the local track plane "
426                  "(absolute reference frame)\n"
427               << "\tN = [ " << ContactNormal.x() << ", " << ContactNormal.y()
428               << ", " << ContactNormal.z() << " ]\n"
429               << "\tLocal contact point (absolute reference frame)\n"
430               << "\tP = [ " << ContactPoint.x() << ", " << ContactPoint.y()
431               << ", " << ContactPoint.z() << " ]\n"
432               << "\tLocal contact point reference frame\n"
433               << getContactPointRF() << "\n"
434               << "\tRelative camber angle\n"
435               << "\tß = " << getRelativeCamber() / G2lib::m_pi << "pi rad\n"
436               << "\tLocal contact point friction\n"
437               << "\tf = " << getContactFriction() << "\n"
438               << "\tLocal contact depth (on center point)\n"
439               << "\tD = " << getContactDepth() << " m\n"
440               << "\tLocal approximated contact area (calculated with local plane)\n"
441               << "\tA = " << getContactArea() << " m^2\n"
442               << "\tLocal approximated contact area (calculated with triangles intersection)\n"
443               << "\tA = " << getContactArea(intersectionTriPtr) << " m^2\n"
444               << "\tLocal approximated contact volume (calculated with local plane)\n"
445               << "\tV = " << getContactVolume() << " m^3\n\n";
446   }
447 }
448
449 //! Perform one point sampling (return intersected triangle friction).
450 real_type
451 MagicFormula::pointSampling(
452     RDF::TriangleRoad_list const &
453         intersectionTriPtr,     //!< Intersected triangles vector
454     vec3 const & RayOrigin,     //!< Ray origin
455     vec3 const & RayDirection,  //!< Ray direction
456     vec3 & SampledPt            //!< Intersection point
457     ) const {
458   vec3 IntersectionPoint;
459   real_type TriangleFriction;
460   std::vector<vec3> IntersectionPointVec;
461   std::vector<real_type> TriangleFrictionVec;
462   for (unsigned t = 0; t < intersectionTriPtr.size(); ++t) {
463     if ( (*intersectionTriPtr[t]).intersectRay(
464             RayOrigin, RayDirection, IntersectionPoint) ) {
465       // Store results
466       IntersectionPointVec.push_back(IntersectionPoint);
467       TriangleFrictionVec.push_back((*intersectionTriPtr[t]).getFriction());
468     }
469   }
470   // Select the highest intersection point
471   if (IntersectionPointVec.size() > 1) {
472     SampledPt        = IntersectionPointVec[0];
473     TriangleFriction = TriangleFrictionVec[0];
474     for (unsigned j = 1; j < IntersectionPointVec.size(); ++j) {
475       if (IntersectionPointVec[j][2] > SampledPt[2]) {
476         SampledPt        = IntersectionPointVec[j];
477         TriangleFriction = TriangleFrictionVec[j];
478       }
479     }
480     return TriangleFriction;
481   } else if (IntersectionPointVec.size() == 0) {
482     RDF_ERROR("There is no terrain under the tire!");
483   } else {  // j == 1
484     SampledPt = IntersectionPointVec[0];
485     return TriangleFrictionVec[0];
486   }
487 }
```

