



Università 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

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Obiettivi . . . . .	1
1.2	Il problema . . . . .	1
<b>2</b>	<b>Il Pneumatico</b>	<b>5</b>
2.1	Introduzione . . . . .	5
2.2	Geometria dello Pneumatico secondo ETRTO . . . . .	5
2.3	La Modellizzazione dello Pneumatico . . . . .	6
2.4	Il Modello della <i>Magic Formula</i> . . . . .	8
2.4.1	La <i>Magic Formula</i> . . . . .	8
2.4.2	Contatto con la Superficie Stradale . . . . .	9
<b>3</b>	<b>La Superficie Stradale</b>	<b>13</b>
3.1	Introduzione . . . . .	13
3.2	Il Formato RDF . . . . .	14
3.2.1	Superfici Semplici . . . . .	14
3.2.2	Superfici Complesse . . . . .	16
<b>4</b>	<b>Algoritmi</b>	<b>19</b>
4.1	Parsificazione . . . . .	19
4.1.1	Introduzione . . . . .	19
4.1.2	Parsificazione del formato RDF . . . . .	19
4.2	<i>Bounding Volume Hierarchy</i> . . . . .	20
4.2.1	Introduzione . . . . .	20
4.2.2	<i>Minimum Bounding Box</i> . . . . .	20
4.2.2.1	<i>Axis Aligned Bounding Box</i> . . . . .	21
4.2.2.2	<i>Arbitrarily Oriented Bounding Box</i> . . . . .	21

4.2.2.3	<i>Object Oriented Bounding Box</i> . . . . .	22
4.2.3	Intersezione tra Alberi AABB . . . . .	22
4.3	Algoritmi Geometrici . . . . .	23
4.3.1	Introduzione . . . . .	23
4.3.2	Intersezione tra Entità Geometriche . . . . .	25
4.3.2.1	Punto-Segmento . . . . .	25
4.3.2.2	Punto-Cerchio . . . . .	25
4.3.2.3	Segmento-Cerchio . . . . .	27
4.3.2.4	Piano-Piano . . . . .	29
4.3.2.5	Piano-Segmento . . . . .	31
4.3.2.6	Piano-Triangolo . . . . .	32
4.3.2.7	Raggio-Triangolo . . . . .	33
<b>5</b>	<b>La Libreria TireGround</b>	<b>39</b>
5.1	Organizzazione . . . . .	39
5.1.1	<i>Namespace</i> TireGround . . . . .	39
5.1.2	<i>Namespace</i> RDF . . . . .	39
5.1.3	<i>Namespace</i> PatchTire . . . . .	41
5.2	Librerie Esterne . . . . .	45
5.2.1	Eigen3 . . . . .	45
5.2.2	Clothoids . . . . .	45
5.3	Utilizzo e Prestazioni . . . . .	46
<b>6</b>	<b>Conclusioni e Lavoro Futuro</b>	<b>47</b>
<b>A</b>	<b>Convenzioni e Notazioni</b>	<b>49</b>
A.0.1	Sistemi di Riferimento . . . . .	49
A.0.2	Matrice di Trasformazione . . . . .	50
<b>B</b>	<b>Codice della Libreria C++</b>	<b>51</b>
B.1	TireGround.hh . . . . .	51
B.2	RoadRDF.hh . . . . .	52
B.3	RoadRDF.cc . . . . .	58
B.4	PatchTire.hh . . . . .	65
B.5	PatchTire.cc . . . . .	76

<b>C</b>	<b>Codice dei Tests</b>	<b>85</b>
C.1	Tests Geometrici . . . . .	85
C.1.1	Geometry-test1.cc . . . . .	85
C.1.2	Geometry-test2.cc . . . . .	86
C.1.3	Geometry-test3.cc . . . . .	88
C.1.4	Geometry-test4.cc . . . . .	89
C.2	Tests per il Modello Magic Formula . . . . .	90
C.2.1	MagicFormula-test1.cc . . . . .	90
C.2.2	MagicFormula-test2.cc . . . . .	91
	<b>Bibliografia</b>	<b>93</b>





# Elenco delle figure

2.1	Esempio di misure, secondo la notazione ETRTO, riportate sulla spalla dello pneumatico. . . . .	7
2.2	Forze e coppie generate dal contatto pneumatico-strada. . . . .	7
2.3	Curve caratteristiche generiche degli pneumatici derivate con il metodo della <i>Magic Formula</i> . . . . .	9
2.4	Geometria del contatto pneumatico-strada. . . . .	10
2.5	Punti campionati nel piano locale della superficie stradale. . . . .	11
2.6	Inclinazione longitudinale e laterale del piano strada locale. . . . .	12
4.1	Esempio di albero di tipo AABB. . . . .	21
4.2	Schema grafico per l'intersezione punto-segmento . . . . .	25
4.3	Schemi per l' <i>output</i> dell'intersezione punto-segmento. . . . .	25
4.4	Schema del codice per l'intersezione punto-segmento. . . . .	26
4.5	Schema del problema di intersezione punto-cerchio. . . . .	26
4.6	Schemi per l' <i>output</i> dell'intersezione punto-cerchio. . . . .	27
4.7	Schemi del codice per l'intersezione punto-cerchio. . . . .	27
4.8	Schema per del codice per l'intersezione segmento-cerchio. . . . .	29
4.9	Rappresentazione del problema di intersezione piano-piano. . . . .	30
4.10	Vettori dei piani $P_1$ , $P_2$ e della retta $L$ . . . . .	31
4.11	Schema per del codice per l'intersezione piano-piano. . . . .	31
4.12	Vettori dei piani $P_1$ , $P_2$ e della retta $L$ . . . . .	32
4.13	Schema per del codice per l'intersezione piano-segmento. . . . .	33
4.14	Schema per del codice per l'intersezione piano-triangolo. . . . .	33
4.15	Rappresentazione del problema di intersezione raggio-triangolo. . . .	34
4.16	Transformation and base change of ray in Möller-Trumbore algorithm.	35
4.17	Schemi per l' <i>output</i> dell'intersezione punto-cerchio. . . . .	37

