UNIVERSITÁ DI TRENTO

Dipartimento di Ingegneria Industriale

Laurea Magistrale in Ingegneria Meccatronica

Valutazione *Real-Time* del Contatto Pneumatico/Strada con Algoritmi Dedicati

Relatore:
Prof. Enrico Bertolazzi

Co-relatore:
Dott. Ing. Matteo Ragni

Candidato:
Davide Stocco

Anno Accademico 2019 · 2020

Sommario

This dissertation details …

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli acronimi

Introduzione 1

1.1 Obiettivi

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

1.2 Il problema

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

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

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

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

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

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

Lo Pneumatico 2

2.1 Introduzione

Gli pneumatici sono probabilmente i componenti più complessi di un'auto in quanto combinano decine di componenti che devono essere formati, assemblati e combinati assieme. Il successo del prodotto finale dipende dalla loro capacità di fondere tutti i componenti separati in un prodotto dal materiale coeso che soddisfa le esigenze del conducente [10]. Gli pneumatici sono caratterizzati da un comportamento altamente non lineare con una dipendenza da diversi fattori costruttivi e ambientali.

2.2 Geometria

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

- a rappresenta larghezza nominale dello 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.

Si prenda come esempio la seguente formula: 195/55R16. La larghezza nominale dello pneumatico è di circa 195 mm nel punto più largo, l'altezza della spalla corrisponde al 55% della larghezza - ovvero 107 mm - e il diametro dei cerchi ai quali lo pneumatico si adatta è di 16 pollici. Con questa notazione è possibile calcolare direttamente il diametro esterno teorico dello pneumatico tramite la seguente:

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

Riprendendo l'esempio usato sopra, il diametro esterno risulterà dunque 24.44 in o 621 mm.
Meno comunemente usato negli Stati Uniti e in Europa (ma spesso in Giappone) è una notazione che indica l'intero diametro del pneumatico invece delle proporzioni dell'altezza della parete laterale, quindi non secondo ETRTO. Per fare lo stesso esempio, una ruota da 16 pollici avrebbe un diametro di 406 mm. L'aggiunta del doppio dell'altezza del pneumatico (2×107 mm) produce un diametro totale di 620 mm. Quindi, un pneumatico 195/55R16 potrebbe in alternativa essere etichettato 195/620R16.
Anche se queste due notazioni sono teoricamente ambigue, in pratica possono essere facilmente distinte perché l'altezza della parete laterale di uno 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 Modellizzazione

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 perché deve essere considerato anche il comportamento non stazionario.

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

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

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

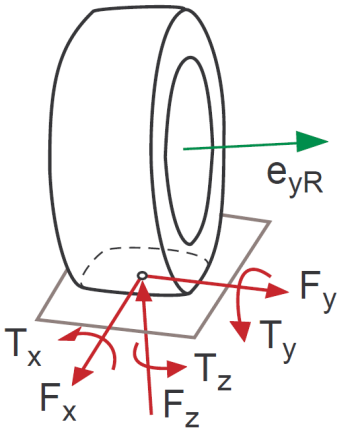


Figura 2.2: Forze e coppie generate dal contatto pneumatico/strada.
Da: Rill, *Road Vehicle Dynamics - Fundamentals and Modeling*.

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

La prima tipologia di modello tenta di rappresentare le caratteristiche fisiche dello 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 in-

sieme di dati. Un esempio ben noto di questo approccio è il **modello di Pacejka** o ***Magic Formula*** [8]. Questo tipo di modellazione è adatta per la simulazione di guida in cui 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 il comfort 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 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.

2.3.1 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 [8]. Il modello *Magic Formula* consiste in una pura descrizione matematica del rapporto input-output del contatto pneumatico-strada. Questa formulazione collega le variabili di forza con lo slip rigido del corpo che vengono trattati nelle sezioni successive. La forma generale della funzione di descrizione può essere scritta come:

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

dove:

- B rappresenta il fattore di rigidezza;
- C rappresenta il fattore di forma;
- D rappresenta il falore massimo della forza o coppia;
- E rappresenta il fattore di curvatura;
- S_v rappresenta lo spostamento in verticale della curva caratteristica;
- S_h rappresenta lo spostamento in orizzontale della curva caratteristica.

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

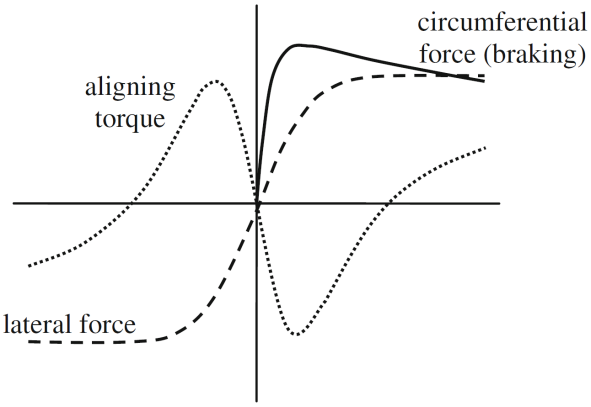


FIGURA 2.3: Curve caratteristiche generiche degli pneumatici derivate con il metodo della *Magic Formula*.
Da: Schramm, Hiller e Bardini, *Vehicle Dynamics: Modeling and Simulation*.

2.3.2 Contatto con la Superficie Stradale

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

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

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

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

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

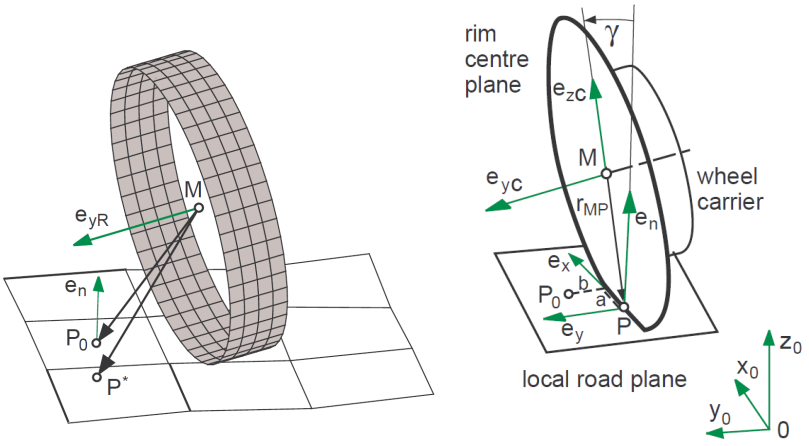


Figura 2.4: Geometria del contatto pneumatico-strada.
Da: Rill, *Road Vehicle Dynamics - Fundamentals and Modeling*.

La prima stima del sistema di riferimento del punto di contatto $RF_{PC^\star}$ è una terna con origine in $P^\star$ e orientazione degli assi definiti dall'orientazione del sistema di riferimento della ruota. Notare che l'origine di $RF_{PC^\star}$ corrisponde alla proiezione lungo l'asse z del sistema di riferimento della ruota sulla *mesh* rappresentante la strada.

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

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

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

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

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

e quindi, i quattro punti di campionamento sono:

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

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

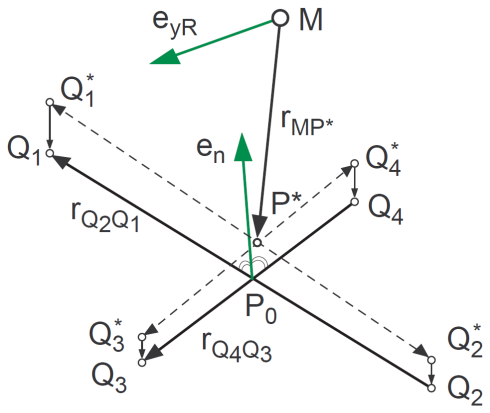


FIGURA 2.5: Punti campionati nel piano locale della superficie stradale.
Da: Rill, *Road Vehicle Dynamics - Fundamentals and Modeling*.

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

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

dove sono $\boldsymbol{r}_{Q_2Q_1}$ e $\boldsymbol{r}_{Q_4Q_3}$ sono i vettori che puntano rispettivamente da Q_1 a Q_2 e da Q_3 a Q_4. Applicando l'equazione 2.6 è ora possibile calcolare i vettori unitari $\boldsymbol{e}_x$ e $\boldsymbol{e}_y$ del piano di locale del punto di contatto. Il punto di contatto P si ottiene

aggiornando le coordinate del primo punto di prova $P^{\star}$, con il valore medio delle tre coordinate spaziali dei quattro punti campione.

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

È ora necessario spostare il punto di contatto P in modo da ricondursi alle condizioni tali per cui il modello di Pacejka è valido. Per fare ciò si andrà a sfuttare un algoritmo di intersezione piano-raggio. Si troverà dapprima la componente della normale al piano strada giacente sul piano mediano dello pneumatico $\boldsymbol{e}_{n_{XZ}}$, si sposterà dunque il punto P sulla proiezione del punto M sul piano strada locale con direzione $-\boldsymbol{e}_{n_{XZ}}$ il come illustrato in Figura 2.6.

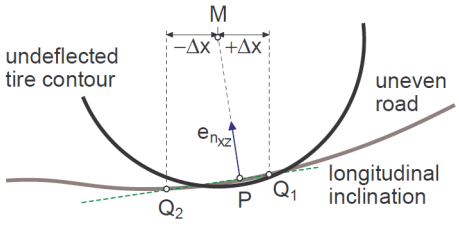


Figura 2.6: Inclinazione del piano strada locale e spostamento del punto di contatto in relazione alla normale.
Da: Rill, *Road Vehicle Dynamics - Fundamentals and Modeling*.

Infine si può mettere assieme tutte le componenti del piano di riferimento del punto di contatto P ottenendo:

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

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

La Superficie Stradale 3

3.1 Introduzione

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

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

3.2 Il Formato RDF

3.2.1 Superfici Semplici

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

```
1  { Comments section }
2
3  [UNITS]
4  LENGTH = 'meter'
5  ANGLE = 'degree'
6
7  [MODEL]
8  ROAD\_TYPE = '...'
9
10 [PARAMETERS]
11 ...
```

Nella sezione `[UNITS]`, vengono impostate le unità di misura utilizzate nel file di dati stradali. La sezione `[MODEL]` viene invece utilizzata per specificare la morfologia della superfice stradale, del tipo:

- `ROAD_TYPE = 'flat'`: superficie stradale piana.
- `ROAD_TYPE = 'plank'`: singolo scalino o dosso orientato perpendicolarmente o obliquo rispetto all'asse X, con o senza bordi smussati.
- `ROAD_TYPE = 'poly_line'`: altezza della strada è in funzione della distanza percorsa.
- `ROAD_TYPE = 'sine'`: superficie stradale costituita da una o più onde sinusoidali con lunghezza d'onda costante.

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

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

- Generali:

  - `MU`: è il fattore di correzione dell'attrito stradale (non il valore dell'attrito stesso), da moltiplicare con i fattori di ridimensionamento LMU del modello di pneumatico.
    Impostazione predefinita: `MU = 1.0`.
  - `OFFSET`: è l'offset verticale del terreno rispetto al sistema di riferimento inerziale.
  - `ROTATION_ANGLE_XY_PLANE`: è l'angolo di rotazione del piano XY attorno all'asse Z della strada, ovvero la definizione dell'asse X positivo della strada rispetto al sistema di riferimento inerziale.

- Strada con scalino:
  - `HEIGHT`: altezza dello scalino.
  - `START`: distanza lungo l'asse X della strada all'inizio dello scalino.
  - `LENGTH`: lunghezza dello scalino (escluso lo smusso) lungo l'asse X della strada.
  - `BEVEL_EDGE_LENGTH`: lunghezza del bordo smussato a $45°$ dello scalino.
  - `DIRECTION`: rotazione dello scalino attorno all'asse Z, rispetto all'asse Y della strada.
    Se lo scalino è posizionato trasversalmente, `DIRECTION = 0`. Se lo scalino è posto lungo l'asse X, `DIRECTION = 90`.

- Polilinea:
  Il blocco `[PARAMETERS]` deve avere un sottoblocco chiamato `(XZ_DATA)` e costituito da tre colonne di dati numerici:
  - La colonna 1 è un insieme di valori X in ordine crescente.
  - Le colonne 2 e 3 sono insiemi di rispettivi valori Z per la traccia sinistra e destra.

  Esempio:

```
1  [PARAMETERS]
2  MU = 1.0
3  OFFSET = 0.0
4  ROTATION_ANGLE_XY_PLANE = 0.0
5
6  { X_road  Z_left  Z_right }
7  (XZ_DATA)
8  -1.0e04 0 0
```

```
0.0500  0 0
0.1000  0 0
0.1500  0 0
... ... ...
```

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

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

  dove

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

  I parametri sono:

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

3.2.2 Superfici Complesse

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

Per descrivere una superficie stradale composta da una *mesh* di triangoli si userà la seguente struttura dati.

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

Esempio:

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

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

- `X_SCALE`: riscala i punti delle coordinate dei nodi lungo l'asse X.
- `Y_SCALE`: riscala i punti delle coordinate dei nodi lungo l'asse Y.
- `Z_SCALE`: riscala i punti delle coordinate dei nodi lungo l'asse Z.
- `ORIGIN`: definisce la posizione dell'origine della sistema di riferimento della superficie stradale.
- `UP`: definisce la direzione positiva dell'asse Z.
- `[ORIENTATION]`: ruota i punti delle coordinate dei nodi secondo la matrice definita.

Esempio:

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

3.3 Parsificazione

3.3.1 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.

3.3.2 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.

Algoritmi Geometrici 4

4.1 *Bounding Volume Hierarchy*

4.1.1 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.

4.1.2 *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.

4.1.2.1 *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.

4.1.2.2 *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 bi-

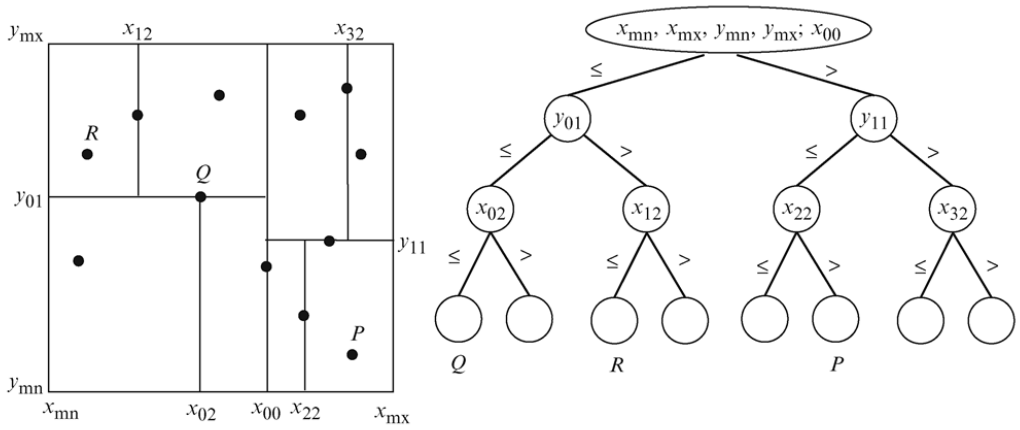


FIGURA 4.1: Esempio di albero di tipo AABB.

dimensionale 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.