```
488
489 void
490 MagicFormula::fourPointsSampling(
491     RDF::TriangleRoad_list const &
492         intersectionTriPtr,   //!< Intersected triangles vector
493     row_vec3 & SampledPtsVec  //!< Intersection points vector
494 ) {
495   // Calculate Delta_X and Delta_Y
496   real_type Delta_X = 0.1 * TireGeometry.getTireRadius();
497   real_type Delta_Y = 0.3 * TireGeometry.getSectionWidth();
498   // Store the four sample positions
499   row_vec3 Qpos(4);
500   Qpos[0] = RF.getOrigin() + RF.getRotationMatrix() * vec3(Delta_X, 0.0, 0.0);
501   Qpos[1] = RF.getOrigin() - RF.getRotationMatrix() * vec3(Delta_X, 0.0, 0.0);
502   Qpos[2] = RF.getOrigin() + RF.getRotationMatrix() * vec3(0.0, Delta_Y, 0.0);
503   Qpos[3] = RF.getOrigin() - RF.getRotationMatrix() * vec3(0.0, Delta_Y, 0.0);
504   // Find intersection in the four positions
505   real_type Friction = 0.0;
506   for (unsigned i = 0; i < 4; ++i) {
507     Friction += pointSampling(intersectionTriPtr, Qpos[i], -RF.getZ(),
508                               SampledPtsVec[i]);
509   }
510   ContactFriction = Friction / 4.0;
511 }
512
513 //! Find the normal vector of the local track plane and local contact point.
514 void
515 MagicFormula::calculateLocalRoadPlane(
516     RDF::TriangleRoad_list const &
517         intersectionTriPtr  //!< Intersected triangles vector
518 ) {
519   // Check if there is an orientation
520   if (RF.isEmpty()) {
521     RDF_ERROR("Tire has no reference frame associated!");
522     ContactNormal = vec3_NaN;
523     ContactPoint  = vec3_NaN;
524   }
525   // Perform the terrain sampling;
526   row_vec3 Qvec(4);
527   fourPointsSampling(intersectionTriPtr, Qvec);
528   // Calculate normal of the local track plane
529   ContactNormal = ((Qvec[0] - Qvec[1]).cross(Qvec[2] - Qvec[3])).normalized();
530   // Calculate first guess of local contact point
531   vec3 P_star(0.0, 0.0, 0.0);
532   for (unsigned i = 0; i < 4; ++i)
533     P_star += Qvec[i];
534   P_star /= 4;
535   // Calculate real local contact point
536   real_type dist = (RF.getOrigin() - P_star).dot(ContactNormal);
537   ContactPoint   = RF.getOrigin() - ContactNormal * dist;
538   // Update relative camber
539   calculateRelativeCamber(Qvec);
540 }
541
542 //! Calculate the relative camber angle [rad].
543 void
544 MagicFormula::calculateRelativeCamber(
545   row_vec3 const & Qvec //!< Intersection points vector
546 ) {
547   vec3 Q3( RF.getRotationMatrixInverse(Qvec[2]-RF.getOrigin()) );
548   vec3 Q4( RF.getRotationMatrixInverse(Qvec[3]-RF.getOrigin()) );
549   RelativeCamber = - std::atan2(Q3.z()-Q4.z(), Q3.y()-Q4.y()); // - per regola della mano destra
550 }
551
552 }  // namespace PatchTire
```

Codice dei Tests C

C.1 Tests Geometrici

C.1.1 Geometry-test1.cc

```
1 // GEOMETRY TEST 1 - RAY/TRIANGLE INTERSECTION ON TRIANGLE EDGE
2
3 #include <fstream>   // fStream - STD File I/O Library
4 #include <iostream>  // Iostream - STD I/O Library
5 #include <string>    // String - STD String Library
6
7 #include "PatchTire.hh"  // Tire Data Processing
8 #include "RoadRDF.hh"    // Tire Data Processing
9
10 // Main function
11 int
12 main() {
13   std::cout
14     << " GEOMETRY TEST 1 - RAY/TRIANGLE INTERSECTION ON TRIANGLE EDGE\n"
15     << "Angle\tIntersections\n";
16
17   RDF::vec3 V1[3];
18   V1[0] = RDF::vec3(1.0, 0.0, 0.0);
19   V1[1] = RDF::vec3(0.0, 1.0, 0.0);
20   V1[2] = RDF::vec3(-1.0, 0.0, 0.0);
21
22   RDF::vec3 V2[3];
23   V2[0] = RDF::vec3(-1.0, 0.0, 0.0);
24   V2[1] = RDF::vec3(0.0, -1.0, 0.0);
25   V2[2] = RDF::vec3(1.0, 0.0, 0.0);
26
27   // Initialize generic Triangle3D
28   RDF::TriangleRoad Triangle1(V1, 0.0);
29   RDF::TriangleRoad Triangle2(V2, 0.0);
30
31   // Initialize rotation matrix
32   RDF::mat3 Rot_X;
33
```