4.18	Schema per del codice per l'intersezione raggio-triangolo con <i>back-face culling</i> . . . . .	37
A.1	Rappresentazione degli assi del sistema di riferimento della vettura secondo la convenzione ISO-V. . . . .	49
A.2	Rappresentazione degli assi del sistema di riferimento dello pneumatico secondo la convenzione ISO-C. . . . .	50

## Elenco delle tabelle

5.1	Attributi della classe BBox2D. . . . .	40
5.2	Attributi della classe Triangle3D. . . . .	40
5.3	Attributi della classe TriangleRoad. . . . .	41
5.4	Attributi della classe MeshSurface. . . . .	41
5.5	Attributi della classe Disk. . . . .	42
5.6	Attributi della classe BBox2D. . . . .	43
5.7	Attributi della classe ReferenceFrame. . . . .	43
5.8	Attributi della classe Shadow. . . . .	44
5.9	Attributi della classe Tire. . . . .	44
5.10	Attributi della classe MagicFormula. . . . .	45



# Elenco degli acronimi

<b>AABB</b> Axis Aligned Bounding Box . . . . .	21
<b>ADAS</b> Advanced Driver-Assistance Systems. . . . .	2
<b>AOBB</b> Arbitrarily Oriented Bounding Box . . . . .	21
<b>BB</b> Bounding Box . . . . .	22
<b>BVH</b> Bounding Volume Hierarchy . . . . .	20
<b>CAD</b> Computer-Aided Design . . . . .	24
<b>CAE</b> Computer-Aided Engineering. . . . .	24
<b>CAGD</b> Computer-Aided Geometric Design . . . . .	24
<b>CAM</b> Computer-Aided Manufacturing. . . . .	24
<b>ETRTO</b> European Tyre and Rim Technical Organisation. . . . .	3
<b>GIS</b> Geographic Information Systems . . . . .	24
<b>HIL</b> Hardware in the Loop . . . . .	2
<b>ISO</b> International Organization for Standardization . . . . .	49
<b>MBB</b> Minimum Bounding Box . . . . .	20
<b>RDF</b> Road Data File. . . . .	13



## 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 dal 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.



## 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  $a/bRc$ . 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:

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

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 ( $2 \times 107$  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.

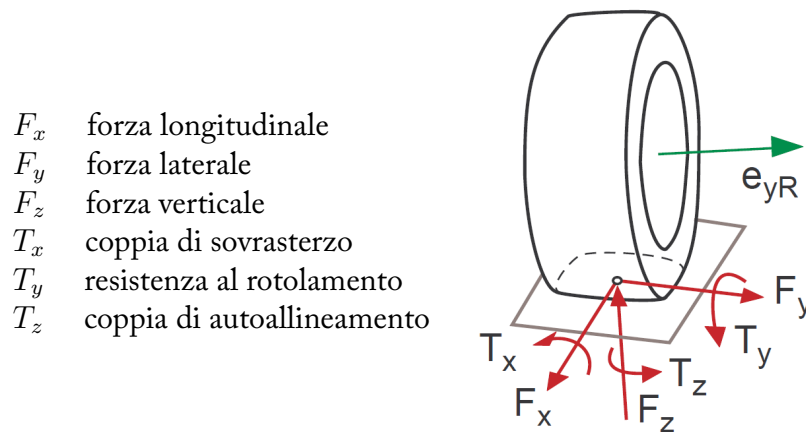


FIGURA 2.2: Forze e coppie generate dal contatto pneumatico-strada.

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:

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

dove:

- $B$  rappresenta il fattore di rigidezza;
- $C$  rappresenta il fattore di forma;
- $D$  rappresenta il valore 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*.

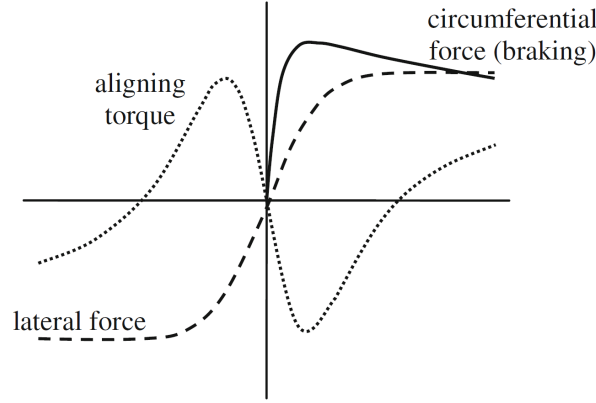


FIGURA 2.3: Curve caratteristiche generiche degli pneumatici derivate con il metodo della *Magic Formula*

### 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:

$$z = z(x, y) \quad (2.3)$$

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  $P^*$ , che è definito come una semplice traslazione del centro ruota  $M$ :

$$P^* = M - R_0 e_{zC} \begin{bmatrix} x^* \\ y^* \\ z^* \end{bmatrix} \quad (2.4)$$





e quindi, i quattro punti di campionamento sono:

$$\begin{aligned} {}^{P^*}r_{Q_{1,2}^*} &= P^* \pm \Delta x e_{xPC^*} \\ {}^{P^*}r_{Q_{3,4}^*} &= P^* \pm \Delta y e_{yPC^*} \end{aligned} \quad (2.8)$$

Al fine di campionare l'impronta di contatto nel modo più efficiente possibile, le distanze di  $\Delta x$  e  $\Delta y$ , 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.

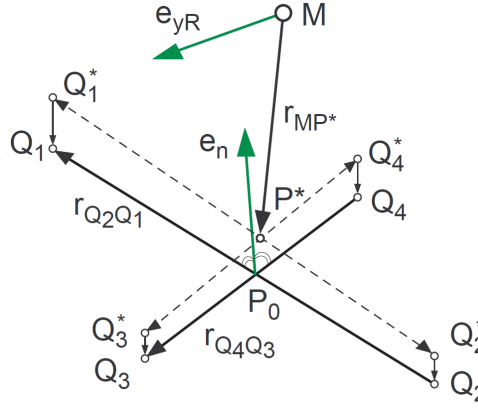


FIGURA 2.5: Punti campionati nel piano locale della superficie stradale.