4.1.2.3 *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.

4.1.3 Intersezione tra Alberi AABB

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

Il vincolo di allineamento dei rettangoli agli assi è presente per motivi di prestazioni, infatti, l'area di sovrapposizione tra due riquadri non ruotati può essere controllata solo con confronti logici. Mentre i riquadri ruotati richiedono ulteriori operazioni trigonometriche, che sono più lente da calcolare. Inoltre, se si hanno entità che possono ruotare, le dimensioni dei rettangoli e/o sotto-rettangoli dovranno

modificarsi in modo da avvolgere ancora l'oggetto o si dovrà optare per un altro tipo di geometria di delimitazione, come le sfere (che sono invarianti alla rotazione).

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

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

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

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

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

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

4.2 Algoritmi Geometrici

4.2.1 Introduzione

La geometria computazionale è la branca dell'informatica che studia le strutture dati e gli algoritmi efficienti per la soluzione di problemi di natura geometrica e la loro implementazione al calcolatore. Storicamente, è considerato uno dei campi più antichi del calcolo, anche se la geometria computazionale moderna è uno sviluppo recente. La ragione principale per lo sviluppo della geometria computazionale è stata dovuta ai progressi compiuti nella computer grafica, *Computer-Aided Design* (CAD), *Computer-Aided Manufacturing* (CAM) e nella visualizzazione matematica. Ad oggi, le applicazioni della geometria computazionale si trovano nella robotica, nella progettazione di circuiti integrati, nella visione artificiale, in *Computer-Aided Engineering* (CAE) e nel *Geographic Information Systems* (GIS). I rami principali della geometria computazionale sono:

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

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

- punto e segmento (nel piano);
- punto e cerchio (nel piano);

- segmento e circonferenza (nel piano);
- piano e piano (nello spazio);
- piano e segmento (nello spazio);
- piano e raggio (nello spazio);
- piano e triangolo (nello spazio);
- raggio e triangolo (nello spazio);

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

4.2.2 Intersezione tra Entità Geometriche

4.2.2.1 Punto-Segmento

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

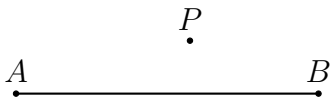


Figura 4.2: Schema del problema di intersezione punto-segmento

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

4.2.2.2 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

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

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

Output di tipo `integer`

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

Output di tipo `bool`

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

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

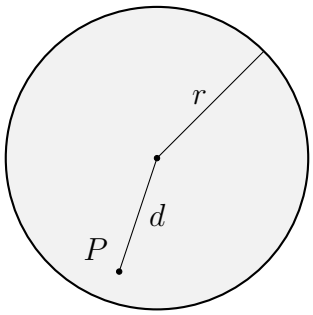


FIGURA 4.5: Schema del problema di intersezione punto-cerchio.

cerchio C e il punto P è data dal teorema di Pitagora. In particolare:

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

il punto P è dunque interno alla circonferenza se $d < r$, appartiene alla circonferenza se $d = r$ ed esterno alla circonferenza se $d > r$. In maniera analoga ma più efficace da punto di vista computazionale si può confrontare d^2 con r^2. Il punto P è dunque interno alla circonferenza se $d^2 < r^2$, appartiene alla circonferenza se

$d^2 = r^2$ ed esterno alla circonferenza se $d^2 > r^2$. Pertanto, il confronto finale sarà tra il numero $(x_p - x_c)^2 + (y_p - y_c)^2$ e r^2.
Gli *inputs* dell'algoritmo per l'intersezione punto-cerchio sono:

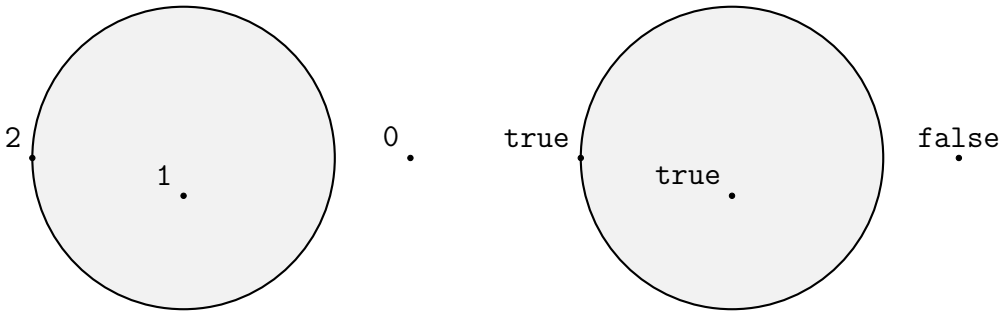
- il centro della circonferenza $C = (x_c, y_c)$;
- il raggio della circonferenza r;
- il punto generico da analizzare $P = (x_p, y_p)$.

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

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

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

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



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

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

Output di tipo `integer`

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

Output di tipo `bool`

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

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

4.2.2.3 Segmento-Circonferenza

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:

$$ax + by = c \tag{4.2}$$

per il segemento e:

$$(x - x_c)^2 + (y - y_c)^2 = r^2 \tag{4.3}$$

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

$$x^2 + y^2 = r^2 \tag{4.4}$$

Per trovere i termini a, b e c del segmento è necessario calcolare la direzione del

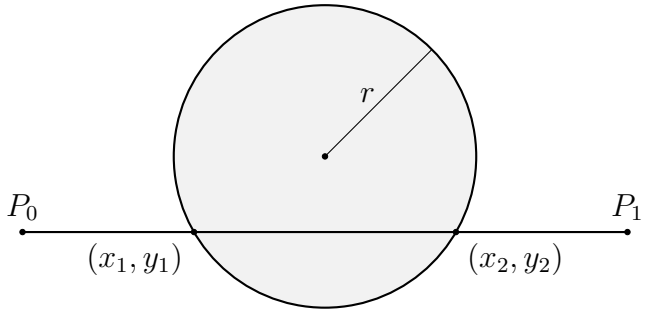


FIGURA 4.8: Schema del problema di intersezione punto-circonferenza.

segmento come differenza tra il punto finale e iniziale del segmento:

$$\vec{d} = P_1 - P_0 \tag{4.5}$$

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

$$\vec{P}_{O1} = P_1 - O \tag{4.6}$$

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

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

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

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

oppure:

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

Se $r^2(a^2+b^2)-c^2 \geq 0$ vale come una disuguaglianza stretta, esistono due punti di intersezione.
Se invece vale $r^2(a^2+b^2)-c^2 = 0$, allora esiste solo un punto di intersezione e la linea è tangente alla circonferenza. 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.

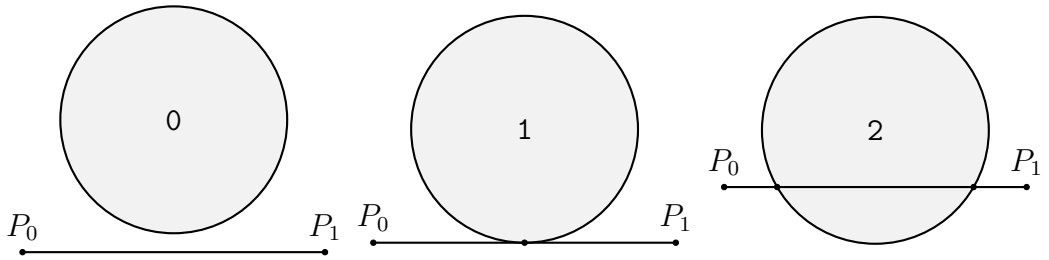


Figura 4.9: Schemi per l'*output* dell'intersezione segmento-cerchio.

4.2.2.4 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:

$$\vec{n}_i \cdot P + d_i = 0 \tag{4.10}$$

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 $\vec{n}_1 \times \vec{n}_2 = 0$.

```
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 4.10: Schema per del codice per l'intersezione segmento-cerchio.

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 modi per farlo, il più semplice è attraverso l'aggiunta dei un terzo piano P_3 avente equazione implicita $\vec{n}_3 \cdot P = 0$ dove $\vec{n}_3 = \vec{n}_1 \times \vec{n}_2$ e $d_3 = 0$ (ovvero passa attraverso l'origine). Questo metodo è funzionante poiché:

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

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

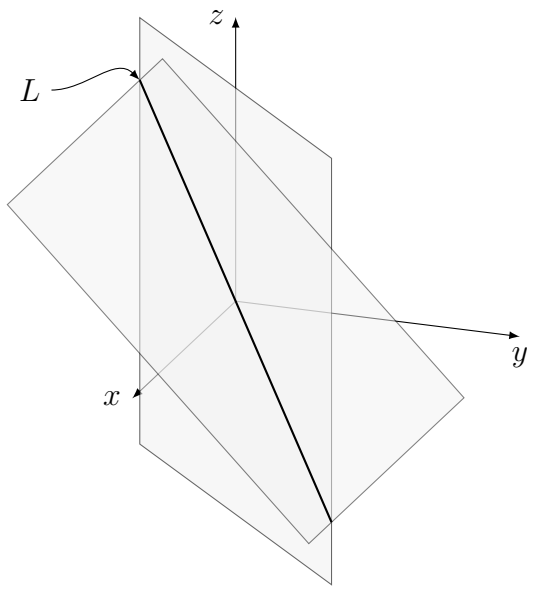


Figura 4.11: Schemi del problema di intersezione piano-piano.

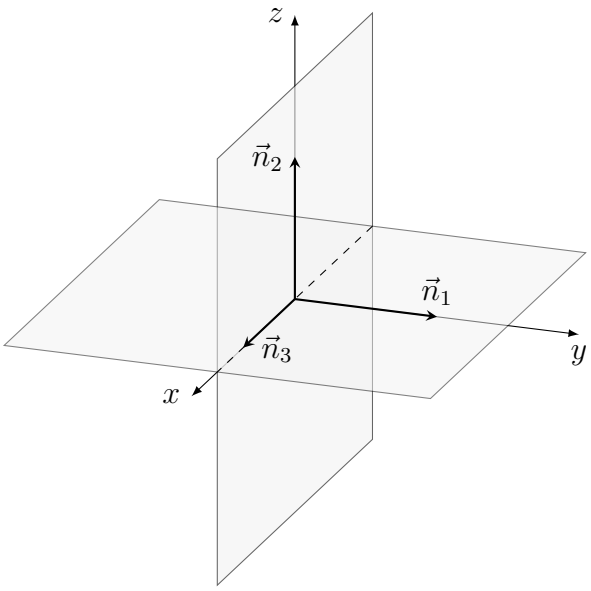


Figura 4.12: Vettori dei piani P_1, P_2 e della retta L.

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

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

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

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

e l'equazione parametrica per la retta L sarà:

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

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

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

4.2.2.5 Piano-Segmento e Piano-Raggio

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:

$$P(t) = P_0 + t(P_1 - P_0) = P_0 + t\vec{u} \tag{4.14}$$

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:

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

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

$$\vec{n} \cdot (\vec{w} + t\vec{u}) = 0 \tag{4.16}$$

Risolvendo si ottiene:

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

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

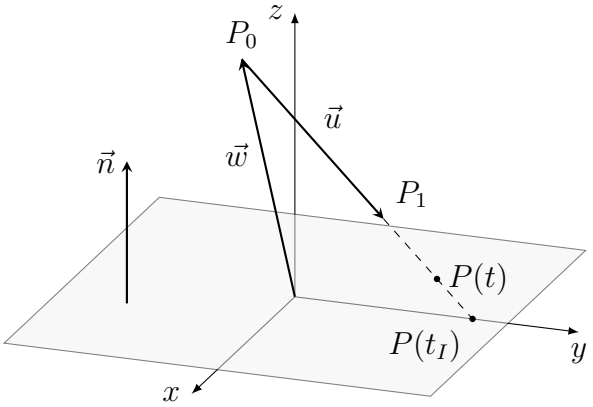


Figura 4.14: Vettori dei piani P_1, P_2 e della retta L.

4.2.2.6 Piano-Triangolo

Per risolvere l'intersezione piano triangolo bastaunsare la soluzione precedentemente trovata per il problame 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  t = n · (V_0 - P_0) / (u · n);
3  if ( t >= 0 && t <= 1 ) {
4    P_tI = P_0 + u * t;
5    return true;
6  } else {
7    return false;
8  }
```

FIGURA 4.15: 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 4.16: Schema per del codice per l'intersezione piano-triangolo.

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

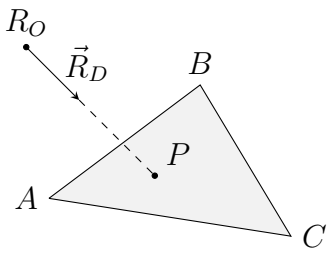


Figura 4.17: Schema del problema di intersezione raggio-triangolo.

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

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

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

$$P = wA + uB + vC \tag{4.18}$$

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

$$P = (1 - u - v)A + uB + vC \tag{4.19}$$

e sviluppando si ottiene:

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

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:

$$P = R_O + t\vec{R}_D \tag{4.21}$$

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:

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

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:

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

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

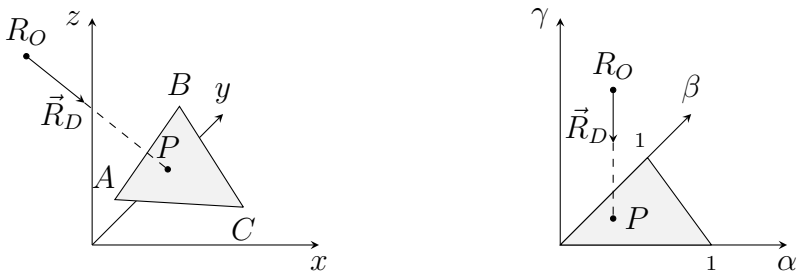


Figura 4.18: Cambiamento di coordinate nell'algoritmo di Möller-Trumbore.