```
34  // Initialize intersection point
35  RDF::vec3 IntersectionPointTri1, IntersectionPointTri2;
36  bool IntersectionBoolTri1, IntersectionBoolTri2;
37
38  // Initialize Ray
39  RDF::vec3 RayOrigin    = RDF::vec3(0.0, 0.0, 0.0);
40  RDF::vec3 RayDirection = RDF::vec3(0.0, 0.0, -1.0);
41
42  // Perform intersection at 0.5° step
43  for ( RDF::real_type angle = 0;
44        angle < G2lib::m_pi;
45        angle += G2lib::m_pi / 360.0 ) {
46
47    Rot_X << 1,          0,           0,
48             0, cos(angle), -sin(angle),
49             0, sin(angle),  cos(angle);
50
51    // Initialize vertices
52    RDF::vec3 VerticesTri1[3], VerticesTri2[3];
53
54    VerticesTri1[0] = Rot_X * V1[0];
55    VerticesTri1[1] = Rot_X * V1[1];
56    VerticesTri1[2] = Rot_X * V1[2];
57
58    VerticesTri2[0] = Rot_X * V2[0];
59    VerticesTri2[1] = Rot_X * V2[1];
60    VerticesTri2[2] = Rot_X * V2[2];
61
62    Triangle1.setVertices(VerticesTri1);
63    Triangle2.setVertices(VerticesTri2);
64
65    IntersectionBoolTri1 = Triangle1.intersectRay(
66      RayOrigin, RayDirection, IntersectionPointTri1
67    );
68    IntersectionBoolTri2 = Triangle2.intersectRay(
69      RayOrigin, RayDirection, IntersectionPointTri2
70    );
71
72    std::cout
73      << angle * 180.0 / G2lib::m_pi << "°\t"
74      << "T1 -> " << IntersectionBoolTri1 << ", T2 -> "
75      << IntersectionBoolTri2 << std::endl;
76
77    // ERROR if no one of the two triangles is hit
78    if ( !IntersectionBoolTri1 && !IntersectionBoolTri2 ) {
79      std::cout << "GEOMETRY TEST 1: Failed\n";
80      break;
81    }
82  }
83
84  // Print triangle normal vector
85  RDF::vec3 N1 = Triangle1.Normal();
86  RDF::vec3 N2 = Triangle2.Normal();
87  std::cout
88    << "\nTriangle 1 face normal = [" << N1[0] << ", " << N1[1] << ", " << N1[2] << "]"
89    << "\nTriangle 2 face normal = [" << N2[0] << ", " << N2[1] << ", " << N2[2] << "]"
90    << "\n\n\nGEOMETRY TEST 1: Completed\n";
91
92  // Exit the program
93  return 0;
94 }
```

C.1.2 Geometry-test2.cc

```
// GEOMETRY TEST 2 - SEGMENT CIRCLE INTERSECTION

#include <fstream>   // fStream - STD File I/O Library
#include <iostream>  // Iostream - STD I/O Library
#include <string>    // String - STD String Library

#include "PatchTire.hh"  // Tire Data Processing
#include "RoadRDF.hh"    // Tire Data Processing

// Main function
int
main() {
  // Initialize disk
  PatchTire::Disk NewDisk(RDF::vec2(0.0, 0.0), 0.0, 1.0);

  // Initialize segments points
  RDF::vec2 SegIn1PtA = RDF::vec2(0.0, 0.0);
  RDF::vec2 SegIn1PtB = RDF::vec2(0.0, 1.0);

  RDF::vec2 SegIn2PtA = RDF::vec2(-2.0, 0.0);
  RDF::vec2 SegIn2PtB = RDF::vec2(2.0, 0.0);

  RDF::vec2 SegOutPtA = RDF::vec2(1.0, 2.0);
  RDF::vec2 SegOutPtB = RDF::vec2(-1.0, 2.0);

  RDF::vec2 SegTangPtA = RDF::vec2(1.0, 1.0);
  RDF::vec2 SegTangPtB = RDF::vec2(-1.0, 1.0);

  // Initialize intersection points and output types
  RDF::vec2 IntSegIn1_1, IntSegIn1_2, IntSegIn2_1, IntSegIn2_2, IntSegOut_1,
      IntSegOut_2, IntSegTang_1, IntSegTang_2;
  RDF::int_type PtIn1, PtIn2, PtOut, PtTang;

  // Calculate intersections
  PtIn1 = NewDisk.intersectSegment(
    SegIn1PtA, SegIn1PtB, IntSegIn1_1, IntSegIn1_2
  );
  PtIn2 = NewDisk.intersectSegment(
    SegIn2PtA, SegIn2PtB, IntSegIn2_1, IntSegIn2_2
  );
  PtOut = NewDisk.intersectSegment(
    SegOutPtA, SegOutPtB, IntSegOut_1, IntSegOut_2
  );
  PtTang = NewDisk.intersectSegment(
    SegTangPtA, SegTangPtB, IntSegTang_1, IntSegTang_2
  );