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  $Q_i^*$ , 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 (Figura 2.6). Pertanto, il vettore normale è definito come:

$$e_n = \frac{r_{Q_1Q_2} \times r_{Q_4Q_3}}{|r_{Q_1Q_2} \times r_{Q_4Q_3}|} \quad (2.9)$$

dove sono  $r_{Q_2Q_1}$  e  $r_{Q_4Q_3}$  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  $e_x$  e  $e_y$  del piano di locale del punto di contatto. Il punto di contatto  $P$  si ottiene aggiornando le coordinate del primo punto di prova  $P^*$ , con il valore medio delle

tre coordinate spaziali dei quattro punti campione.

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

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

$$RF_{PC} = \left[ \begin{array}{ccc|c} [e_x] & [e_y] & [e_z] & x_P \\ & & & y_P \\ & & & z_P \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \quad (2.11)$$

Attraverso questo approccio, la normale del piano strada locale  $e_n$  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.

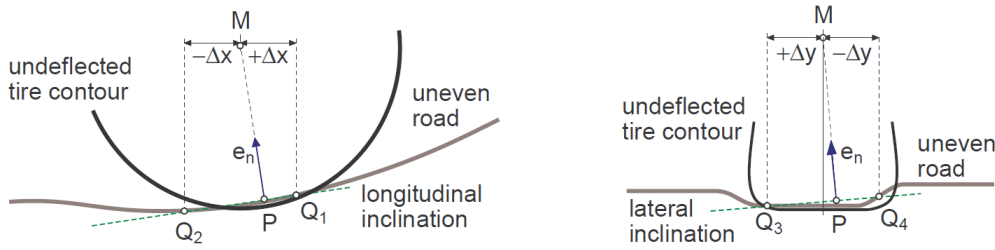


FIGURA 2.6: Inclinazione longitudinale e laterale del piano strada locale.

## 3.1 Introduzione

Oltre allo pneumatico, la superficie stradale rappresenta il secondo importante elemento che definisce il contatto. Perché 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 triangolo.

È importante notare che la discretizzazione del manto stradale è un passaggio molto importante in quanto, se campionato 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 occorre 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  $45^\circ$  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
```

```

6 { X_road Z_left Z_right }
7 (XZ_DATA)
8 -1.0e04 0 0
9 0.0500 0 0
10 0.1000 0 0
11 0.1500 0 0
12 ... ... ...

```

---

- Sinusoide:

La strada a superficie sinusoidale è implementata come:

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

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  $(n_1, n_2, n_3, \mu)$  data dal numero di identificazione dei tre vertici componenti  $i$ -esimo triangolo e dal coefficiente 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 dichiarazione dei nodi della *mesh*.

- X\_SCALE: riscalda i punti delle coordinate dei nodi lungo l'asse  $X$ .
- Y\_SCALE: riscalda i punti delle coordinate dei nodi lungo l'asse  $Y$ .
- Z\_SCALE: riscalda 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
```

---



## 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 parsificare 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.

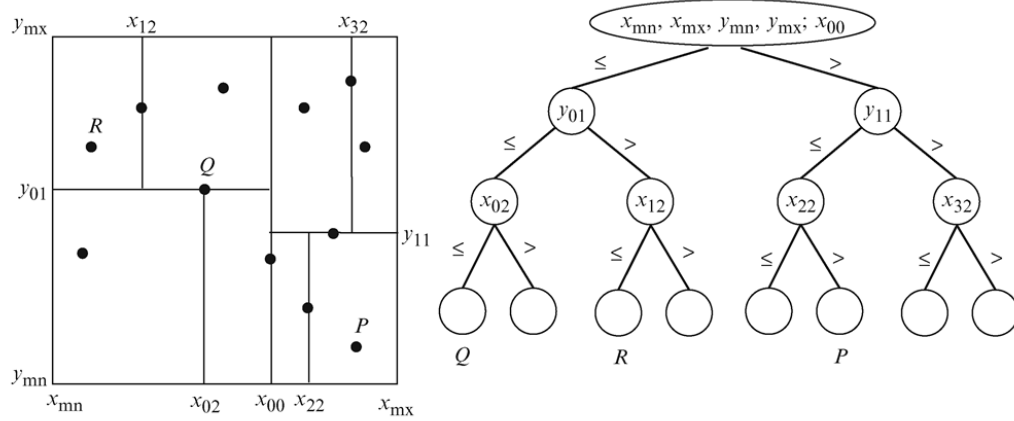


FIGURA 4.1: Esempio di albero di tipo AABB.

#### 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 porterebbe 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  $A = [A.minX, A.maxX; A.minY, A.maxY]$  e  $B = [B.minX, B.maxX; B.minY, B.maxY]$ , 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 importante 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 triangoli, il metodo presentato consente di ridurre logaritmamente il numero di comparazioni necessarie per ottenere il risultato.
- Riduzione del numero di triangoli da processare per ottenere il piano strada locale e il punto di contatto virtuale dello pneumatico. Infatti, vengono solamente processati quelli posti in corrispondenza dell'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'involuppo 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  $A = (x_A, y_A)$  e  $B = (x_B, y_B)$ .

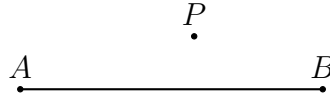


FIGURA 4.2: Schema grafico per l'intersezione punto-segmento

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

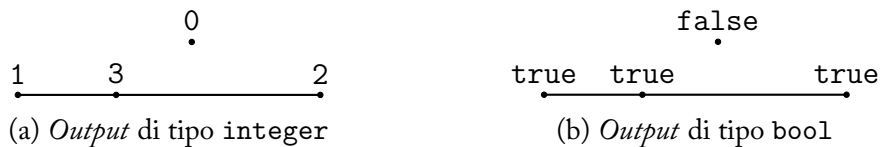


FIGURA 4.3: Schemi per l'*output* dell'intersezione punto-segmento.