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

Möller e Trumbore spiegano che la prima parte dell'equazione 4.23 (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;

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

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

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

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.

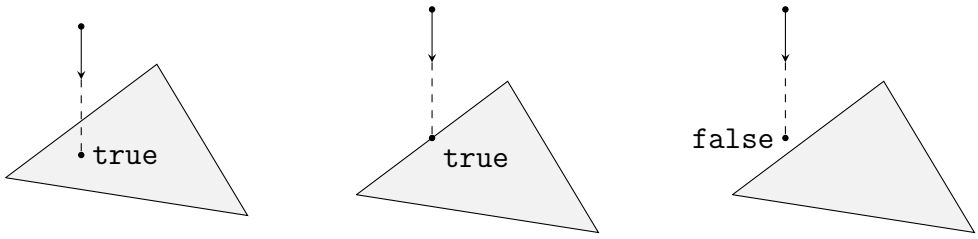


FIGURA 4.19: Schemi per l'*output* dell'intersezione punto-cerchio.

```
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_O - 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_O - 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 4.20: Schema per del codice per l'intersezione raggio-triangolo con *back-face culling*.

La Libreria TireGround 5

5.1 Organizzazione

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

5.1.1 *Namespace* TireGround

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

5.1.2 *Namespace* RDF

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

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

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

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

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

Tabella 5.1: Attributi della classe `BBox2D`.

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

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

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

Tabella 5.2: Attributi della classe `Triangle3D`.

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

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

Tabella 5.3: Attributi della classe `TriangleRoad`.

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

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

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

Tabella 5.4: Attributi della classe `MeshSurface`.

5.1.3 *Namespace* `PatchTire`

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

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

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

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

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

Tabella 5.5: Attributi della classe `Disk`.

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

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

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

Tabella 5.6: Attributi della classe `ETRTO`.

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

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

Tabella 5.7: Attributi della classe `ReferenceFrame`.

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

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

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

Tabella 5.8: Attributi della classe `Shadow`.

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

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

Tabella 5.9: Attributi della classe `Tire`.

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

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

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

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

Tabella 5.10: Attributi della classe `MagicFormula`.

5.2 Librerie Esterne

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

5.2.1 `Eigen3`

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

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

5.2.2 `Clothoids`

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

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

5.2.3 Doxygen

Doxygen è un *software* comunemente utilizzato per generare documentazione direttamente dalle annotazioni nei file C++. Questo *tool* supporta anche altri linguaggi di programmazione popolari come C, Objective-C, C#, PHP, Java, Python, Fortran, VHDL, Tcl e in una certa misura D.

Doxygen può essere utile per i seguenti motivi.

- Può generare una documentazione da utilizzare *online* (in HTML) e/o un manuale di riferimento *offline* (in LaTeX) da una serie di *file* sorgente oppurtunamente annotati. C'è anche il supporto per generare *output* in RTF (MicroSoft Word), PostScript, PDF con *hyperlink* e HTML compresso. La documentazione viene estratta direttamente dalle fonti, il che rende molto più semplice mantenere la documentazione coerente con il codice sorgente.
- È possibile configurare doxygen per estrarre la struttura del codice da *file* sorgente non documentati. Questo è molto utile per analizzare rapidamente ed efficaciemente i *file* sorgente di grandi dimensioni. Doxygen può anche visualizzare le relazioni tra i vari elementi mediante grafici di dipendenza, diagrammi di ereditarietà e diagrammi di collaborazione, tutti generati automaticamente.

Doxygen è sviluppato su Mac OS X e Linux, ma è configurato per essere altamente portabile. Di conseguenza, funziona anche con la maggior parte degli altri sistemi Unix. Inoltre, sono disponibili eseguibili per Windows.

5.3 Utilizzo e Prestazioni

Conclusioni e Lavoro Futuro 6

Convenzioni e Notazioni A

A.0.1 Sistemi di Riferimento

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

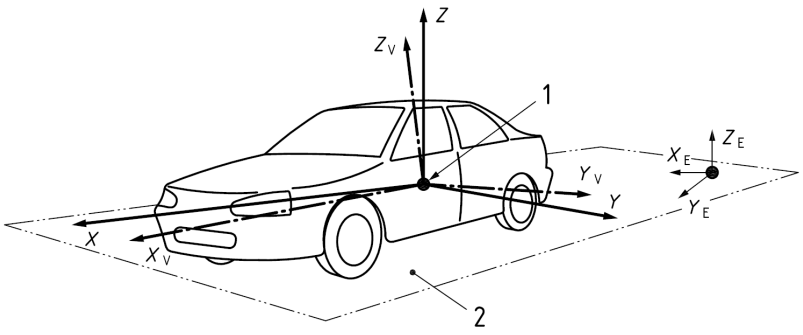


Figura A.1: Rappresentazione degli assi del sistema di riferimento della vettura secondo la convenzione ISO-V.
Da: Normalización (Ginebra), *Road Vehicles, Vehicle Dynamics and Road-holdin Ability: Vocabulary*.

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].

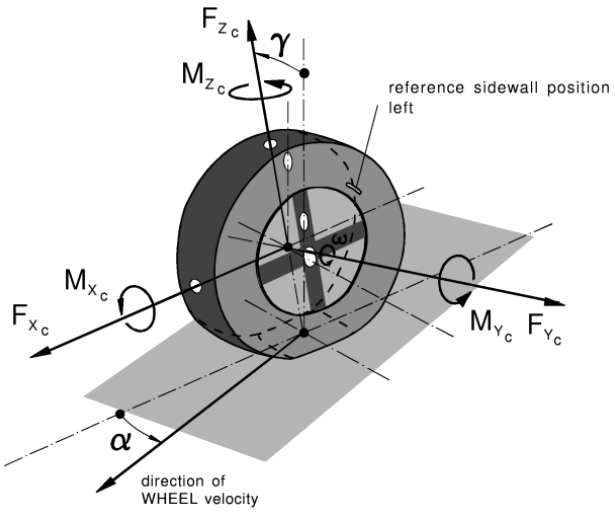


Figura A.2: Rappresentazione degli assi del sistema di riferimento dello pneumatico secondo la convenzione ISO-C.
Da: Documentazione `MFeval`.

A.0.2 Matrice di Trasformazione

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

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

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

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

le informazioni traslazionali vengono trascurate imponendo l'elemento fittizio pari a 0.

Codice della Libreria C++ B

B.1 TireGround.hh

```
/*!

\mainpage

Department of Industrial Engineering \n
Master Degree in Mechatronics Engineering \n
\n
EN: <b> Real-Time Computation of Tire/Road Contact using Tailored Algorithms </b> \n
IT: <b> Valutazione Real-Time del Contatto Pneumatico/Strada con Algoritmi Dedicati </b> \n
\n
Academic Year 2019 · 2020 \n

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

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

Co-supervisor:
----------
Dr.Eng. Matteo Ragni \n
AnteMotion S.r.l.

*/

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

```
37
38 #pragma once
39
40 #include <Eigen/Dense>  // Eigen linear algebra Library
41 #include <chrono>       // STD Time Measurement Library
42 #include <cmath>        // STD math Library
43 #include <fstream>      // STD File I/O Library
44 #include <iostream>     // STD I/O Library
45 #include <string>       // STD String Library
46 #include <vector>       // STD Vector/Array Library
47
48 //! Typedefs for tire computations routine
49 namespace TireGround {
50
51   typedef double real_type; //!< Real number type
52   typedef int    int_type;  //!< Integer number type
53
54   typedef Eigen::Vector2i vec2_int; //!< 2D vector type of real integer type
55
56   typedef Eigen::Vector2d vec2; //!< 2D vector type of real number type
57   typedef Eigen::Vector3d vec3; //!< 3D vector type of real number type
58   typedef Eigen::Vector4d vec4; //!< 4D vector type of real number type
59   typedef Eigen::Matrix3d mat3; //!< 3x3 matrix type of real number type
60   typedef Eigen::Matrix4d mat4; //!< 4x4 matrix type of real number type
61
62   typedef Eigen::Matrix<real_type,1,Eigen::Dynamic>              row_vecN; //!< Row vector type
       real number type
63   typedef Eigen::Matrix<real_type,Eigen::Dynamic,1>              col_vecN; //!< Column vector type
       real number type
64   typedef Eigen::Matrix<real_type,Eigen::Dynamic,Eigen::Dynamic> matN;     //!< Matrix type of real
        number type
65
66   typedef Eigen::Matrix<vec2,1,Eigen::Dynamic>              row_vec2; //!< Row vector type of 2D
       vector
67   typedef Eigen::Matrix<vec2,Eigen::Dynamic,1>              col_vec2; //!< Column vector type of 2D
        vector
68   typedef Eigen::Matrix<vec2,Eigen::Dynamic,Eigen::Dynamic> mat_vec2; //!< Matrix type of 2D vector
69
70   typedef Eigen::Matrix<vec3,1,Eigen::Dynamic>              row_vec3; //!< Row vector type of 3D
       vector
71   typedef Eigen::Matrix<vec3,Eigen::Dynamic,1>              col_vec3; //!< Column vector type of 3D
        vector
72   typedef Eigen::Matrix<vec3,Eigen::Dynamic,Eigen::Dynamic> matN_vec3; //!< Matrix type of 3D
       vector
73
74   typedef Eigen::Matrix<matN,Eigen::Dynamic,Eigen::Dynamic> matN_mat4; //!<Matrix type of 4x4
       matrix
75
76   typedef std::basic_ostream<char> ostream_type; //!< Output stream type
77
78   real_type const epsilon = std::numeric_limits<real_type>::epsilon(); //!< Epsilon type
79
80   static real_type quietNaN = std::numeric_limits<real_type>::quiet_NaN(); //!< Not-a-Number type
81
82   static mat3 mat3_NaN = mat3::Constant(quietNaN); //!< Not-a-Number 3x3 matrix type
83   static vec3 vec3_NaN = vec3::Constant(quietNaN); //!< Not-a-Number 3D vector type
84
85 }  // namespace TireGround
86
87 ///
88 /// eof: TireGround.hh
89 ///
```