  // Diplay results
  std::cout
    << "GEOMETRY TEST 2 - SEGMENT DISK INTERSECTION\n\n"
    << "Radius = " << NewDisk.getRadius() << std::endl
    << "Origin = [" << NewDisk.getOriginXZ().x() << ", " << NewDisk.getOriginXZ().y() << "]\n"
    << std::endl
    << "Segment 1 with two intersections -> " << PtIn1 << " intersections found\n"
    << "Segment Point A\t= [" << SegIn1PtA.x() << ", " << SegIn1PtA.y() << "]\n"
    << "Segment Point B\t= [" << SegIn1PtB.x() << ", " << SegIn1PtB.y() << "]\n"
    << "Intersection Point 1\t= [" << IntSegIn1_1.x() << ", " << IntSegIn1_1.y() << "]\n"
    << "Intersection Point 2\t= [" << IntSegIn1_2.x() << ", " << IntSegIn1_2.y() << "]\n"
    << std::endl
    << "Segment 2 with two intersections -> " << PtIn2 << " intersections found\n"
    << "Segment Point A\t= [" << SegIn2PtA.x() << ", " << SegIn2PtA.y() << "]\n"
    << "Segment Point B\t= [" << SegIn2PtB.x() << ", " << SegIn2PtB.y() << "]\n"
    << "Intersection Point 1\t= [" << IntSegIn2_1.x() << ", " << IntSegIn2_1.y() << "]\n"
    << "Intersection Point 2\t= [" << IntSegIn2_2.x() << ", " << IntSegIn2_2.y() << "]\n"
    << std::endl
```

```
66     << "Segment with no intersections -> " << PtOut << " intersections found\n"
67     << "Segment Point A\t= [" << SegOutPtA.x() << ", " << SegOutPtA.y() << "]\n"
68     << "Segment Point B\t= [" << SegOutPtB.x() << ", " << SegOutPtB.y() << "]\n"
69     << "Intersection Point 1\t= [" << IntSegOut_1.x() << ", " << IntSegOut_1.y() << "]\n"
70     << "Intersection Point 2\t= [" << IntSegOut_2.x() << ", " << IntSegOut_2.y() << "]\n"
71     << std::endl
72     << "Segment with one intersection -> " << PtTang << " intersection found\n"
73     << "Segment Point A\t= [" << SegTangPtA.x() << ", " << SegTangPtA.y() << "]\n"
74     << "Segment Point B\t= [" << SegTangPtB.x() << ", " << SegTangPtB.y() << "]\n"
75     << "Intersection Point 1\t= [" << IntSegTang_1.x() << ", " << IntSegTang_1.y() << "]\n"
76     << "Intersection Point 2\t= [" << IntSegTang_2.x() << ", " << IntSegTang_2.y() << "]\n"
77     << "\nCheck the results...\n"
78     << "\nGEOMETRY TEST 2: Completed\n";
79
80   // Exit the program
81   return 0;
82 }
```

C.1.3 Geometry-test3.cc

```
 1 // GEOMETRY TEST 3 - POINT INSIDE CIRCLE
 2
 3 #include <fstream>   // fStream - STD File I/O Library
 4 #include <iostream>  // Iostream - STD I/O Library
 5 #include <string>    // String - STD String Library
 6
 7 #include "PatchTire.hh"  // Tire Data Processing
 8 #include "RoadRDF.hh"    // Tire Data Processing
 9
10 // Main function
11 int
12 main() {
13   // Initialize disk
14   PatchTire::Disk NewDisk(RDF::vec2(0.0, 0.0), 0.0, 1.0);
15
16   // Query points and intersection bools
17   RDF::vec2 PointIn     = RDF::vec2(0.0, 0.0);
18   RDF::vec2 PointOut    = RDF::vec2(2.0, 0.0);
19   RDF::vec2 PointBorder = RDF::vec2(1.0, 0.0);
20
21   bool PtInBool, PtOutBool, PtBordBool;
22
23   // Calculate intersection
24   PtInBool   = NewDisk.isPointInside( PointIn );
25   PtOutBool  = NewDisk.isPointInside( PointOut );
26   PtBordBool = NewDisk.isPointInside( PointBorder );
27
28   std::cout
29     << "GEOMETRY TEST 3 - POINT INSIDE DISK\n\n"
30     << "Radius = " << NewDisk.getRadius() << std::endl
31     << "Origin = [" << NewDisk.getOriginXZ().x() << ", " << NewDisk.getOriginXZ().y() << "]\n";
32
33   // Show results
34   if ( PtInBool && !PtOutBool && PtBordBool ) {
35     std::cout
36       << "Point inside\t= ["
37       << PointIn.x() << ", " << PointIn.y() << "] -> Bool = " << PtInBool << std::endl
38       << "Point outside\t= ["
39       << PointOut.x() << ", " << PointOut.y()  << "] -> Bool = " << PtOutBool << std::endl
40       << "Point on border\t= ["
41       << PointBorder.x() << ", " << PointBorder.y() << "] -> Bool = "<< PtBordBool
42       << std::endl;
43   } else {
44     std::cout << "GEOMETRY TEST 3: Failed";
```