#### 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:

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

<i>Output</i> di tipo integer	<i>Output</i> di tipo bool
<pre> 1  if ( AB.cross(AP) &gt; epsilon ) 2    { return 0; } 3  KAP = AB.dot(AP); 4  if ( KAP &lt; -epsilon ) 5    { return 0; } 6  if ( abs(KAP) &lt; epsilon ) 7    { return 1; } 8  KAB = AB.dot(AB); 9  if ( KAP &gt; KAB ) 10   { return 0; } 11 if ( abs(KAP-KAB) &lt; epsilon ) 12   { return 2; } 13 return 3; </pre>	<pre> 1  if ( AB.cross(AP) &gt; epsilon ) 2    { return false; } 3  KAP = AB.dot(AP); 4  if ( KAP &lt; -epsilon ) 5    { return false; }; 6  if ( abs(KAP) &lt; epsilon ) 7    { return true; } 8  KAB = AB.dot(AB); 9  if ( KAP &gt; KAB ) 10   { return false; } 11 if ( abs(KAP-KAB) &lt; epsilon ) 12   { return true; } 13 return true; </pre>

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

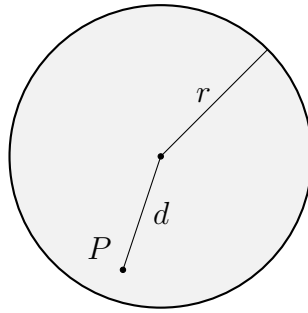


FIGURA 4.5: Schema del problema di intersezione punto-cerchio.

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.

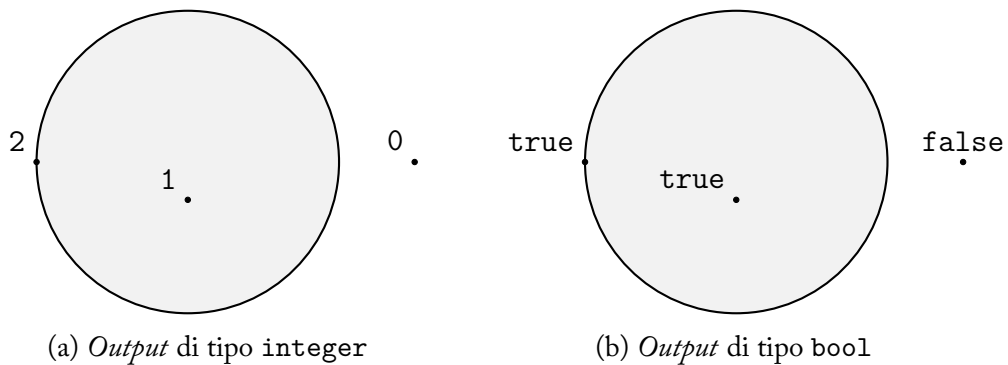


FIGURA 4.6: Schemi per l'*output* dell'intersezione punto-cerchio.

<i>Output</i> di tipo integer	<i>Output</i> di tipo bool
<pre> 1  d = (x_p-x_c)^2 + (y_p-y_c)^2; 2  if ( d &gt; r^2 ){ return 0; } 3  else if ( d &lt; r^2 ){ return 1; } 4  else { return 2; }</pre>	<pre> 1  d = (x_p-x_c)^2 + (y_p-y_c)^2; 2  if ( d &gt; r^2 ){ return true; } 3  else { return false; }</pre>

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

#### 4.3.2.3 Segmento-Cerchio

Per l'intersezione di un segmento, avente punto iniziale e finale rispettivamente  $P_0$  e  $P_1$ , con una circonferenza, avente centro  $C = (x_c, y_c)$ , è necessario prima di tutto riscrivere le equazione di entrambe le entità come:

$$ax + by = c \quad (4.2)$$

per il segmento e:

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad (4.3)$$

per la circonferenza. Assumendo che il centro  $C$  sia posto sull'origine, la precedente equazione si può semplificare come:

$$x^2 + y^2 = r^2 \quad (4.4)$$

Per trovare i termini  $a$ ,  $b$  e  $c$  del segmento è necessario calcolare la direzione del segmento come differenza tra il punto finale e iniziale del segmento:

$$\vec{d} = P_1 - P_0 \quad (4.5)$$

È neccerio anche trovare il vettore tra l'origine e il punto  $P_1$ :

$$\vec{P}_{O1} = P_1 - O \quad (4.6)$$

I termini  $a$ ,  $b$  e  $c$  del segmento saranno quini pari a:

$$\begin{aligned} a &= \vec{d} \cdot \vec{d} \\ b &= 2(\vec{d} \cdot \vec{P}_{O1}) \\ c &= \vec{P}_{O1} \cdot \vec{P}_{O1} - r^2 \end{aligned} \quad (4.7)$$

Risolve l'equazione 4.2 per  $x$  o  $y$  è ora molto semplice. Basta infatti sostituirla nell'equazione 4.4 per ottenere le soluzioni  $(x_1, y_1)$  e  $(x_2, y_2)$  con:

$$x_{1/2} = \frac{ac \pm b\sqrt{r^2(a^2 + b^2) - c^2}}{a^2 + b^2} \quad (4.8)$$

oppure:

$$y_{1/2} = \frac{bc \mp a\sqrt{r^2(a^2 + b^2) - c^2}}{a^2 + b^2} \quad (4.9)$$

Se  $r^2(a^2 + b^2) - c^2 \geq 0$  vale come una disuguaglianza stretta, esistono due punti di intersezione; in questo caso la linea è chiamata una linea secante del cerchio e il segmento di linea che collega i punti di intersezione è chiamato accordo del cerchio. Se invece vale  $r^2(a^2 + b^2) - c^2 = 0$ , allora esiste solo un punto di intersezione e la linea è tangente al cerchio. Se la disuguaglianza debole non regge, la linea non interseca la circonferenza.

Dal punto di vista del codice l'*output* può essere un intero il cui valore può essere:

- 0 se la linea non interseca la circonferenza;
- 1 se la linea interseca la circonferenza in un solo punto, ovvero è tangente;
- 2 se la linea interseca la circonferenza in due punti.

---

```

1  a = d · d;
2  b = 2 * (d · P_01);
3  c = P_01 · P_01 - r^2;
4  discriminant = r^2 * (a^2 + b^2) - c^2;
5  if ( a <= epsilon || discriminant < 0.0 ) {
6      IntPt_1 = (quiteNaN, quiteNaN);
7      IntPt_2 = (quiteNaN, quiteNaN);
8      return 0;
9  } else if ( abs(discriminant) < epsilon ) {
10     t = - b / (2 * a);
11     IntPt_1 = P_1 + t * d;
12     IntPt_2 = (quiteNaN, quiteNaN);
13     return 1;
14 } else {
15     t = (-b + sqrt(discriminant)) / (2 * a);
16     IntPt_1 = P_1 + t * d;
17     t = (-b - sqrt(discriminant)) / (2 * a);
18     IntPt_2 = P_1 + t * d;
19     return 2;
20 }

```

---

FIGURA 4.8: Schema per del codice per l'intersezione segmento-cerchio.