B.2 RoadRDF.hh

```
///
/// file: RoadRDF.hh
///

#pragma once

#include <AABBtree.hh>
#include "TireGround.hh"

// Print progress to console while loading (large models)
#define RDF_CONSOLE_OUTPUT

// Print rdf errors
#ifndef RDF_ERROR
  #define RDF_ERROR(MSG) {                 \
    std::ostringstream ost; ost << MSG;    \
    throw std::runtime_error( ost.str() ); \
  }
#endif

// Check for rdf errors
#ifndef RDF_ASSERT
  #define RDF_ASSERT(COND,MSG) \
    if ( !(COND) ) RDF_ERROR( MSG )
#endif

//! RDF mesh computations routines
namespace RDF {

  using namespace TireGround;

  class TriangleRoad;

  typedef std::shared_ptr<TriangleRoad> TriangleRoad_ptr;  //!< Shared pointer to TriangleRoad
      object
  typedef std::vector<TriangleRoad_ptr> TriangleRoad_list; //!< Vector of shared pointers to
      TriangleRoad objects

  /*\
   |  ____  ____            ____  ____
   | | __ )|  _ )  _____  _|___ \|  _ \
   | |  _ \|  _ \ / _ \ \/ / __) | | | |
   | | |_) | |_) | (_) >  < / __/| |_| |
   | |____/|____/ \___/_/\_\_____|____/
   |
  \*/

  //! 2D Bounding Box class
  class BBox2D {
  private:
    real_type Xmin; //!< Xmin shadow domain point
    real_type Ymin; //!< Ymin shadow domain point
    real_type Xmax; //!< Xmax shadow domain point
    real_type Ymax; //!< Ymax shadow domain point

  public:

    //! Default constructor
    BBox2D() {}

    //! Variable set constructor
    BBox2D(
      vec3 const Vertices[3] //!< Vertices reference vector
```

```
62    ) {
63      updateBBox2D( Vertices );
64    }
65
66    //! Set Xmin shadow domain
67    void
68    setXmin(real_type const _Xmin) { Xmin = _Xmin; }
69
70    //! Set Ymin shadow domain
71    void
72    setYmin(real_type const _Ymin) { Ymin = _Ymin; }
73
74    //! Set Xmax shadow domain
75    void
76    setXmax(real_type const _Xmax) { Xmax = _Xmax; }
77
78    //! Set Ymax shadow domain
79    void
80    setYmax(real_type const _Ymax) { Ymax = _Ymax; }
81
82    //! Get Xmin shadow domain
83    real_type
84    getXmin(void) const { return Xmin; }
85
86    //! Get Ymin shadow domain
87    real_type
88    getYmin(void) const { return Ymin; }
89
90    //! Get Xmax shadow domain
91    real_type
92    getXmax(void) const { return Xmax; }
93
94    //! Get Ymax shadow domain
95    real_type
96    getYmax(void) const { return Ymax; }
97
98    //! Clear the bounding box domain
99    void clear(void);
100
101   //! Print bounding box domain
102   void
103   print(
104     ostream_type & stream //!< Output stream type
105   ) const {
106     stream
107       << "BBOX (xmin,ymin,xmax,ymax) = ( " << Xmin << ", " << Ymin
108       << ", " << Xmax << ", " << Ymax << " )\n";
109   }
110
111   //! Update the bounding box domain with three input vertices
112   void
113   updateBBox2D(
114     vec3 const Vertices[3] //!< Vertices reference vector
115     );
116 };
117
118 /*\
119  |   _____     _                   _      _____ ____
120  |  |_   _| __(_) __ _ _ __   __ _| | ___|___ /|  _ \
121  |    | || '__| |/ _` | '_ \ / _` | |/ _ \ |_ \| | | |
122  |    | || |  | | (_| | | | | (_| | |  __/___) | |_| |
123  |    |_||_|  |_|\__,_|_| |_|\__, |_|\___|____/|____/
124  |                            |___/
125 \*/
126
```

```
127 //! 3D triangle (pure geometrical description)
128 class Triangle3D {
129 protected:
130   vec3   Vertices[3];  //!< Vertices reference vector
131   vec3   Normal;       //!< Triangle normal versor
132   BBox2D TriangleBBox; //!< Triangle 2D bounding box (XY plane)
133
134   Triangle3D( Triangle3D const & ) = delete;               //!< Deleted copy constructor
135   Triangle3D & operator = ( Triangle3D const & ) = delete; //!< Deleted copy operator
136
137 public:
138   //! Variable set constructor
139   Triangle3D() {
140     Vertices[0] = vec3(0,0,0);
141     Vertices[1] = vec3(0,0,0);
142     Vertices[2] = vec3(0,0,0);
143     Normal      = vec3(0,0,0);
144     TriangleBBox.updateBBox2D(Vertices);
145   }
146
147   //! Variable set constructor
148   Triangle3D(
149     vec3 const _Vertices[3] //!< Vertices reference vector
150   ) {
151     Vertices[0] = _Vertices[0];
152     Vertices[1] = _Vertices[1];
153     Vertices[2] = _Vertices[2];
154     TriangleBBox.updateBBox2D(Vertices);
155     calcNormal();
156   }
157
158   //! Set new vertices and update bounding box domain
159   void
160   setVertices(
161     vec3 const _Vertices[3] //!< Vertices reference vector
162   ) {
163     Vertices[0] = _Vertices[0];
164     Vertices[1] = _Vertices[1];
165     Vertices[2] = _Vertices[2];
166     TriangleBBox.updateBBox2D(Vertices);
167     calcNormal();
168   }
169
170   //! Set new vertices then update bounding box domain and normal versor
171   void
172   setVertices(
173     vec3 const & Vertex_0, //!< 1st vertex
174     vec3 const & Vertex_1, //!< 2nd vertex
175     vec3 const & Vertex_2  //!< 3rd vertex
176   ) {
177     Vertices[0] = Vertex_0;
178     Vertices[1] = Vertex_1;
179     Vertices[2] = Vertex_2;
180     TriangleBBox.updateBBox2D(Vertices);
181     calcNormal();
182   }
183
184   //! Get normal versor
185   vec3 const &
186   getNormal(void) const { return Normal; }
187
188   //! Get i-th vertex
189   vec3 const &
190   getVertex( unsigned i ) const { return Vertices[i]; }
191
```

```
192     //! Get Triangle3D bonding box BBox2D
193     BBox2D const &
194     getBBox(void) const { return TriangleBBox; }
195
196     //! Print vertices data
197     void
198     print(
199       ostream_type & stream //!< Output stream type
200     ) const {
201       stream
202         << "V0:\t" << Vertices[0].x() << ", " << Vertices[0].y() << ", " << Vertices[0].z() << "\n"
203         << "V0:\t" << Vertices[1].x() << ", " << Vertices[1].y() << ", " << Vertices[1].z() << "\n"
204         << "V0:\t" << Vertices[2].x() << ", " << Vertices[2].y() << ", " << Vertices[2].z() << "\n"
                ;
205     }
206
207     //! Check if a ray hits a Triangle3D object through Möller-Trumbore
208     //! intersection algorithm
209     bool
210     intersectRay(
211       vec3 const & RayOrigin,    //!< Ray origin position
212       vec3 const & RayDirection, //!< Ray direction vector
213       vec3       & IntPt         //!< Intersection point
214     ) const;
215
216     //! Check if an edge of the Triangle3D object hits a and find the
217     //! intersection point
218     int_type
219     intersectEdgePlane(
220       vec3 const & PlaneN,  //!< Plane normal vector
221       vec3 const & PlaneP,  //!< Plane known point
222       int_type     Edge,    //!< Triangle edge number (0:2)
223       vec3       & IntPt_1, //!< Intersection point 1
224       vec3       & IntPt_2  //!< Intersection point 2
225     ) const;
226
227     //! Check if a plane intersects a Triangle3D object and find the
228     //! intersection points
229     bool
230     intersectPlane(
231       vec3        const & PlaneN, //!< Plane normal vector
232       vec3        const & PlaneP, //!< Plane known point
233       std::vector<vec3> & IntPts  //!< Intersection points
234     ) const;
235
236     private:
237
238     //! Calculate Triangle3D object face normal versor
239     void
240     calcNormal(void) {
241       Normal = ((Vertices[1] - Vertices[0]).cross(Vertices[2] - Vertices[0])).normalized();
242     }
243
244   };
245
246   /*\
247    |   _____     _                    _      ____                  _
248    |  |_   _| __(_) __ _ _ __   __ _| | ___|  _ \ ___   __ _  __| |
249    |    | || '__| |/ _` | '_ \ / _` | |/ _ \ |_) / _ \ / _` |/ _` |
250    |    | || |  | | (_| | | | | (_| | |  __/  _ < (_) | (_| | (_| |
251    |    |_||_|  |_|\__,_|_| |_|\__, |_|\___|_| \_\___/ \__,_|\__,_|
252    |                           |___/
253   \*/
254   //! 3D triangles for road representation
255   class TriangleRoad : public Triangle3D {
```

```
256   private:
257     real_type Friction; //!< Face friction coefficient
258
259     TriangleRoad( TriangleRoad const & ) = delete;               //!< Deleted copy constructor
260     TriangleRoad & operator = ( TriangleRoad const & ) = delete; //!< Deleted copy operator
261
262   public:
263     //! Default set constructor
264     TriangleRoad() : Triangle3D()
265     { Friction = 0.0; }
266
267     //! Variable set constructor
268     TriangleRoad(
269       vec3 const _Vertices[3], //!< Vertices reference vector
270       real_type  _Friction     //!< Friction coefficient
271     ) : Triangle3D( _Vertices ) {
272       Friction = _Friction;
273     }
274
275     //! Set friction coefficient
276     void
277     setFriction(
278       real_type _Friction //!< New friction coefficient
279     ) { Friction = _Friction; }
280
281     //! Get friction coefficent on the face
282     real_type
283     getFriction(void) const { return Friction; }
284   };
285
286   /*\
287    |         _                   _ _   _
288    |    __ _| | __ _  ___  _ __(_) |_| |__  _ __ ___  ___
289    |   / _` | |/ _` |/ _ \| '__| | __| '_ \| '_ ` _ \/ __|
290    |  | (_| | | (_| | (_) | |  | | |_| | | | | | | | \__ \
291    |   \__,_|_|\__, |\___/|_|  |_|\__|_| |_|_| |_| |_|___/
292    |           |___/
293   \*/
294
295   //! Algorithms for RDF mesh computations routine
296   namespace algorithms {
297
298     //! Split a string into a string array at a given token
299     void
300     split(
301       std::string              const & in,   //!< Input string
302       std::vector<std::string>       & out,  //!< Output string vector
303       std::string              const & token //!< Token
304     );
305
306     //! Get tail of string after first token and possibly following spaces
307     std::string
308     tail(
309       std::string const & in //!< Input string
310     );
311
312     //! Get first token of string
313     std::string
314     firstToken(
315       std::string const & in //!< Input string
316     );
317
318     //! Get element at given index position
319     template<typename T>
320     T const &
```

```
321     getElement(
322       std::vector<T> const & elements, //!< Elements vector
323       std::string    const & index     //!< Index position
324     );
325   }  // namespace algorithms
326
327   /*\
328    |   __  __           _     ____              __
329    |  |  \/  | ___  ___| |__ / ___| _   _ _ __ / _| __ _  ___ ___
330    |  | |\/| |/ _ \/ __| '_ \\___ \| | | | '__| |_ / _` |/ __/ _ \
331    |  | |  | |  __/\__ \ | | |___) | |_| | |  |  _| (_| | (_|  __/
332    |  |_|  |_|\___||___/_| |_|____/ \__,_|_|  |_|  \__,_|\___\___|
333    |
334   \*/
335
336   //! Mesh surface
337   class MeshSurface {
338   private:
339     TriangleRoad_list                PtrTriangleVec; //!< Road triangles pointer vector
340     std::vector<G2lib::BBox::PtrBBox> PtrBBoxVec;     //!< Bounding boxes pointers
341     G2lib::AABBtree::PtrAABB          PtrTree;        //!< Mesh tree pointer
342
343     MeshSurface( MeshSurface const & ) = delete;               //!< Deleted copy constructor
344     MeshSurface & operator = ( MeshSurface const & ) = delete; //!< Deleted copy operator
345
346   public:
347     //! Default set constructor
348     MeshSurface()
349     : PtrTree( std::make_shared<G2lib::AABBtree>() )
350     {};
351
352     //! Variable set constructor
353     MeshSurface(
354       TriangleRoad_list const & _PtrTriangleVec //!< Road triangles pointer vector list
355     ) : MeshSurface() {
356       this->PtrTriangleVec = _PtrTriangleVec;
357       updatePtrBBox();
358       PtrTree->build(PtrBBoxVec);
359     };
360
361     //! Variable set constructor
362     MeshSurface(
363       std::string const & Path //!< Path to the RDF file
364     ) : MeshSurface() {
365       bool load = LoadFile(Path);
366       RDF_ASSERT( load, "Error while reading file" );
367     }
368
369     //! Get all triangles inside the mesh as a vector
370     TriangleRoad_list const &
371     getTrianglesList(void) const
372     { return PtrTriangleVec; }
373
374     //! Get i-th TriangleRoad
375     TriangleRoad_ptr const &
376     getTriangle( unsigned i ) const
377     { return PtrTriangleVec[i]; }
378
379     //! Get AABBtree object
380     G2lib::AABBtree::PtrAABB
381     getAABBPtr(void) const
382     { return PtrTree; }
383
384     //! Print data in file
385     void printData( std::string const & FileName );
```

```
386
387     //! Get the mesh G2lib bounding boxes pointers vector
388     std::vector<G2lib::BBox::PtrBBox> const &
389     getPtrBBoxList() const
390     { return PtrBBoxVec; }
391
392     //! Copy the MeshSurface object
393     void
394     set(
395       MeshSurface const & in //!< MeshSurface object to be copied
396     ) {
397       this->PtrTriangleVec = in.PtrTriangleVec;
398       this->PtrBBoxVec     = in.PtrBBoxVec;
399       this->PtrTree        = in.PtrTree;
400     }
401
402     //! Load the RDF model and print information on a file
403     bool
404     LoadFile(
405       std::string const & Path //!< Path to the RDF file
406     );
407
408     //! Update the local intersected TriangleRoad vector list
409     RDF::TriangleRoad_list
410     intersectAABBtree(
411       G2lib::AABBtree::PtrAABB const & ExternalTreePtr //!< External AABBtree object pointer
412     );
413
414   private:
415     //! Update the mesh G2lib bounding boxes pointers
416     void updatePtrBBox(void);
417
418     //! Generate vertices
419     void
420     GenVerticesFromRawRDF(
421       std::vector<vec3> const & iNodes,   //!< Nodes list
422       std::string       const & icurline, //!< Current line
423       vec3                      oVerts[3] //!< Vertex
424     );
425   };
426
427 }  // namespace RDF
428
429 ///
430 /// eof: RoadRDF.hh
431 ///
```