```
45   }
46
47   std::cout << "\nGEOMETRY TEST 3: Completed\n";
48
49   // Exit the program
50   return 0;
51 }
```

C.1.4 Geometry-test4.cc

```
 1 // GEOMETRY TEST 4 - POINT ON SEGMENT
 2
 3 #include <fstream>   // fStream - STD File I/O Library
 4 #include <iostream>  // Iostream - STD I/O Library
 5 #include <string>    // String - STD String Library
 6
 7 #include "PatchTire.hh"  // Tire Data Processing
 8 #include "RoadRDF.hh"    // Tire Data Processing
 9
10 // Main function
11 int
12 main() {
13   // Initialize segment points
14   RDF::vec2 PointA = RDF::vec2(0.0, 0.0);
15   RDF::vec2 PointB = RDF::vec2(1.0, 1.0);
16
17   // Query points and intersection bools
18   RDF::vec2 PointIn     = RDF::vec2(0.5, 0.5);
19   RDF::vec2 PointOut    = RDF::vec2(-1.0, -1.0);
20   RDF::vec2 PointBorder = RDF::vec2(1.0, 1.0);
21
22   // Calculate intersection
23   bool PtInBool   = PatchTire::algorithms::intersectPointSegment(PointA, PointB, PointIn);
24   bool PtOutBool  = PatchTire::algorithms::intersectPointSegment(PointA, PointB, PointOut);
25   bool PtBordBool = PatchTire::algorithms::intersectPointSegment(PointA, PointB, PointBorder);
26
27   std::cout
28     << "GEOMETRY TEST 4 - POINT ON SEGMENT\n\n"
29     << "Point A = [" << PointA[0] << ", " << PointA[1] << "]\n"
30     << "Point B = [" << PointB[0] << ", " << PointB[1] << "]\n\n";
31
32   // Show results
33   if ( PtInBool && !PtOutBool && PtBordBool ) {
34     std::cout
35       << "Point inside\t= ["
36       << PointIn[0] << ", " << PointIn[1] << "] -> Bool = " << PtInBool
37       << "\nPoint outside\t= ["
38       << PointOut[0] << ", " << PointOut[1] << "] -> Bool = " << PtOutBool
39       << "\nPoint on border\t= ["
40       << PointBorder[0] << ", " << PointBorder[1] << "] -> Bool = " << PtBordBool
41       << std::endl;
42   } else {
43     std::cout << "GEOMETRY TEST 4: Failed";
44   }
45
46   std::cout << "\nGEOMETRY TEST 4: Completed\n";
47
48   // Exit the program
49   return 0;
50 }
```

C.2 Tests per il Modello Magic Formula

C.2.1 MagicFormula-test1.cc