#### 4.3.2.4 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:

$$\vec{n}_i \cdot P + d_i = 0 \quad (4.10)$$

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 tre 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

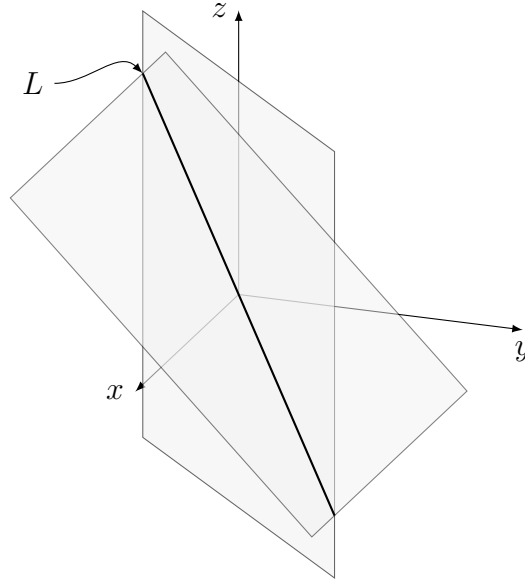


FIGURA 4.9: Rappresentazione del problema di intersezione piano-piano.

modi per farlo, il più semplice è attraverso l'aggiunta di 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 tre piani è:

$$P_0 = \frac{-d_1(\vec{n}_2 \times \vec{n}_3) - d_2(\vec{n}_3 \times \vec{n}_1) - d_3(\vec{n}_1 \times \vec{n}_2)}{\vec{n}_1 \cdot (\vec{n}_2 \times \vec{n}_3)} \quad (4.11)$$

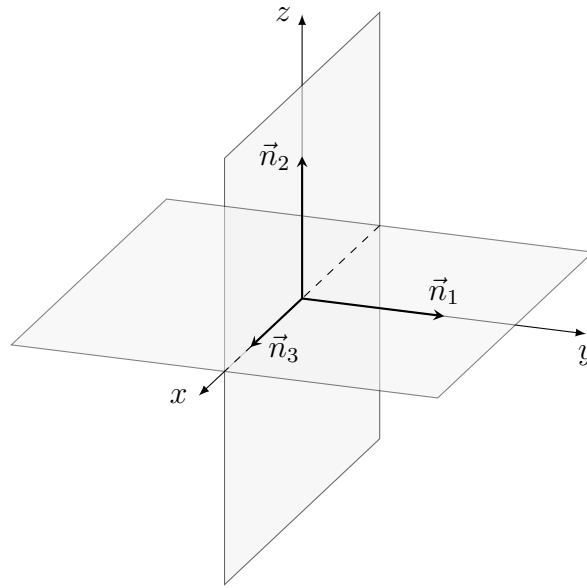
e ponendo  $d_3 = 0$  per  $P_3$ , si ottiene:

$$P_0 = \frac{-d_1(\vec{n}_2 \times \vec{n}_3) - d_2(\vec{n}_3 \times \vec{n}_1)}{\vec{n}_1 \cdot (\vec{n}_2 \times \vec{n}_3)} = \frac{(d_2\vec{n}_1 - d_1\vec{n}_2) \times \vec{n}_3}{(\vec{n}_1 \times \vec{n}_2) \cdot \vec{n}_3} = \frac{(d_2\vec{n}_1 - d_1\vec{n}_2) \times \vec{u}}{|\vec{u}|^2} \quad (4.12)$$

e l'equazione parametrica per la retta  $L$  sarà:

$$L(s) = \frac{(d_2\vec{n}_1 - d_1\vec{n}_2) \times \vec{u}}{|\vec{u}|^2} + s\vec{u} \quad (4.13)$$

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

FIGURA 4.10: Vettori dei piani  $P_1$ ,  $P_2$  e della retta  $L$ .

---

```

1  u = n_1 × n_2;
2  if ( u.norm() > epsilon ) {
3    d_1 = - V_1 · n_1;
4    d_2 = - V_2 · n_2;
5    u_1 = d_1 * n_1;
6    u_2 = - d_2 * n_2;
7    P_0 = (u1 + u2) × u / (u · u);
8    return true;
9  } else {
10   return false;
11 }

```

---

FIGURA 4.11: Schema per del codice per l'intersezione piano-piano.

#### 4.3.2.5 Piano-Segmento

Nello spazio delle coordinate tridimensionali, una linea  $L$  può essere o parallela a un piano  $P$  o può intersecarlo in un singolo punto. Sia  $L$  data dall'equazione parametrica:

$$P(s) = P_0 + s(P_1 - P_0) = P_0 + s\vec{u} \quad (4.14)$$

mentre il piano  $P$  sia dato da un punto  $V_0$  appartenente ad esso e da un vettore normale  $\vec{n} = (a, b, c)$ . Per prima cosa è necessario controllare se  $L$  è parallelo a  $P$  verificando se  $\vec{n} \cdot \vec{u} = 0$ , il che significa che il vettore di direzione della linea  $\vec{u}$  è

perpendicolare al piano normale  $\vec{n}$ . Se questo è vero, allora  $L$  e  $P$  sono paralleli e non si intersecano, oppure  $L$  giace totalmente nel piano  $P$ . Disgiunzione o coincidenza possono essere determinate testando se in  $P$  esiste un punto specifico di  $L$ , per esempio  $P_0$ , ovvero se soddisfa l'equazione di linea implicita:

$$\vec{n} \cdot (P_0 - V_0) = 0 \quad (4.15)$$

Se la linea e il piano non sono paralleli, allora  $L$  e  $P$  si intersecano in un unico punto  $P(s_I)$ . Nel punto di intersezione, il vettore  $P(s) - V_0 = \vec{w} + s\vec{u}$  è perpendicolare a  $\vec{n}$ , dove  $\vec{w} = P_0 - V_0$ . Ciò equivale alla condizione del prodotto scalare:

$$\vec{n} \cdot (\vec{w} + s\vec{u}) = 0 \quad (4.16)$$

Risolvendo si ottiene:

$$s_I = -\frac{\vec{n} \cdot \vec{w}}{\vec{n} \cdot \vec{u}} = -\frac{\vec{n} \cdot (V_0 - P_0)}{\vec{n} \cdot (P_1 - P_0)} \quad (4.17)$$

Se la linea  $L$  è un segmento finito da  $P_0$  a  $P_1$ , è sufficiente verificare che  $0 \leq s_I \leq 1$  per verificare che vi sia un'intersezione tra il segmento e il piano. Per raggio, c'è invece un'intersezione con il piano quando  $s_I \geq 0$ .