B.3 RoadRDF.cc

```
 1 #include "RoadRDF.hh"
 2
 3 namespace RDF {
 4
 5   /*\
 6    |  ____  ____           ____  ____
 7    | | __ )| __ )  _____  |___ \|  _ \
 8    | |  _ \|  _ \ / _ \ \/ / __) | | | |
 9    | | |_) | |_) | (_) >  < / __/| |_| |
10    | |____/|____/ \___/_/\_\_____|____/
11    |
12   \*/
13
14   void
```

```
15  BBox2D::clear(
16    void
17  ) {
18    Xmin = std::numeric_limits<real_type>::quiet_NaN();
19    Ymin = std::numeric_limits<real_type>::quiet_NaN();
20    Xmax = std::numeric_limits<real_type>::quiet_NaN();
21    Ymax = std::numeric_limits<real_type>::quiet_NaN();
22  }
23
24  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
25
26  void
27  BBox2D::updateBBox2D(
28    vec3 const Vertices[3]
29  ) {
30    G2lib::minmax3( Vertices[0].x(), Vertices[1].x(), Vertices[2].x(), Xmin, Xmax );
31    G2lib::minmax3( Vertices[0].y(), Vertices[1].y(), Vertices[2].y(), Ymin, Ymax );
32  }
33
34  /*\
35   |   _____     _                   _      _____ ____
36   |  |_   _| __(_) __ _ _ __   __ _| | ___|___ /|  _ \
37   |    | || '__| |/ _` | '_ \ / _` | |/ _ \ |_ \| | | |
38   |    | || |  | | (_| | | | | (_| | |  __/___) | |_| |
39   |    |_||_|  |_|\__,_|_| |_|\__, |_|\___|____/|____/
40   |                           |___/
41  \*/
42
43  bool
44  Triangle3D::intersectRay(
45    vec3 const & RayOrigin,
46    vec3 const & RayDirection,
47    vec3       & IntPt
48  ) const {
49    vec3 E1(Vertices[1] - Vertices[0]);
50    vec3 E2(Vertices[2] - Vertices[0]);
51    vec3 A(RayDirection.cross(E2));
52    real_type det = A.dot(E1);
53    real_type t_param;
54
55    if ( det > epsilon ) {
56      vec3 T(RayOrigin - Vertices[0]);
57      real_type u = A.dot(T);
58      if ( u < 0.0 || u > det ) return false;
59      vec3 B(T.cross(E1));
60      real_type v = B.dot(RayDirection);
61      if ( v < 0.0 || u + v > det ) return false;
62      t_param = (B.dot(E2)) / det;
63    } else if (det < -epsilon) {
64      vec3 T(RayOrigin - Vertices[0]);
65      real_type u = A.dot(T);
66      if ( u > 0.0 || u < det ) return false;
67      vec3 B(T.cross(E1));
68      real_type v = B.dot(RayDirection);
69      if ( v > 0.0 || u + v < det ) return false;
70      t_param = (B.dot(E2)) / det;
71    } else {
72      return false;
73    }
74    // At this stage we can compute t to find out where the intersection
75    // point is on the line
76    if ( t_param >= 0 ) {  // ray intersection
77      IntPt = RayOrigin + RayDirection * t_param;
78      return true;
79    } else {
```

```
80      // This means that there is a line intersection on negative side
81      return false;
82    }
83  }
84
85  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
86
87  int_type
88  Triangle3D::intersectEdgePlane(
89    vec3     const & PlaneN,
90    vec3     const & PlaneP,
91    int_type         Edge,
92    vec3           & IntPt_1,
93    vec3           & IntPt_2
94  ) const {
95    // Check that edge number is between 0 and 2
96    RDF_ASSERT( (Edge >=0 && Edge <= 2) , "Side number must be between 0 and 2.")
97    // If it does not lays
98    vec3 const & PointA = Vertices[Edge];
99    vec3 const & PointB = Vertices[(Edge+1) % 3]; // Operatore modulo
100   vec3 Direction(PointB - PointA);
101
102   // Check if the segment lays on the plane
103   if ( (PlaneN - Normal).norm() < epsilon &&
104        Direction.dot(PlaneN) < epsilon ) {
105     IntPt_1 = PointA;
106     IntPt_2 = PointB;
107     return 2;
108   }
109   // If it does not lay on the plane, then find the single point
110   real_type d = -PlaneP.dot(PlaneN);
111   real_type t = -(PointA.dot(PlaneN) + d) / (Direction.dot(PlaneN));
112   if ( t >= 0 && t <= 1 ) {
113     IntPt_1 = PointA + Direction * t;
114     return 1;
115   } else {
116     return 0;
117   }
118 }
119
120 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
121
122 bool
123 Triangle3D::intersectPlane(
124   vec3        const & PlaneN,
125   vec3        const & PlaneP,
126   std::vector<vec3> & IntPts
127 ) const {
128   // Clear intersection points vector
129   IntPts.clear();
130   // Check if the triangle lays on the plane
131   if ( (Normal-PlaneN).norm() < epsilon &&
132        (Vertices[0]-PlaneP).dot(PlaneN) < epsilon ) {
133     IntPts.push_back(Vertices[0]);
134     IntPts.push_back(Vertices[1]);
135     IntPts.push_back(Vertices[2]);
136     return true;
137   } else {
138     int_type number;
139     vec3 IntPt1, IntPt2;
140     // Fill intersection points vector and check the results (there must be only 2 intersection
          points)
141     for (int_type i = 0; i < 3; i++)
142     {
143       number = intersectEdgePlane( PlaneN, PlaneP, i, IntPt1, IntPt2 );
```

```
144         switch (number) {
145         case 0:
146           break;
147         case 1:
148           IntPts.push_back(IntPt1);
149           break;
150         case 2:
151           IntPts.push_back(IntPt1);
152           IntPts.push_back(IntPt2);
153           break;
154         default:
155           RDF_ERROR("\nTriangle-Plane intersection not handled.")
156           break;
157         }
158       }
159       if ( IntPts.size() == 2 ) {
160         return true;
161       } else if ( IntPts.size() == 0 ) {
162         return false;
163       } else {
164         RDF_ERROR(
165           "\nTriangle-Plane intersection not handled. There are " <<
166           IntPts.size() << " intersections."
167         );
168         return false;
169       }
170     }
171   }
172
173   /*\
174    |         _                    _ _   _
175    |    __ _| | __ _  ___  _ __(_) |_| |__  _ __ ___  ___
176    |   / _` | |/ _` |/ _ \| '__| | __| '_ \| '_ ` _ \/ __|
177    |  | (_| | | (_| | (_) | |  | | |_| | | | | | | | \__ \
178    |   \__,_|_|\__, |\___/|_|  |_|\__|_| |_|_| |_| |_|___/
179    |           |___/
180   \*/
181
182   namespace algorithms {
183
184     void
185     split(
186       std::string const        & in,
187       std::vector<std::string> & out,
188       std::string const        & token
189     ) {
190       out.clear();
191       std::string temp;
192
193       for ( int i = 0; i < int(in.size()); ++i ) {
194         std::string test = in.substr(i, token.size());
195         if (test == token) {
196           if (!temp.empty()) {
197             out.push_back(temp);
198             temp.clear();
199             i += (int)token.size() - 1;
200           } else {
201             out.push_back("");
202           }
203         } else if (i + token.size() >= in.size()) {
204           temp += in.substr(i, token.size());
205           out.push_back(temp);
206           break;
207         } else {
208           temp += in[i];
```

```
209       }
210     }
211   }
212
213   // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
214
215   std::string
216   tail(
217     std::string const & in
218   ) {
219     size_t token_start = in.find_first_not_of(" \t");
220     size_t space_start = in.find_first_of(" \t", token_start);
221     size_t tail_start  = in.find_first_not_of(" \t", space_start);
222     size_t tail_end    = in.find_last_not_of(" \t");
223     if (tail_start != std::string::npos && tail_end != std::string::npos) {
224       return in.substr(tail_start, tail_end - tail_start + 1);
225     } else if (tail_start != std::string::npos) {
226       return in.substr(tail_start);
227     }
228     return "";
229   }
230
231   // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
232
233   std::string
234   firstToken(
235     std::string const & in
236   ) {
237     if (!in.empty()) {
238       size_t token_start = in.find_first_not_of(" \t\r\n");
239       if (token_start != std::string::npos) {
240         size_t token_end = in.find_first_of(" \t\r\n", token_start);
241         if (token_end != std::string::npos) {
242           return in.substr(token_start, token_end - token_start);
243         } else {
244           return in.substr(token_start);
245         }
246       }
247     }
248     return "";
249   }
250
251   // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
252
253   template<typename T>
254   T const &
255   getElement(
256     std::vector<T> const & elements,
257     std::string    const & index
258   ) {
259     int_type id = std::stoi(index);
260     if ( id < 0 ) std::cerr << "ELEMENTS indexes cannot be negative\n";
261     return elements[id - 1];
262   }
263
264 }  // namespace algorithms
265
266 /*\
267  |   __  __           _     ____                __
268  |  |  \/  | ___  ___| |__ / ___| _   _ _ __ / _| __ _  ___ ___
269  |  | |\/| |/ _ \/ __| '_ \\___ \| | | | '__| |_ / _` |/ __/ _ \
270  |  | |  | |  __/\__ \ | | |___) | |_| | |  |  _| (_| | (_|  __/
271  |  |_|  |_|\___||___/_| |_|____/ \__,_|_|  |_|  \__,_|\___\___|
272  |
273 \*/
```

```
274
275 void
276 MeshSurface::printData(
277   std::string const & FileName
278 ) {
279   // Create Out.txt
280   std::ofstream file(FileName);
281   // Print introduction
282   file
283     << "LOADED RDF MESH DATA\n\n"
284     << "Legend:\n"
285     << "\tVi: i-th vertex\n"
286     << "\t N: normal to the face\n"
287     << "\t F: friction coefficient\n\n";
288   for ( unsigned i = 0; i < PtrTriangleVec.size(); ++i ) {
289     TriangleRoad const & Ti = *PtrTriangleVec[i];
290     vec3 const & V0 = Ti.getVertex(0);
291     vec3 const & V1 = Ti.getVertex(1);
292     vec3 const & V2 = Ti.getVertex(2);
293     vec3 const &  N = Ti.getNormal();
294     // Print vertices, normal and friction
295     file
296       << "TRIANGLE " << i
297       << "\n\tV0:\t" << V0.x() << ", " << V0.y() << ", " << V0.z()
298       << "\n\tV1:\t" << V1.x() << ", " << V1.y() << ", " << V1.z()
299       << "\n\tV2:\t" << V2.x() << ", " << V2.y() << ", " << V2.z()
300       << "\n\t N:\t" << N.x()  << ", " << N.y()  << ", " << N.z()
301       << "\n\t F:\t" << Ti.getFriction()
302       << "\n\n";
303   }
304   // Close File
305   file.close();
306 }
307
308 // . . . . . .
309
310 void
311 MeshSurface::updatePtrBBox(
312   void
313 ) {
314   PtrBBoxVec.clear();
315   RDF::BBox2D iBBox;
316   for (unsigned id = 0; id < PtrTriangleVec.size(); ++id) {
317     iBBox = (*PtrTriangleVec[id]).getBBox();
318     PtrBBoxVec.push_back(G2lib::BBox::PtrBBox(
319       new G2lib::BBox(
320         iBBox.getXmin(),
321         iBBox.getYmin(),
322         iBBox.getXmax(),
323         iBBox.getYmax(),
324         id, 0
325       )
326     ));
327     iBBox.clear();
328   }
329 }
330
331 // . . . . . .
332
333 bool
334 MeshSurface::LoadFile(
335   std::string const & Path
336 ) {
337   // Check if the file is an ".rdf" file, if not return false
338   if (Path.substr(Path.size() - 4, 4) != ".rdf") {
```

```
339       std::cerr << "Not a RDF file\n";
340       return false;
341     }
342      // Check if the file had been correctly open, if not return false
343     std::ifstream file(Path);
344     if (!file.is_open()) {
345       std::cerr << "RDF file not opened\n";
346       return false;
347     }
348     // Vector for nodes coordinates
349     std::vector<vec3> Nodes;
350     bool nodes_parse    = false;
351     bool elements_parse = false;
352
353 #ifdef RDF_CONSOLE_OUTPUT
354     int_type const outputEveryNth = 5000;
355     int_type outputIndicator      = outputEveryNth;
356 #endif
357     std::string curline;
358     while (std::getline(file, curline)) {
359 #ifdef RDF_CONSOLE_OUTPUT
360       if ((outputIndicator = ((outputIndicator + 1) % outputEveryNth)) == 1) {
361         std::cout
362           << "\r- "
363           << "Loading mesh..."
364           << "\t triangles > "
365           << PtrTriangleVec.size() << std::endl;
366       }
367 #endif
368       std::string token = algorithms::firstToken(curline);
369       if ( token == "[NODES]" || token == "NODES" ) {
370         nodes_parse    = true;
371         elements_parse = false;
372         continue;
373       } else if (token == "[ELEMENTS]" || token == "ELEMENTS") {
374         nodes_parse    = false;
375         elements_parse = true;
376         continue;
377       } else if (token[0] == '{') {
378         // commento multiriga, continua a leggere fino a che trovo '}'
379         continue;
380       } else if (token[0] == '%' || token[0] == '#' || token[0] == '\r') {
381         // Check comments or empty lines
382         continue;
383       }
384       // Generate a vertex position
385       if (nodes_parse) {
386         std::vector<std::string> spos;
387         vec3 vpos;
388         algorithms::split(algorithms::tail(curline), spos, " ");
389         vpos[0] = std::stod(spos[0]);
390         vpos[1] = std::stod(spos[1]);
391         vpos[2] = std::stod(spos[2]);
392         Nodes.push_back(vpos);
393       }
394       // Generate a face (vertices & indices)
395       if (elements_parse) {
396         // Generate the triangle vertices from the elements
397         vec3 iVerts[3];
398         GenVerticesFromRawRDF( Nodes, curline, iVerts );
399         // Get the triangle friction from current line
400         std::vector<std::string> curlinevec;
401         algorithms::split(curline, curlinevec, " ");
402         real_type iFriction = std::stod(curlinevec[4]);
403         // Create a shared pointer for the last triangle and push it in the pointer vector
```

```
404         PtrTriangleVec.push_back(TriangleRoad_ptr(new TriangleRoad(iVerts,iFriction)));
405       }
406     }
407 #ifdef RDF_CONSOLE_OUTPUT
408     std::cout << std::endl;
409 #endif
410     file.close();
411     if (PtrTriangleVec.empty()) {
412       perror("Loaded mesh is empty");
413       return false;
414     } else {
415       // Update the local intersected triangles list
416 #ifdef RDF_CONSOLE_OUTPUT
417     std::cout << std::endl << "Building AABB tree... ";
418 #endif
419       updatePtrBBox();
420       PtrTree->build(PtrBBoxVec);
421 #ifdef RDF_CONSOLE_OUTPUT
422     std::cout << "Done" << std::endl << std::endl;
423 #endif
424       return true;
425     }
426   }
427
428   // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
429
430   void
431   MeshSurface::GenVerticesFromRawRDF(
432     std::vector<vec3> const & iNodes,
433     std::string       const & icurline,
434     vec3                      oVerts[3]
435   ) {
436     std::vector<std::string> svert;
437     vec3                     vVert;
438     algorithms::split( icurline, svert, " " );
439
440     int_type control_size = int(svert.size() - 4);
441     for ( int i = 1; i < int(svert.size() - control_size); ++i ) {
442       // Calculate and store the vertex
443       vVert = algorithms::getElement(iNodes, svert[i]);
444       oVerts[i-1] = vVert;
445     }
446   }
447
448   // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
449
450   RDF::TriangleRoad_list
451   MeshSurface::intersectAABBtree(
452     G2lib::AABBtree::PtrAABB const & ExternalTreePtr
453   ) {
454     RDF::TriangleRoad_list intersectionTriPtr;
455     G2lib::AABBtree::VecPairPtrBBox intersectionList;
456     PtrTree->intersect(*ExternalTreePtr, intersectionList);
457     for ( unsigned i = 0; i < intersectionList.size(); ++i ) {
458       intersectionTriPtr.push_back(
459         getTriangle((*intersectionList[i].first).Id()));
460     }
461     return intersectionTriPtr;
462   }
463
464 }  // namespace RDF
```