```
1 // PATCH EVALUATION TEST 1 - LOAD THE DATA FROM THE RDF FILE THEN PRINT IT INTO
2 // A FILE Out.txt. THEN CHARGE THE TIRE DATA AND ASSOCIATE THE CURRENT MESH TO
3 // IT.
4
5 #include <chrono>    // chrono - STD Time Measurement Library
6 #include <fstream>   // fStream - STD File I/O Library
7 #include <iostream>  // Iostream - STD I/O Library
8
9 #include "PatchTire.hh" // Tire Data Processing
10 #include "RoadRDF.hh"   // Tire Data Processing
11 #include "TicToc.hh"    // Processing Time Library
12
13 // Main function
14 int
15 main() {
16
17   try {
18
19     // Instantiate a TicToc object
20     TicToc tictoc;
21
22     std::cout
23       << "MAGIC FORMULA TIRE TEST 1 - CHECK INTERSECTION ON UNKNOWN MESH.\n\n";
24
25     // Load .rdf File
26     RDF::MeshSurface Road("./RDF_files/Eight.rdf");
27
28     // Print OutMesh.txt file
29     // Road.printData("OutMesh.txt");
30
31     // Make a new tire
32     PatchTire::ETRTO Tire;
33     Tire = PatchTire::ETRTO(205, 60, 15);
34
35     // Display current tire data on command line
36     Tire.print(std::cout);
37
38     // Orient the tire in the space
39     RDF::real_type Yaw    = 0.1*G2lib::m_pi;
40     RDF::real_type Camber = 0.1*G2lib::m_pi;
41
42     // Transformation matrix for X and Z-axis rotation
43     TireGround::mat3 Rot_Z;
44     Rot_Z << cos(Yaw), -sin(Yaw), 0,
45              sin(Yaw),  cos(Yaw), 0,
46                     0,        0, 1;
47     TireGround::mat3 Rot_X;
48     Rot_X << 1,           0,           0,
49              0, cos(Camber), -sin(Camber),
50              0, sin(Camber),  cos(Camber);
51     // Update Rotation Matrix
52     TireGround::mat3 RotMat = Rot_Z * Rot_X;
53
54     TireGround::vec3 Origin(0.8, 19.0, 0.26 ); //0.8, 19.0, 0.26
55     PatchTire::ReferenceFrame Pose(Origin, RotMat);
56
57     // Initialize the Magic Formula Tire
58     PatchTire::MagicFormula TireSD( Tire );
59
```

```
60    // Start chronometer
61    tictoc.tic();
62
63    // Set an orientation and calculate parameters
64    TireSD.setup( Road, Pose.getTotalTransformationMatrix(), true);
65
66    /* Example: Get results
67    |  PatchTire::vec3 N = TireSD.getContactNormal();
68    |  PatchTire::vec3 P = TireSD.getContactPoint();
69    |  PatchTire::real_type RelCamber = TireSD.getRelativeCamber();
70    |  PatchTire::real_type ContactFriction = TireSD.getContactFriction();
71    |  PatchTire::real_type Depth = TireSD.getContactDepth();
72    |  PatchTire::real_type Area = TireSD.getContactArea();
73    |  PatchTire::real_type Volume = TireSD.getContactVolume();
74    */
75
76    // Stop chronometer
77    tictoc.toc();
78
79    // This constructs a duration object using milliseconds
80    std::cout
81      << "Execution time = " << tictoc.elapsed_ms() << " ms\n"
82      << "\nCheck the results...\n"
83      << "\nMAGIC FORMULA TIRE TEST 1: Completed\n";
84
85  } catch ( std::exception const & exc ) {
86    std::cerr << exc.what() << '\n';
87  }
88  catch (...) {
89    std::cerr << "Unknown error\n";
90  }
91 }
```

C.2.2 MagicFormula-test2.cc

```
1 // PATCH EVALUATION TEST 2 - CHECK MF_Pacejka_SCP INTERSECTION
2
3 #include <fstream>   // fStream - STD File I/O Library
4 #include <iostream>  // Iostream - STD I/O Library
5 #include <string>    // String - STD String Library
6
7 #include "PatchTire.hh" // Tire Data Processing
8 #include "RoadRDF.hh"   // Tire Data Processing
9 #include "TicToc.hh"    // Processing Time Library
10
11 // Main function
12 int
13 main() {
14
15   try {
16
17     // Instantiate a TicToc object
18     TicToc tictoc;
19
20     std::cout
21       << "MAGIC FORMULA TIRE TEST 2 - CHECK INTERSECTION ON KNOWN MESH.\n\n";
22
23     // Initialize a quite big triangle
24     RDF::vec3  Vertices[3];
25     Vertices[0] = RDF::vec3(100.0, 0.0, 0.0);
26     Vertices[1] = RDF::vec3(0.0, 100.0, 0.0);
27     Vertices[2] = RDF::vec3(0.0, -100.0, 0.0);
28     RDF::TriangleRoad_list PtrTriangleVec;
29     PtrTriangleVec.push_back( RDF::TriangleRoad_ptr( new RDF::TriangleRoad(Vertices, 0.0) ) );
```