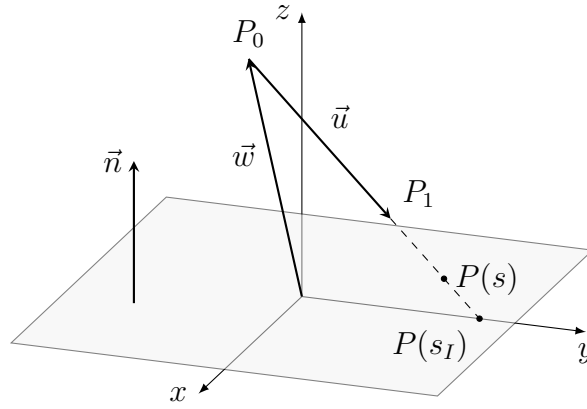


FIGURA 4.12: Vettori dei piani  $P_1$ ,  $P_2$  e della retta  $L$ .

#### 4.3.2.6 Piano-Triangolo

Per risolvere l'intersezione piano triangolo basta usare la soluzione precedentemente trovata per il problema dell'intersezione tra piano e segmento. Nello specifico, basta trattare i lati del triangolo come tre segmenti distinti e per ognuno di esso applicare la funzione per l'intersezione piano-segmento. Vi saranno tre possibili soluzioni:

---

```

1  u = P_1 - P_0;
2  d = v_0 · n;
3  s = -(P_1 · n) + d) / (u · n);
4  if (s >= 0 && s <= 1) {
5      P_sI = P_0 + u * s;
6      return true;
7  } else {
8      return false;
9  }

```

---

FIGURA 4.13: Schema per del codice per l'intersezione piano-segmento.

- il triangolo non viene intersecato dal piano;
- il triangolo viene intersecato dal piano in uno dei suoi tre vertici;
- il triangolo viene intersecato dal piano, formando quindi due punti d'intersezione nel suo perimetro.

---

```

1  if ( intersectSegmentPlane( n, V_0, 1, IntPt_1 )
      )
2  { IntPts.push_back(IntPt1); }
3  if ( intersectSegmentPlane( n, V_0, 2, IntPt2 ) )
4  { IntPts.push_back(IntPt2); }
5  if ( intersectSegmentPlane( n, V_0, 3, IntPt3 ) )
6  { IntPts.push_back(IntPt3); }
7  if ( IntPts.size() == 2 )
8  { return true; }
9  else if ( IntPts.size() == 0 )
10 { return false; }
11 else
12 { return false; }

```

---

FIGURA 4.14: Schema per del codice per l'intersezione piano-triangolo.

#### 4.3.2.7 Raggio-Triangolo

Dato un triangolo avente vertici  $(A, B, C)$  e un raggio  $R$  con origine  $R_O$  e direzione  $\vec{R}_D$ , 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 numerosi algoritmi per risolvere questo problema, esistono quindi diverse soluzioni

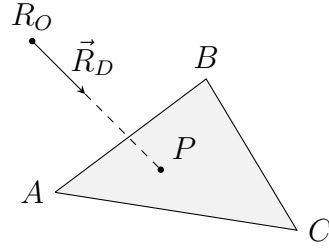


FIGURA 4.15: Rappresentazione del problema di intersezione raggio-triangolo.

al problema di intersezione raggio-triangolo. Tre degli algoritmi più importanti sono:

- l'algoritmo di *Badouel*;
- l'algoritmo di *Segura*;
- l'algoritmo 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:

$$P = wA + uB + vC \quad (4.18)$$

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

$$P = (1 - u - v)A + uB + vC \quad (4.19)$$

e sviluppando si ottiene:

$$P = A - uA - vA + uB + vC = A + u(B - A) + v(C - A) \quad (4.20)$$

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:

$$P = R_O + t\vec{R}_D \quad (4.21)$$

dove  $t$  è la distanza dall'origine del raggio all'intersezione  $P$ . Sostituendo  $P$  nell'equazione 4.20 con l'equazione del raggio si ottiene:

$$\begin{aligned} R_O + t\vec{R}_D &= A + u(B - A) + v(C - A) \\ O - A &= -tD + u(B - A) + v(C - A) \end{aligned} \quad (4.22)$$



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 4.22 usando la seguente notazione:

$$\begin{bmatrix} -D & (B - A) & (C - A) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = R_O - A \quad (4.23)$$

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  $(0, 0)$ ,  $(1, 0)$ ,  $(0, 1)$ .

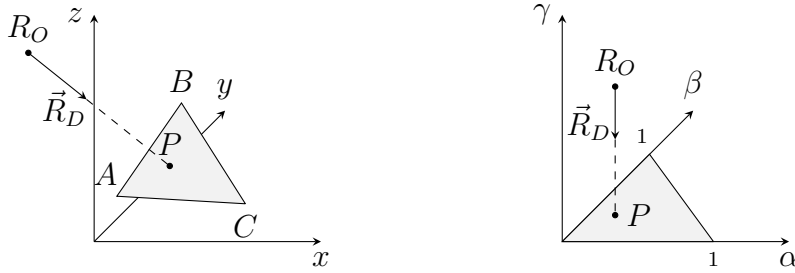


FIGURA 4.16: Transformation and base change of ray in Möller-Trumbore algorithm.