B.4 PatchTire.hh

```
///
/// file: PatchTire.hh
///

#pragma once

#include "RoadRDF.hh" // RDF file extention loader

//! Tire computations routines
namespace PatchTire {

  using namespace TireGround;

  /*\
   |   ____  _     _
   |  |  _ \(_)___| | __
   |  | | | | / __| |/ /
   |  | |_| | \__ \   <
   |  |____/|_|___/_|\_\
   |
   |  ^ Z
   |  |     -----
   |  |   /       \
   |  |  |    O    |  O = OriginXZ
   |  |   \       /
   |  |     -----
   |  Y------------------> X
   |
   |  ISO Reference Frame
  \*/

  class ReferenceFrame;
  class ETRTO;

  //! Tire disk
  class Disk {
  private:
    vec2      OriginXZ; //!< X0,Z0 origin vector
    real_type OffsetY;  //!< Y0 (= D) origin coordinate (offset from center)
    real_type Radius;   //!< Radius

    Disk( Disk const & ) = delete;                     //!< Deleted copy constructor
    Disk const & operator = ( Disk const & ) = delete; //!< Deleted copy operator

  public:
    //! Enable && operator
    Disk( Disk && ) = default;

    //! Default constructor
    Disk()
    : OriginXZ( vec2(quietNaN, quietNaN) )
    , OffsetY( quietNaN )
    , Radius( quietNaN )
    {}

    //! Variable set constructor
    Disk(
      vec2 const & _OriginXZ, //!< X0,Z0 origin coordinate
      real_type    _OffsetY,  //!< Y0 (= D) origin coordinate (offset from center)
      real_type    _Radius    //!< Radius
    ) {
      OriginXZ = _OriginXZ;
      OffsetY  = _OffsetY;
      Radius   = _Radius;
    }
```

```
66
67    //! Copy the Disk object
68    void
69    set(
70      Disk const & in //!< Disk object to be copied
71    ) {
72      this->OriginXZ = in.OriginXZ;
73      this->OffsetY  = in.OffsetY;
74      this->Radius   = in.Radius;
75    }
76
77    //! Set origin on XZ plane
78    void
79    setOriginXZ(
80      vec2 const & _OriginXZ //!< New origin on XZ plane
81    ) {
82      OriginXZ = _OriginXZ;
83    }
84
85    //! Get origin vector XZ-axes coordinates
86    vec2 const & getOriginXZ(void) const { return OriginXZ; }
87
88    //! Get origin vector XYZ-axes coordinates
89    vec3 getOriginXYZ(void) const
90    { return vec3(OriginXZ.x(), OffsetY, OriginXZ.y()); }
91
92    //! Get origin Y-axis coordinate
93    real_type getOffsetY(void) const { return OffsetY; }
94
95    //! Get Disk radius
96    real_type getRadius(void) const { return Radius; }
97
98    //! Check if a point is inside or outside a circumfererence
99    bool
100   isPointInside(
101     vec2 const & Point //!< Query point
102   ) const;
103
104   //! Find the intersection points between a circle and a line segment
105   //! (output integer gives number of intersection points)
106   int_type
107   intersectSegment(
108     vec2 const & Point_1,     //!< Line segment point 1
109     vec2 const & Point_2,     //!< Line segment point 2
110     vec2       & Intersect_1, //!< Intersection point 1
111     vec2       & Intersect_2  //!< Intersection point 2
112   ) const;
113
114   //! Check if two plane intersects and find the intersecting rect
115   bool
116   intersectPlane(
117     vec3 const & Plane_Normal,   //!< Plane normal vector in Disk reference frame
118     vec3 const & Plane_Point,    //!< Plane known point in Disk reference frame
119     vec3       & Line_Direction, //!< Rect direction vector in Disk reference frame
120     vec3       & Line_Point      //!< Plane known point in Disk reference frame
121   ) const;
122
123   //! Get the contact patch length inside the single disk of a segment described by
124   //! the intersection of triangles on XZ plane
125   real_type
126   getPatchLength(
127     RDF::TriangleRoad_list const & intersectionTriPtr, //!< Shadow/MeshSurface intersected
          triangles
128     ReferenceFrame         const & RF                  //!< Tire ReferenceFrame
129   ) const;
```

```
130
131   //! Get the contact patch length inside the single disk [m] given a plane
132   real_type
133   getPatchLength(
134     vec3           const & Plane_Normal, //!< Plane normal in Disk reference frame
135     vec3           const & Plane_Point,  //!< Plane point in Disk reference frame
136     ReferenceFrame const & RF            //!< Tire ReferenceFrame
137   ) const;
138
139   //! Get the contact patch length inside the single disk [m] given two points
140   //! in Disk reference frame
141   real_type
142   getPatchLength(
143     vec2           const & PointXZ_1, //!< Point 1 in Disk reference frame
144     vec2           const & PointXZ_2, //!< Point 2 in Disk reference frame
145     ReferenceFrame const & RF         //!< Tire ReferenceFrame
146   ) const;
147
148   //! Get the contact patch length inside the single disk [m] give a sequence
149   //! points in Disk reference frame
150   real_type
151   getPatchLength(
152     col_vec2       const & XZ_sequence, //!< Points sequence in Disk RF
153     ReferenceFrame const & RF           //!< Tire ReferenceFrame
154   ) const;
155
156 };
157
158 /*\
159  |   _____ _____ ____ _____ ___
160  |  | ____|_   _|  _ \_   _/ _ \
161  |  |  _|   | | | |_) || || | | |
162  |  | |___  | | |  _ < | || |_| |
163  |  |_____| |_| |_| \_\|_| \___/
164 \*/
165
166 //! Tire ETRTO denomination
167 class ETRTO {
168 private:
169   real_type SectionWidth;   //!< Tire section width [mm]
170   real_type AspectRatio;    //!< Tire aspect ratio [%]
171   real_type RimDiameter;    //!< Rim diameter [in]
172   real_type SidewallHeight; //!< Sidewall height [m]
173   real_type TireDiameter;   //!< External diameter [m]
174
175 public:
176   //! Default constructor
177   ETRTO() {}
178
179   //! Variable set constructor
180   ETRTO(
181     real_type _SectionWidth, //!< Tire section width [mm]
182     real_type _AspectRatio,  //!< Tire aspect ratio [%]
183     real_type _RimDiameter   //!< Rim diameter [in]
184   ) {
185     SectionWidth = _SectionWidth;
186     AspectRatio  = _AspectRatio;
187     RimDiameter  = _RimDiameter;
188     calcSidewallHeight();
189     calcTireDiameter();
190   }
191
192   //! Get sidewall height [m]
193   real_type getSidewallHeight(void) const { return SidewallHeight; }
194
```

```
195   //! Get external tire diameter [m]
196   real_type getTireDiameter(void) const { return TireDiameter; }
197
198   //! Get external tire radius [m]
199   real_type getTireRadius(void) const { return TireDiameter / 2.0; }
200
201   //! Get section width [m]
202   real_type getSectionWidth(void) const { return SectionWidth / 1000.0; }
203
204   //! Display tire data
205   void
206   print(
207     ostream_type & stream //!< Output stream type
208   ) const {
209     stream
210       << "Current Tire Data:\n"
211       << "\tSection width = " << SectionWidth << " mm\n"
212       << "\tAspect ratio  = " << AspectRatio << " %\n"
213       << "\tRim diameter  = " << RimDiameter << " in\n"
214       << "\tS.wall Height = " << getSidewallHeight() * 1000 << " mm\n"
215       << "\tTire diameter = " << getTireDiameter() * 1000 << " mm\n\n";
216   }
217
218 private:
219   //! Calculate sidewall height [m]
220   void
221   calcSidewallHeight(void)
222   { SidewallHeight = SectionWidth / 1000.0 * AspectRatio / 100; }
223
224   //! Calculate external tire diameter [m]
225   void
226   calcTireDiameter(void)
227   { TireDiameter = RimDiameter * 0.0254 + getSidewallHeight() * 2.0; }
228 };
229
230 /*\
231  |   ____       __                                 _____
232  |  |  _ \ ___ / _| ___ _ __ ___ _ __   ___ ___|  ___| __ __ _ _ __ ___   ___
233  |  | |_) / _ \ |_ / _ \ '__/ _ \ '_ \ / __/ _ \ |_  | '__/ _` | '_ ` _ \ / _ \
234  |  |  _ <  __/  _|  __/ | |  __/ | | | (_|  __/  _| | | | (_| | | | | | |  __/
235  |  |_| \_\___|_|  \___|_|  \___|_| |_|\___\___|_|  |_|  \__,_|_| |_| |_|\___|
236  |
237 \*/
238
239 //! Reference frame
240 class ReferenceFrame {
241 private:
242   vec3 Origin;         //!< Origin position
243   mat3 RotationMatrix; //!< 3x3 rotation matrix
244
245   ReferenceFrame( ReferenceFrame const & ) = delete;               //!< Deleted copy constructor
246   ReferenceFrame & operator = ( ReferenceFrame const & ) = delete; //!< Deleted copy operator
247
248 public:
249   //! Default constructor
250   ReferenceFrame()
251   : Origin(vec3_NaN)
252   , RotationMatrix(mat3_NaN)
253   {}
254
255   //! Variable set constructor
256   ReferenceFrame(
257     vec3 const & _Origin,        //!< Origin position
258     mat3 const & _RotationMatrix //!< 3x3 rotation matrix
259   )
```

```
260     : Origin(_Origin)
261     , RotationMatrix(_RotationMatrix)
262     { }
263
264     //! Check if ReferenceFrame object is empty
265     bool
266     isEmpty(void) {
267       if ( Origin != Origin || RotationMatrix != RotationMatrix){
268         return true;
269       } else {
270         return false;
271       }
272     }
273
274     //! Get current 3x3 rotation matrix
275     mat3 const & getRotationMatrix(void) const { return RotationMatrix; }
276
277     //! Get current rotation matrix inverse with LU decomposition
278     vec3 getRotationMatrixInverse(vec3 const & Point) const {
279       // DA INDAGARE https://eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html
280       Eigen::PartialPivLU<RDF::mat3> RF_LU(RotationMatrix);
281       return RF_LU.solve(Point);
282     }
283
284     //! Get current X-axis versor
285     vec3 getX(void) const { return RotationMatrix.col(0); }
286
287     //! Get current Y-axis versor
288     vec3 getY(void) const { return RotationMatrix.col(1); }
289
290     //! Get current Z-axis versor
291     vec3 getZ(void) const { return RotationMatrix.col(2); }
292
293     //! Get origin position
294     vec3 const & getOrigin(void) const { return Origin; }
295
296     //! Set origin position
297     void setOrigin(
298       vec3 const & _Origin //!< Origin position
299     ) { Origin = _Origin; }
300
301     //! Set 3x3 rotation matrix
302     void setRotationMatrix(
303       mat3 const & _RotationMatrix //!< 3x3 rotation matrix
304     ) { RotationMatrix = _RotationMatrix; }
305
306     //! Set 4x4 total transformation matrix
307     void
308     setTotalTransformationMatrix(
309       mat4 const & TM //!< 4x4 total transformation matrix
310     ) {
311       Origin         = TM.block<3,1>(0,3);
312       RotationMatrix = TM.block<3,3>(0,0);
313     }
314
315     //! Get 4x4 total transformation matrix
316     mat4
317     getTotalTransformationMatrix(void) {
318       mat4 out;
319       out << RotationMatrix, Origin, vec4(0.0, 0.0, 0.0, 1.0).transpose();
320       return out;
321     }
322
323     //! Copy the tire ReferenceFrame object
324     void
```

```
325   set(
326     ReferenceFrame const & in //!< ReferenceFrame object to be copied
327   ) {
328     this->Origin         = in.Origin;
329     this->RotationMatrix = in.RotationMatrix;
330   }
331
332   //! Get current Euler angles [rad] for X-axis \n
333   //! Warning: Factor as Rz*Rx*Ry!
334   // https://www.geometrictools.com/Documentation/EulerAngles.pdf
335   real_type getEulerAngleX(void) const;
336
337   //! Get current Euler angles [rad] for Y-axis \n
338   //! Warning: Factor as Rz*Rx*Ry!
339   // https://www.geometrictools.com/Documentation/EulerAngles.pdf
340   real_type getEulerAngleY(void) const;
341
342   //! Get current Euler angles [rad] for Z-axis \n
343   //! Warning: Factor as Rz*Rx*Ry!
344   // https://www.geometrictools.com/Documentation/EulerAngles.pdf
345   real_type getEulerAngleZ(void) const;
346
347 };
348
349 /*\
350  |   ____  _                 _
351  |  / ___|| |__   __ _  __| | _____      __
352  |  \___ \| '_ \ / _` |/ _` |/ _ \ \ /\ / /
353  |   ___) | | | | (_| | (_| | (_) \ V  V /
354  |  |____/|_| |_|\__,_|\__,_|\___/ \_/\_/
355 \*/
356
357 //! 2D shadow (2D bounding box enhacement)
358 class Shadow {
359 private:
360   real_type Xmin;                                //!< Xmin shadow domain point
361   real_type Ymin;                                //!< Ymin shadow domain point
362   real_type Xmax;                                //!< Xmax shadow domain point
363   real_type Ymax;                                //!< Ymax shadow domain point
364   std::vector<G2lib::BBox::PtrBBox> PtrBBoxVec; //!< Vector of shared pointers to BBox objects
365   G2lib::AABBtree::PtrAABB PtrTree =
366       std::make_shared<G2lib::AABBtree>();      //!< Shared pointer to Mesh AABBtree
367
368   Shadow( Shadow const & ) = delete;                    //!< Deleted copy constructor
369   Shadow const & operator = ( Shadow const & ) = delete; //!< Deleted copy operator
370
371 public:
372   //! Default constructor
373   Shadow() {}
374
375   //! Variable set constructor
376   Shadow(
377     ETRTO          const & TireGeometry, //!< Tire ETRTO denomination
378     ReferenceFrame const & RF            //!< Tire ReferenceFrame
379   ) {
380     update( TireGeometry, RF );
381   }
382
383   //! Get shared pointer  to Shadow oject
384   G2lib::AABBtree::PtrAABB
385   getAABBPtr(void) const {
386     std::vector<G2lib::BBox::PtrBBox> PtrBBoxVec;
387     G2lib::AABBtree::PtrAABB PtrTree = std::make_shared<G2lib::AABBtree>();
388     updatePtrBBox( PtrBBoxVec );
389     PtrTree->build(PtrBBoxVec);
```

```
390       return PtrTree;
391     }
392
393     //! Get Xmin shadow domain point
394     real_type getXmin(void) const { return Xmin; }
395
396     //! Get Ymin shadow domain point
397     real_type getYmin(void) const { return Ymin; }
398
399     //! Get Xmax shadow domain point
400     real_type getXmax(void) const { return Xmax; }
401
402     //! Get Ymax shadow domain point
403     real_type getYmax(void) const { return Ymax; }
404
405     //! Clear the tire shadow domain
406     void
407     clear(void) {
408       Xmin = quietNaN;
409       Ymin = quietNaN;
410       Xmax = quietNaN;
411       Ymax = quietNaN;
412     }
413
414     //! Print tire shadow data
415     void
416     print(
417       ostream_type & stream //!< Output stream type
418     ) const {
419       stream
420         << "BBOX (xmin,ymin,xmax,ymax) = ( " << Xmin << ", " << Ymin
421         << ", " << Xmax << ", " << Ymax << " )\n";
422     }
423
424     //! Update the 2D tire shadow domain
425     void
426     update(
427       ETRTO          const & TireGeometry, //!< Tire ETRTO denomination
428       ReferenceFrame const & RF            //!< Tire ReferenceFrame