```
30
31    // Build the mesh
32    RDF::MeshSurface Road(PtrTriangleVec);
33
34    // Make a new tire
35    PatchTire::ETRTO Tire;
36    Tire = PatchTire::ETRTO(205, 60, 15);
37
38    // Display current tire data on command line
39    Tire.print(std::cout);
40
41    // Orient the tire in the space
42    RDF::real_type Yaw    = 0.1*G2lib::m_pi;
43    RDF::real_type Camber = 0.1*G2lib::m_pi;
44
45    // Transformation matrix for X and Z-axis rotation
46    TireGround::mat3 Rot_Z;
47    Rot_Z << cos(Yaw), -sin(Yaw), 0,
48             sin(Yaw),  cos(Yaw), 0,
49                    0,        0, 1;
50    TireGround::mat3 Rot_X;
51    Rot_X << 1,           0,           0,
52             0, cos(Camber), -sin(Camber),
53             0, sin(Camber),  cos(Camber);
54    // Update Rotation Matrix
55    TireGround::mat3 RotMat = Rot_Z * Rot_X;
56
57    TireGround::vec3 Origin( 50.0, 10.0, 0.26 );
58    PatchTire::ReferenceFrame Pose(Origin, RotMat);
59
60    // Initialize the Magic Formula Tire
61    PatchTire::MagicFormula TireSD( Tire );
62
63    // Start chronometer
64    tictoc.tic();
65
66    // Set an orientation and calculate parameters (true = print results)
67    TireSD.setup( Road, Pose.getTotalTransformationMatrix(), true);
68
69    /* Example: Get results
70    |  PatchTire::vec3 N = TireSD.getContactNormal();
71    |  PatchTire::vec3 P = TireSD.getContactPoint();
72    |  PatchTire::real_type RelCamber = TireSD.getRelativeCamber();
73    |  PatchTire::real_type ContactFriction = TireSD.getContactFriction();
74    |  PatchTire::real_type Depth = TireSD.getContactDepth();
75    |  PatchTire::real_type Area = TireSD.getContactArea();
76    |  PatchTire::real_type Volume = TireSD.getContactVolume();
77    */
78
79    // Stop chronometer
80    tictoc.toc();
81
82    // This constructs a duration object using milliseconds
83    std::cout
84      << "Execution time = " << tictoc.elapsed_ms() << " ms\n"
85      << "\nCheck the results...\n"
86      << "\nMAGIC FORMULA TIRE TEST 2: Completed\n";
87
88  } catch ( std::exception const & exc ) {
89    std::cerr << exc.what() << '\n';
90  }
91  catch (...) {
92    std::cerr << "Unknown error\n";
93  }
94 }
```

Bibliografia

[1] Lars Nyborg Egbert Bakker e Hans B. Pacejka. "Tyre Modelling for Use in Vehicle Dynamics Studies". In: *SAE Transactions* 96 (1987), pp. 190–204. ISSN: 0096736X.

[2] Juan J. Jiménez, Rafael J. Segura e Francisco R. Feito. "A Robust Segment/-Triangle Intersection Algorithm for Interference Tests. Efficiency Study". In: *Comput. Geom. Theory Appl.* 43.5 (lug. 2010), pp. 474–492. ISSN: 0925-7721. DOI: 10.1016/j.comgeo.2009.10.001. URL: http://dx.doi.org/10.1016/j.comgeo.2009.10.001.

[3] Dick De Waard Karel A. Brookhuis e Wiel H. Janssen. "Behavioural impacts of advanced driver assistance systems–an overview". In: *European Journal of Transport and Infrastructure Research 1.3* (2019).

[4] Matteo Larcher. "Development of a 14 Degrees of Freedom Vehicle Model for Realtime Simulations in 3D Environment". Master Thesis. University of Trento.

[5] Anu Maria. "Introduction to modeling and simulation". In: *Winter simulation conference* 29 (gen. 1997), pp. 7–13.

[6] Tomas Möller e Ben Trumbore. "Fast, Minimum Storage Ray-triangle Intersection". In: *J. Graph. Tools* 2.1 (ott. 1997), pp. 21–28. ISSN: 1086-7651. DOI: 10.1080/10867651.1997.10487468. URL: http://dx.doi.org/10.1080/10867651.1997.10487468.

[7] Hans Pacejka. *Tire and vehicle dynamics, 3rd Edition*. 2012.

[8] Georg Rill. *Road vehicle dynamics: fundamentals and modeling*. 2011.