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 sa 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

re 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 4.23 (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 4.23, 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;

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\begin{vmatrix} -D & E_1 & E_2 \end{vmatrix}} \begin{bmatrix} \begin{vmatrix} T & E_1 & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & T & E_2 \end{vmatrix} \\ \begin{vmatrix} -D & E_1 & T \end{vmatrix} \end{bmatrix} \quad (4.24)$$

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  $3 \times 3$ ) non è altro che un triplo prodotto scalare, quindi si può riscrivere l'equazione precedente come:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix} \quad (4.25)$$

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  $v$ .

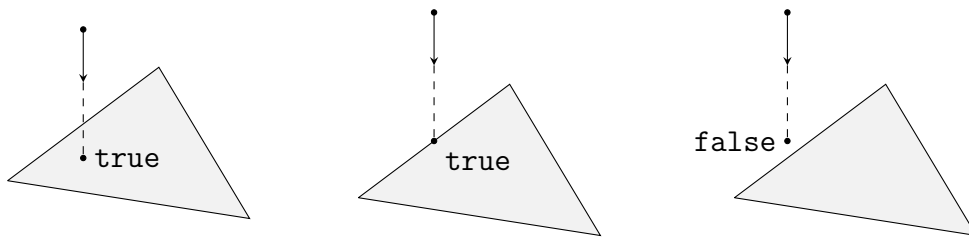


FIGURA 4.17: Schemi per l'output dell'intersezione punto-cerchio.

---

```

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



## 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`.

- `updateBBBox2D` – 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	TriangleBBBox	•	•	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 permetterà 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 $\mu$

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 `PtrBBBox`, 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
<code>TriangleRoad_list</code>	Friction	•		Vettore dei triangoli
<code>std::vector&lt;PtrBBBox&gt;</code>	<code>PtrBBBoxVec</code>	•		Vettore delle BB
<code>PtrAABB</code>	<code>PtrTree</code>	•		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 tutte le classi e le 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 tipologie 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  $4 \times 4$  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
ETRT0	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 operazioni 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 libreria

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

### 5.3 Utilizzo e Prestazioni





## 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*.

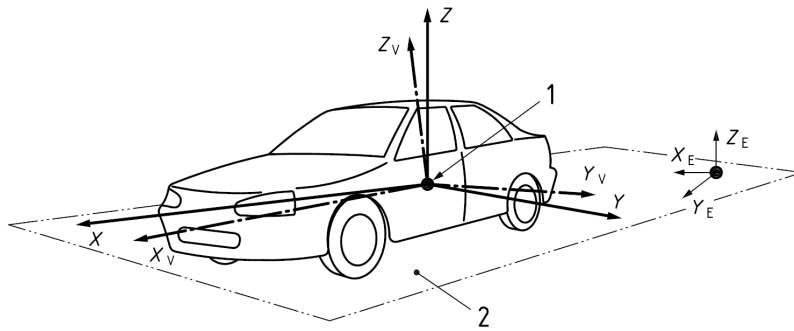


FIGURA A.1: Rappresentazione degli assi del sistema di riferimento della vettura secondo la convenzione ISO-V.

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].

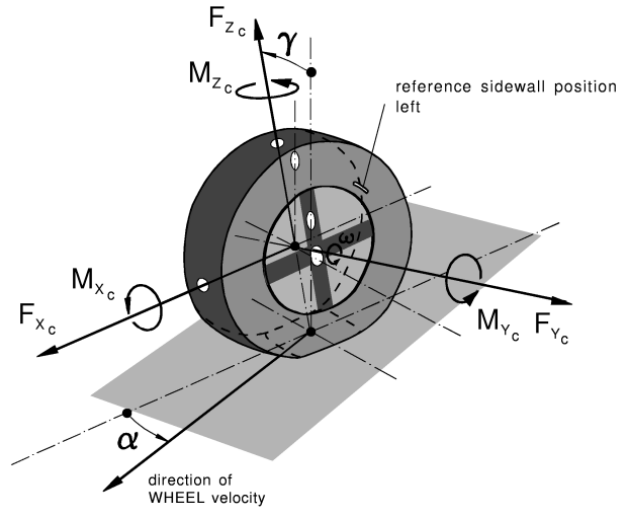


FIGURA A.2: Rappresentazione degli assi del sistema di riferimento dello pneumatico secondo la convenzione ISO-C.

### 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:

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

dove  $R_m$  è la matrice di rotazione  $3 \times 3$  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.



## B.1 TireGround.hh

```
1 /*!
2
3 \mainpage
4
5 Master's Thesis
6
7 Algorithm for Tire Contact Patch Evaluation in Soft Real-Time
8
9 Academic Year 2019 · 2020
10
11 Graduant:
12 -----
13
14 Davide Stocco\n
15 Department of Industrial Engineering\n
16 University of Trento\n
17 davide.stocco@studenti.unitn.it
18
19 Supervisor:
20 -----
21
22 Prof. Enrico Bertolazzi\n
23 Department of Industrial Engineering\n
24 University of Trento\n
25 enrico.bertolazzi@unitn.it
26
27 */
28
29 ///
30 /// file: TireGround.hh
31 ///
32
33 #pragma once
34
35 #include <Eigen/Dense> // Eigen linear algebra Library
36 #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 {

```



```

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 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
120 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
121 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
122 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
123 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
124 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
125 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
126 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
127 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
128 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
129 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
130 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
131 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
132 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
133 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
134 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
135 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
136 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
137 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
138 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
139 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
140 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
141 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
142 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
143 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
144 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
145 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
146 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
147 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
148 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
149 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
150 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
151 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
152 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
153 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
154 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
155 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

```

```

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 };

```











```
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)
```





```

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         return true;
397     }
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 |  | | 0        | 0 = OriginXZ
33 |  | \          /
34 |  |      _ _ _ _ _
35 |  Y-----> X
36 |
37 | ISO Reference Frame!
38 \*/
39
40 class ReferenceFrame;
41 class ETRT0;
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 circumference.
102     //!< If output bool is true the point is inside the circumference,
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 1
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

```

## B. CODICE DELLA LIBRERIA C++

```

151 //!< 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; }

```



```

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.geometricktools.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.geometricktools.com/Documentation/EulerAngles.pdf

```