429     );
430
431   private:
432     //! Update the vector of shared pointers to BBox objects
433     void
434     updatePtrBBox(
435       std::vector<G2lib::BBox::PtrBBox> & PtrBBoxVec //!< Bounding boxes pointers
436     ) const {
437       PtrBBoxVec.resize(1);
438       PtrBBoxVec[0] = G2lib::BBox::PtrBBox(
439         new G2lib::BBox(Xmin, Ymin, Xmax, Ymax, 0, 0)
440       );
441     }
442   };
443
444   /*\
445    |   ____                        _ _              ____      _     _
446    |  / ___|  __ _ _ __ ___  _ __ | (_)_ __   __ _ / ___|_ __(_) __| |
447    |  \___ \ / _` | '_ ` _ \| '_ \| | | '_ \ / _` | |  _| '__| |/ _` |
448    |   ___) | (_| | | | | | | |_) | | | | | | (_| | |_| | |  | | (_| |
449    |  |____/ \__,_|_| |_| |_| .__/|_|_|_| |_|\__, |\____|_|  |_|\__,_|
450    |                       |_|              |___/
451   \*/
452
453   //! Patch evaluation precision
454   class SamplingGrid {
```

```
455  private:
456    int_type Xdiv; //!< Number of points in X-axis (sampling points for each disk)
457    int_type Ydiv; //!< Number of points in Y-axis (number of disks)
458
459    SamplingGrid( SamplingGrid const & ) = delete;               //!< Deleted copy constructor
460    SamplingGrid & operator = ( SamplingGrid const & ) = delete; //!< Deleted copy operator
461
462  public:
463    //! Default constructor
464    SamplingGrid()
465    : Xdiv(quietNaN)
466    , Ydiv(quietNaN)
467    {}
468
469    //! Variable set constructor
470    SamplingGrid(
471      int_type _Xdiv, //!< Number of points in X-axis (sampling points for each disk)
472      int_type _Ydiv  //!< Number of points in Y-axis (number of disks)
473    )
474    : Xdiv(_Xdiv)
475    , Ydiv(_Ydiv)
476    {
477      RDF_ASSERT( _Xdiv > 0, "Sampling points on X-axis cannot be less than 1.");
478      RDF_ASSERT( _Ydiv > 0, "Sampling points on Y-axis cannot be less than 1.");
479    }
480
481    //! Get number of divisions in X-axis
482    int_type getXdiv(void) const { return Xdiv; }
483
484    //! Get number of divisions in Y-axis
485    int_type getYdiv(void) const { return Ydiv; }
486
487    //! Set number of divisions
488    void
489    set(
490      int_type _Xdiv,
491      int_type _Ydiv
492    ) {
493      Xdiv = _Xdiv;
494      Ydiv = _Ydiv;
495      RDF_ASSERT( _Xdiv > 1,"Sampling points on X-axis cannot be less than 2.");
496      RDF_ASSERT( _Ydiv > 1,"Sampling points on Y-axis cannot be less than 2.");
497    }
498
499    //! Copy the SamplingGrid object
500    void
501    set(
502      SamplingGrid const & in //!< SamplingGrid object to be copied
503    ) {
504      this->Xdiv = in.Xdiv;
505      this->Ydiv = in.Ydiv;
506    }
507
508  };
509
510  /*\
511   |         _                    _ _    _
512   |    __ _| | __ _  ___  _ __(_) |_| |__  _ __ ___  ___
513   |   / _` | |/ _` |/ _ \| '__| | __| '_ \| '_ ` _ \/ __|
514   |  | (_| | | (_| | (_) | |  | | |_| | | | | | | | \__ \
515   |   \__,_|_|\__, |\___/|_|  |_|\__|_| |_|_| |_| |_|___/
516   |           |___/
517  \*/
518
519  //! Algorithms for tire computations routine
```

```
520 namespace algorithms {
521
522   //! Check if a point lays inside or outside a line segment \n
523   //! Warning: The point query point must be on the same rect of the line segment!
524   bool
525   intersectPointSegment(
526     vec2 const & Point_1, //!< Line segment point 1
527     vec2 const & Point_2, //!< Line segment point 2
528     vec2 const & Point    //!< Query point
529   );
530
531   //! Check if a segment hits a plane and find the intersection point
532   bool
533   intersectRayPlane(
534     vec3 const & planeN,        //!< Plane normal vector
535     vec3 const & planeP,        //!< Plane known point
536     vec3 const & RayPoint,      //!< Ray point
537     vec3 const & RayDirection,  //!< Ray direction
538     vec3       & IntersectionPt //!< Intersection point
539   );
540
541 }  // namespace algorithms
542
543 /*\
544  |   _____ _
545  |  |_   _(_)_ __ ___
546  |    | | | | '__/ _ \
547  |    | | | | | |  __/
548  |    |_| |_|_|  \___|
549  |
550  |
551  |          ^ Z
552  |          |
553  |        --|--
554  |      /   |   \
555  |     |  O=Y----|---> X
556  |      \       /
557  |        -----
558  |
559  |          ^ Y
560  |          |
561  |      ____|____
562  |     |    |    |
563  |     |  O=Z----|---> X
564  |     |_________|
565  |
566  |  O = OriginXZ
567  |  ISO Reference Frame!
568 \*/
569
570 //! Base class for Tire models
571 class Tire {
572 protected:
573   SamplingGrid   Precision;    //!< Solver precision
574   ETRTO          TireGeometry; //!< Tire ETRTO denomination
575   ReferenceFrame RF;           //!< ReferenceFrame
576   int_type       TirePose;     //!< Flag for tire position wrt the mesh (out = 0, in = 1, in air
          = 2)
577
578   Tire( Tire const & ) = delete;                    //!< Deleted copy constructor
579   Tire const & operator = ( Tire const & ) = delete; //!< Deleted copy operator
580
581 public:
582   //! Default destructor
583   ~Tire() {};
```

```
584
585     //! Variable set constructor
586     Tire(
587       ETRTO    const & Geometry, //!< Tire ETRTO denomination
588       int_type const   Xdiv,     //!< Number of points in X-axis (sampling points for each disk)
589       int_type const   Ydiv      //!< Number of points in Y-axis (number of disks)
590     )
591     : Precision( Xdiv, Ydiv ),
592       TireGeometry( Geometry ),
593       RF()
594     {}
595
596     //! Variable set constructor
597     Tire(
598       real_type const SectionWidth, //!< Tire section width [mm]
599       real_type const AspectRatio,  //!< Tire aspect ratio [%]
600       real_type const RimDiameter,  //!< Rim diameter [in]
601       int_type  const Xdiv,         //!< Number of points in X-axis (sampling points for each disk)
602       int_type  const Ydiv          //!< Number of points in Y-axis (number of disks)
603     )
604     : Precision( Xdiv, Ydiv ),
605       TireGeometry( SectionWidth, AspectRatio, RimDiameter ),
606       RF()
607     {}
608
609     //! Display tire data
610     void
611     print(
612       ostream_type & stream //!< Output stream type
613     ) const {
614       TireGeometry.print( stream );
615     }
616
617     //! Copy the tire ReferenceFrame object
618     void
619     setReferenceFrame(
620       ReferenceFrame const & _RF //!< ReferenceFrame object to be copied
621     ) { RF.set(_RF); }
622
623     //! Get tire ReferenceFrame object
624     ReferenceFrame const &
625     getReferenceFrame(void) const
626     { return RF; }
627
628     //! Set a new tire origin
629     void
630     setOrigin(
631       vec3 const & Origin //!< Tire origin
632     ) { RF.setOrigin( Origin ); }
633
634     //! Set a new 3x3 rotation matrix
635     void
636     setRotationMatrix(
637       mat3 const & RotationMatrix //!< Rotation matrix
638       )
639     { RF.setRotationMatrix( RotationMatrix ); }
640
641     //! Set 4x4 total transformation matrix
642     void
643     setTotalTransformationMatrix(
644       mat4 const & TM //!< 4x4 total transformation matrix
645     ) { RF.setTotalTransformationMatrix(TM); }
646
647     //! Get current Euler angles [rad] for X-axis \n
648     //! Warning: Factor as Rz*Rx*Ry!
```

```
649     real_type getEulerAngleX(void) const { return RF.getEulerAngleX(); };
650
651     //! Get current Euler angles [rad] for Y-axis \n
652     //! Warning: Factor as Rz*Rx*Ry!
653     real_type getEulerAngleY(void) const { return RF.getEulerAngleY(); };
654
655     //! Get current Euler angles [rad] for Z-axis \n
656     //! Warning: Factor as Rz*Rx*Ry!
657     real_type getEulerAngleZ(void) const { return RF.getEulerAngleZ(); };
658
659     //! Get contact depth at center point [m] \n
660     //! Warning: (if negative the tire does not touch the ground)!
661     virtual void getRho(
662       real_type & Rho //!< Depth at center point
663     ) const = 0;
664
665     //! Get contact depth matrix [m] \n
666     //! Warning: (if negative the tire does not touch the ground)!
667     virtual void getRho(
668       matN & Rho //!< Depth matrix
669     ) const = 0;
670
671     //! Get contact depth time derivative [m/s]
672     virtual
673     void
674     getRhoDot(
675       real_type const & Rho,    //!< Previous time step Rho [m]
676       real_type const & Time,   //!< Time step [s]
677       real_type       & RhoDot  //!< Penetration derivative [m/s]
678     ) const = 0;
679
680     //! Get contact depth time derivative matrix [m/s]
681     virtual
682     void
683     getRhoDot(
684       matN      const & Rho,   //!< Previous time step Rho [m]
685       real_type const & Time,  //!< Time step [s]
686       matN            & RhoDot //!< Penetration derivative [m/s]
687     ) const = 0;
688
689     //! Get contact normal versor
690     virtual
691     void
692     getContactNormal(
693       vec3 & Normal //!< Contact point normal direction
694     ) const = 0;
695
696     //! Get contact normal versors matrix
697     virtual
698     void
699     getContactNormal(
700       matN_vec3 & Normal //!< Contact point normal direction matrix
701     ) const = 0;
702
703     //! Get contact point
704     virtual
705     void
706     getContactPoint(
707       vec3 & Point //!< Contact point
708     ) const = 0;
709
710      //! Get contact point matrix
711     virtual
712     void
713     getContactPoint(
```

```
714       matN_vec3 & Point //!< Contact point matrix
715     ) const = 0;
716
717     //! Get contact point friction
718     virtual
719     void
720     getContactFriction(
721       real_type & Friction //!< Contact point friction
722     ) const = 0;
723
724     //! Get contact point friction matrix
725     virtual
726     void
727     getContactFriction(
728       matN & Friction //! Contact point friction matrix
729     ) const = 0;
730
731     //! Get contact point reference frame with 4x4 transformation matrix
732     virtual
733     void
734     getContactPointRF(
735       mat4 & PointRF //!< Contact point reference frame
736     ) const = 0;
737
738     //! Get contact point reference frame matrix with 4x4 transformation matrix
739     virtual
740     void
741     getContactPointRF(
742       matN_mat4 & PointRF //!< Contact point reference frame matrix
743     ) const = 0;
744
745     //! Get approximed contact area [m^2] with triangles intersection
746     virtual
747     void
748     getContactAreaTri(
749       RDF::TriangleRoad_list const & intersectionTriPtr, //!< Shadow/MeshSurface intersected
              triangles
750       real_type                    & Area                //!< Approximed contact area
751     ) const = 0;
752
753     //! Get approximed contact area [m^2] with local plane intersection
754     virtual void getContactArea(
755       real_type & Area //!< Approximed contact area
756     ) const = 0;
757
758     //! Get approximed contact volume [m^3] with local plane intersection
759     virtual void getContactVolume(
760       real_type & Volume //!< Approximed contact volume
761     ) const = 0;
762
763     //! Update current tire position and find contact parameters
764     virtual
765     void
766     setup(
767       RDF::MeshSurface       & Mesh,           //!< MeshSurface object (road)
768       mat4             const & TM,             //!< 4x4 total transformation matrix
769       real_type        const   DefaultH = 0.0, //!< Default height for out-of-mesh condition
770       bool                     print = false   //!< Flag for printing results
771     ) = 0;
772
773     //! Get relative camber angle [rad]
774     void
775     getRelativeCamber(
776       real_type & RelativeCamber //!< Relative camber angle
777     ) const;
```

```
778
779   //! Dimension of the contact points data structure
780   vec2_int
781   gridSize (
782     void
783   ) const {
784     return vec2_int(Precision.getXdiv(),Precision.getYdiv());
785   };
786
787   protected:
788   //! Perform one point sampling (ray-triangle intersection)
789   int_type
790   pointSampling(
791     RDF::TriangleRoad_list const &
792         intersectionTriPtr,           //!< Shadow/MeshSurface intersected triangles
793     vec3 const & RayOrigin,           //!< Ray origin
794     vec3 const & RayDirection,        //!< Ray direction
795     vec3       & SampledPt,           //!< Intersection point
796     real_type  & TriFriction,         //!< Intersected triangle friction
797     vec3       & TriNormal = vec3_NaN  //!< Intersected triangle normal
798   ) const;
799
800 };
801
802 /*\
803  |   __  __            _      _____                          _
804  |  |  \/  | __ _  __ _(_) ___|  ___|__  _ __ _ __ ___  _   _| | __ _
805  |  | |\/| |/ _` |/ _` | |/ __| |_ / _ \| '__| '_ ` _ \| | | | |/ _` |
806  |  | |  | | (_| | (_| | | (__|  _| (_) | |  | | | | | | |_| | | (_| |
807  |  |_|  |_|\__,_|\__, |_|\___|_|  \___/|_|  |_| |_| |_|\__,_|_|\__,_|
808  |               |___/
809 \*/
810
811 //! Pacejka MagicFormula contact model
812 class MagicFormula : public Tire {
813 protected:
814   Disk      SingleDisk;      //!< Single Disk
815   vec3      ContactNormal;   //!< Contact normal versor
816   vec3      ContactPoint;    //!< Contact point
817   real_type ContactFriction; //!< Contact friction
818
819   MagicFormula( MagicFormula const & ) = delete;                    //!< Deleted copy
          constructor
820   MagicFormula const & operator = ( MagicFormula const & ) = delete; //!< Deleted copy operator
821
822 public:
823   //! Default destructor
824   ~MagicFormula() {};
825
826   //! Variable set constructor
827   MagicFormula(
828     ETRTO const & Geometry //!< Tire ETRTO denomination
829   ) : Tire( Geometry, 1, 1) {
830     SingleDisk.set( Disk( vec2(0.0, 0.0), 0.0, TireGeometry.getTireRadius() ) );
831   }
832
833   //! Variable set constructor
834   MagicFormula(
835     real_type const SectionWidth, //!< Tire section width [mm]
836     real_type const AspectRatio,  //!< Tire aspect ratio [%]
837     real_type const RimDiameter   //!< Rim diameter [in]
838   )
839   : Tire( SectionWidth, AspectRatio, RimDiameter, 1, 1 )
840   {
841      SingleDisk.set( Disk( vec2(0.0, 0.0), 0.0, TireGeometry.getTireRadius() ) );
```

```
842     }
843