```

476 | | | | | _/_ \
477 | | | | | | _/_
478 | | | | | \_/_
479 |
480 |         ^ Z
481 |         |
482 |         --|--
483 |        / | \
484 |       | 0=Y----|----> X
485 |      \ | /
486 |       ----
487 |
488 |         ^ Y
489 |         |
490 |        ----|----
491 |       | | |
492 |      | 0=Z----|----> X
493 |     |-----|
494 |
495 | 0 = 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

```





```

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 approximated 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 approximated 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 approximated 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 approximated 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             //!< Transformation 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,  //!< Transformation 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      |  _ _ _ _ _ \ O _ _ _ _ | _ _
12      |  | | | | / _ _ | | / /
13      |  | | | | \ _ _ \ <
14      |  | _ _ _ / | _ _ _ / _ \ \
15      \*/
16
17     //! Check if a point lays inside or outside a circumference.
18     //! If output bool is true the point is inside the circumference,
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 P0( Point - OriginXZ );
26         return P0.dot(P0) <= 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 1
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 discriminant = B*B - 4 * A * C;
45     if ( A <= epsilon || discriminant < 0 ) {
46         // No real solutions
47         Intersect_1 = vec2(quietNaN, quietNaN);
48         Intersect_2 = vec2(quietNaN, quietNaN);
49         return 0;
50     } else if ( std::abs(discriminant) < epsilon ) {
51         // One solution
52         t_param = -B / (2*A);
53         Intersect_1 = Point_1 + t_param * d;
54         Intersect_2 = vec2(quietNaN, quietNaN);
55         return 1;
56     } else {
57         // Two solutions
58         t_param = (-B + std::sqrt(discriminant)) / (2 * A);
59         Intersect_1 = Point_1 + t_param * d;
60         t_param = (-B - std::sqrt(discriminant)) / (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 = .cross(Line_Direction) / Line_Direction.dot(Line_Direction); // In absolute reference
89         frame return
90         return true;
91     } else {
92         return false;
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 reference 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

```



```

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 | \ _ _ _ _ _ / \ _ _ _ _ _ / \ _ _ _ _ _ / \ _ _ _ _ _ /
292 | \ _ _ _ _ _ / \ _ _ _ _ _ / \ _ _ _ _ _ / \ _ _ _ _ _ /

```



## B. CODICE DELLA LIBRERIA C++

[illegible]



```

423         << "\tCamber angle\n"
424         << "\t = " << 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         << "\t = " << 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";
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

```

## 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\nGEOMETRY TEST 1: Completed\n";
91
92 // Exit the program
93 return 0;
94 }
```

### C.1.2 Geometry-test2.cc

```

1 // GEOMETRY TEST 2 - SEGMENT CIRCLE 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
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     // Initialize segments points
17     RDF::vec2 SegIn1PtA = RDF::vec2(0.0, 0.0);
18     RDF::vec2 SegIn1PtB = RDF::vec2(0.0, 1.0);
19
20     RDF::vec2 SegIn2PtA = RDF::vec2(-2.0, 0.0);
21     RDF::vec2 SegIn2PtB = RDF::vec2(2.0, 0.0);
22
23     RDF::vec2 SegOutPtA = RDF::vec2(1.0, 2.0);
24     RDF::vec2 SegOutPtB = RDF::vec2(-1.0, 2.0);
25
26     RDF::vec2 SegTangPtA = RDF::vec2(1.0, 1.0);
27     RDF::vec2 SegTangPtB = RDF::vec2(-1.0, 1.0);
28
29     // Initialize intersection points and output types
30     RDF::vec2 IntSegIn1_1, IntSegIn1_2, IntSegIn2_1, IntSegIn2_2, IntSegOut_1,
31     IntSegOut_2, IntSegTang_1, IntSegTang_2;
32     RDF::int_type PtIn1, PtIn2, PtOut, PtTang;
33
34     // Calculate intersections
35     PtIn1 = NewDisk.intersectSegment(
36         SegIn1PtA, SegIn1PtB, IntSegIn1_1, IntSegIn1_2
37     );
38     PtIn2 = NewDisk.intersectSegment(
39         SegIn2PtA, SegIn2PtB, IntSegIn2_1, IntSegIn2_2
40     );
41     PtOut = NewDisk.intersectSegment(
42         SegOutPtA, SegOutPtB, IntSegOut_1, IntSegOut_2
43     );
44     PtTang = NewDisk.intersectSegment(
45         SegTangPtA, SegTangPtB, IntSegTang_1, IntSegTang_2
46     );
47
48     // Display results
49     std::cout
50     << "GEOMETRY TEST 2 - SEGMENT DISK INTERSECTION\n\n"
51     << "Radius = " << NewDisk.getRadius() << std::endl
52     << "Origin = [" << NewDisk.getOriginXZ().x() << ", " << NewDisk.getOriginXZ().y() << "]\n"
53     << std::endl
54     << "Segment 1 with two intersections -> " << PtIn1 << " intersections found\n"
55     << "Segment Point A\t= [" << SegIn1PtA.x() << ", " << SegIn1PtA.y() << "]\n"
56     << "Segment Point B\t= [" << SegIn1PtB.x() << ", " << SegIn1PtB.y() << "]\n"
57     << "Intersection Point 1\t= [" << IntSegIn1_1.x() << ", " << IntSegIn1_1.y() << "]\n"
58     << "Intersection Point 2\t= [" << IntSegIn1_2.x() << ", " << IntSegIn1_2.y() << "]\n"
59     << std::endl
60     << "Segment 2 with two intersections -> " << PtIn2 << " intersections found\n"
61     << "Segment Point A\t= [" << SegIn2PtA.x() << ", " << SegIn2PtA.y() << "]\n"
62     << "Segment Point B\t= [" << SegIn2PtB.x() << ", " << SegIn2PtB.y() << "]\n"
63     << "Intersection Point 1\t= [" << IntSegIn2_1.x() << ", " << IntSegIn2_1.y() << "]\n"
64     << "Intersection Point 2\t= [" << IntSegIn2_2.x() << ", " << IntSegIn2_2.y() << "]\n"
65     << 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     }
```



```

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:ETRTOTire;
36 Tire = PatchTire:ETRTOTire(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:MagicFormulaTireSD( 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.