844     //! Get contact normal versor
845     void
846     getContactNormal(
847       vec3 & Normal //!< Contact point normal versor
848     ) const override {
849       Normal = ContactNormal;
850     }
851
852     //! Get contact normal versor matrix
853     void
854     getContactNormal(
855       matN_vec3 & Normal //!< Contact point normal versor matrix
856     ) const override {
857       Normal.resize(1,1);
858       Normal(0,0) = ContactNormal;
859     }
860
861     //! Get contact point
862     void
863     getContactPoint(
864       vec3 & Point //!< Contact point
865     ) const override {
866       Point = ContactPoint;
867     }
868
869     //! Get contact point matrix
870     void
871     getContactPoint(
872       matN_vec3 & Point //!< Contact point matrix
873     ) const override {
874       Point.resize(1,1);
875       Point(0,0) = ContactPoint;
876     }
877
878     //! Get contact friction
879     void
880     getContactFriction(
881       real_type & Friction //!< Contact friction
882     ) const override {
883       Friction = ContactFriction;
884     }
885
886     //! Get contact friction matrix
887     void
888     getContactFriction(
889       matN & Friction //!< Contact friction matrix
890     ) const override {
891       Friction.resize(1,1);
892       Friction(0,0) = ContactFriction;
893     }
894
895     //! Get contact point reference frame with 4x4 total transformation matrix
896     void
897     getContactPointRF(
898       mat4 & PointRF //!< 4x4 total transformation matrix
899     ) const override;
900
901     //! Get contact point reference frame matrix with 4x4 total transformation matrix
902     void
903     getContactPointRF(
904       matN_mat4 & PointRF //!< 4x4 total transformation matrix matrix
905     ) const override{
906       mat4 _PointRF;
```

```
907       getContactPointRF(_PointRF);
908       PointRF.resize(1,1);
909       PointRF(0,0) = _PointRF;
910     };
911
912     //! Get contact depth at center point [m] \n
913     //! Warning: (if negative the tire does not touch the ground)!
914     void
915     getRho(
916       real_type & Rho //!< Depth at center point
917     ) const override {
918       if ( TirePose == 2 ) {
919         Rho = 0.0;
920       } else {
921         Rho = TireGeometry.getTireRadius()-(RF.getOrigin()-ContactPoint).norm();
922       }
923     }
924
925     //! Get contact depth matrix [m] \n
926     //! Warning: (if negative the tire does not touch the ground)!
927     void
928     getRho(
929       matN & Rho //!< Depth matrix
930     ) const override {
931       real_type _Rho = 0.0;
932       Rho.resize(1,1);
933       if ( TirePose == 2 ) {
934         Rho(0,0) = 0.0;
935       } else {
936         getRho(_Rho);
937         Rho(0,0) = _Rho;
938       }
939     };
940
941     //! Get contact depth time derivative [m/s]
942     void
943     getRhoDot(
944       real_type const & Rho,    //!< Previous time step Rho [m]
945       real_type const & Time,   //!< Time step [s]
946       real_type       & RhoDot  //!< Penetration derivative [m/s]
947     ) const override {
948       if ( TirePose == 2 ) {
949         RhoDot = 0.0;
950       } else {
951         real_type _Rho = 0.0;
952         getRho(_Rho);
953         RhoDot = (_Rho - Rho) / Time;
954       }
955     }
956
957     //! Get contact depth time derivative matrix [m/s]
958     void
959     getRhoDot(
960       matN      const & Rho,   //!< Previous time step Rho [m]
961       real_type const & Time,  //!< Time step [s]
962       matN            & RhoDot //!< Penetration derivative [m/s]
963     ) const override{
964       RhoDot.resize(1,1);
965       if ( TirePose == 2 ) {
966         RhoDot(0,0) = 0.0;
967       } else {
968         real_type _RhoDot = 0.0;
969         getRhoDot(Rho(0,0), Time, _RhoDot);
970         RhoDot(0,0) = _RhoDot;
971       }
```

```
972     };
973
974     //! Get approximed contact area [m^2] with triangles intersection
975     void
976     getContactAreaTri(
977       RDF::TriangleRoad_list const & intersectionTriPtr, //!< Shadow/MeshSurface intersected
              triangles
978       real_type                    & Area                //!< Approximed contact area
979     ) const override {
980       if ( TirePose == 0 ) {
981         Area = quietNaN;
982       } else if ( TirePose == 2 ) {
983         Area = 0.0;
984       } else {
985         Area = SingleDisk.getPatchLength( intersectionTriPtr, RF ) * TireGeometry.getSectionWidth()
              ;
986       }
987     }
988
989     //! Get approximed contact area [m^2] with local plane intersection
990     void
991     getContactArea(
992       real_type & Area //!< Approximed contact area
993     ) const override {
994       if ( TirePose == 2 ) {
995         Area = 0.0;
996       } else {
997         Area = SingleDisk.getPatchLength( ContactNormal, ContactPoint, RF ) * TireGeometry.
              getSectionWidth();
998       }
999     }
1000
1001    //! Get approximed contact volume [m^3] with local plane intersection
1002    void
1003    getContactVolume(
1004      real_type & Volume //!< Approximed contact volume
1005    ) const override;
1006
1007    //! Update with precomputed tire position and contact parameters
1008    void
1009    setupPrecalculatedRoad(
1010      vec3      const & _ContactNormal,   //!< Contact normal versor
1011      vec3      const & _ContactPoint,    //!< Contact point
1012      real_type const   _ContactFriction, //!< Contact friction
1013      mat4      const & _TM               //!< 4x4 total transformation matrix
1014    ) {
1015      // Set the new reference frame
1016      RF.setTotalTransformationMatrix(_TM);
1017      // Update class members
1018      ContactNormal = _ContactNormal;
1019      ContactPoint  = _ContactPoint;
1020    }
1021
1022    //! Update current tire position and find contact parameters
1023    void
1024    setup(
1025      RDF::MeshSurface       & Mesh,           //!< MeshSurface object (road)
1026      mat4             const & TM,             //!< 4x4 total transformation matrix
1027      real_type        const   DefaultH = 0.0, //!< Default height for out-of-mesh condition
1028      bool                     print = false   //!< Flag for printing results
1029    ) override;
1030
1031  protected:
1032    //! Perform triangles sampling on 4 points at ±0.1*R0 along X and ±0.3*W along Y
1033    void
```

```
1034    fourPointsSampling(
1035      RDF::TriangleRoad_list const & intersectionTriPtr, //!< Shadow/MeshSurface intersected
              triangles
1036      real_type              const   DefaultH           //!< Default height for out-of-mesh
              condition
1037    );
1038
1039    //! Calculate the real contact point (on Disk plane)
1040    void
1041    calculateContactPoint(
1042    row_vec3  const & Qvec,     //!< Intersection points
1043    real_type const   DefaultH  //!< Default height for out-of-mesh condition
1044    );
1045  };
1046
1047  /*\
1048   |   __ __       _ _   _ ____  _     _
1049   |  |  \/  |_   _| | |_(_)  _ \(_)___| | __
1050   |  | |\/| | | | | __| | | | | / __| |/ /
1051   |  | |  | | |_| | |_| | |_| | \__ \   <
1052   |  |_|  |_|\__,_|_|\__|_|____/|_|___/_|\_\
1053   |
1054  \*/
1055
1056  //! Multi-disk tire contact model
1057  class MultiDisk : public Tire {
1058  private:
1059    std::vector<Disk> DiskVector;            //!< Disks vector
1060    matN_vec3         ContactNormal;         //!< Contact normal versors matrix
1061    matN_vec3         ContactPoint;          //!< Contact point matrix
1062    matN              ContactFriction;       //!< Contact friction matrix
1063    vec3              SingleContactNormal;   //!< Single normal versor
1064    vec3              SingleContactPoint;    //!< Single contact point
1065    real_type         SingleContactFriction; //!< Single contact friction
1066
1067    MultiDisk( MultiDisk const & ) = delete;                    //!< Deleted copy constructor
1068    MultiDisk const & operator = ( MultiDisk const & ) = delete; //!< Deleted copy operator
1069
1070  public:
1071    //! Default destructor
1072    ~MultiDisk() {};
1073
1074    //! Variable set constructor
1075    MultiDisk(
1076      ETRTO    const & Geometry, //!< Tire ETRTO denomination object
1077      int_type const   Xdiv,     //!< Number of points in X-axis (sampling points for each disk)
1078      int_type const   Ydiv      //!< Number of points in Y-axis (number of disks)
1079    ) : Tire( Geometry, Xdiv, Ydiv ) {
1080      // Locate the disks
1081      Eigen::VectorXd Dvec(Ydiv);
1082      offsetDisks( Dvec );
1083      for ( int_type i = 0; i < Dvec.size(); ++i ) {
1084        DiskVector.push_back( Disk( vec2(0, 0), Dvec[i], TireGeometry.getTireRadius() ) );
1085      }
1086      // Resize the contact point, friction and normal matrices
1087      ContactNormal.resize( Ydiv, Xdiv );
1088      ContactPoint.resize( Ydiv, Xdiv );
1089      ContactFriction.resize( Ydiv, Xdiv );
1090    }
1091
1092    //! Variable set constructor
1093    MultiDisk(
1094      real_type const SectionWidth, //!< Tire section width [mm]
1095      real_type const AspectRatio,  //!< Tire aspect ratio [%]
1096      real_type const RimDiameter,  //!< Rim diameter [in]
```

```
1097      int_type  const Xdiv,         //!< Number of points in X-axis (sampling points for each disk)
1098      int_type  const Ydiv          //!< Number of points in Y-axis (number of disks)
1099    ) : Tire( SectionWidth, AspectRatio, RimDiameter, Xdiv, Ydiv )
1100    {
1101      // Locate the disks
1102      Eigen::VectorXd Dvec( Ydiv );
1103      offsetDisks( Dvec );
1104      for ( int_type i = 0; i < Ydiv; ++i )
1105        DiskVector.push_back( Disk( vec2(0, 0), Dvec[i], TireGeometry.getTireRadius() ) );
1106      // Resize the contact point, friction and normal matrices
1107      ContactNormal.resize( Ydiv, Xdiv );
1108      ContactPoint.resize( Ydiv, Xdiv );
1109      ContactFriction.resize( Ydiv, Xdiv );
1110    }
1111
1112    //! Get grid step on X-axis [m]
1113    real_type
1114    getXstep(void) const
1115    { return TireGeometry.getTireDiameter() / (Precision.getXdiv() - 1); }
1116
1117    /*/! Get grid step on Y-axis [m]
1118    real_type
1119    getYstep(void) const
1120    { return TireGeometry.getSectionWidth() / (Precision.getYdiv() - 1); }*/
1121
1122    //! Get step on Y-axis between disks [m]
1123    real_type
1124    getDiskStep(void) const
1125    { return TireGeometry.getSectionWidth() / Precision.getYdiv(); }
1126
1127    //! Get contact normal mean versor
1128    void
1129    getContactNormal(
1130      vec3 & Normal //!< Contact normal mean versor
1131    ) const override {
1132      Normal = SingleContactNormal;
1133    }
1134
1135    //! Get contact normal versors matrix
1136    void
1137    getContactNormal(
1138      matN_vec3 & Normal //!< Contact normal versors matrix
1139    ) const override {
1140      Normal = ContactNormal;
1141    }
1142
1143    //! Get Disk contact normal versors vector
1144    void
1145    getContactNormalDisk(
1146      int_type const   i,     //!< i-th Disk
1147      col_vec3       & Normal //!< Contact normal versors vector
1148    ) const {
1149      Normal = ContactNormal.col(i);
1150    }
1151
1152    //! Get Disk contact normal mean versor
1153    void
1154    getContactNormalDisk(
1155      int_type const   i,     //!< i-th Disk
1156      vec3           & Normal //!< Contact normal mean versor
1157    ) const {
1158      Normal = mean(ContactNormal.col(i));
1159    }
1160
1161    //! Get single contact point
```

```
1162    void
1163    getContactPoint(
1164      vec3 & Point //!< Single contact point
1165    ) const override {
1166      Point = SingleContactNormal;
1167    }
1168
1169    //! Get contact point matrix
1170    void
1171    getContactPoint(
1172      matN_vec3 & Point //!< Contact point matrix
1173    ) const override {
1174      Point = ContactNormal;
1175    }
1176
1177    //! Get mean contact friction
1178    void
1179    getContactFriction(
1180      real_type & Friction //!< Mean contact friction
1181    ) const override {
1182      Friction = SingleContactFriction;
1183    }
1184
1185    //! Get contact friction matrix
1186    void
1187    getContactFriction(
1188      matN & Friction //!< Contact friction matrix
1189    ) const override {
1190      Friction = ContactFriction;
1191    }
1192
1193    //! Get Disk contact friction vector
1194    void
1195    getContactFrictionDisk(
1196      int_type const   i,       //!< i-th Disk
1197      col_vecN       & Friction //!< Contact friction vector
1198    ) const {
1199      Friction = ContactFriction.col(i);
1200    }
1201
1202    //! Get Disk mean contact friction
1203    void
1204    getContactFrictionDisk(
1205      int_type  const   i,       //!< i-th Disk
1206      real_type       & Friction //!< Mean contact friction
1207    ) const {
1208      Friction = ContactFriction.col(i).mean();
1209    }
1210
1211    //! Get contact point reference frame with 4x4 total transformation matrix
1212    void
1213    getContactPointRF(
1214      mat4 & PointRF //!< 4x4 total transformation matrix
1215    ) const override;
1216
1217    //! Get contact point reference frame matrix with 4x4 total transformation matrix
1218    void
1219    getContactPointRF(
1220      matN_mat4 & PointRF //!< 4x4 total transformation matrix matrix
1221    ) const override;
1222
1223    //! Get contact depth at center point [m] \n
1224    //! Warning: (if negative the tire does not touch the ground)!
1225    void
1226    getRho(
```

```
1227      real_type & Rho //!< Depth at center point
1228    ) const override;
1229
1230    //! Get contact depth matrix [m] \n
1231    //! Warning: (if negative the tire does not touch the ground)!
1232    void
1233    getRho(
1234      matN & Rho //!< Depth matrix
1235    ) const override;
1236
1237    //! Get contact depth time derivative [m/s]
1238    void
1239    getRhoDot(
1240      real_type const & Rho,    //!< Previous time step Rho [m]
1241      real_type const & Time,   //!< Time step [s]
1242      real_type       & RhoDot  //!< Penetration derivative [m/s]
1243    ) const override;
1244
1245    //! Get contact depth time derivative matrix [m/s]
1246    void
1247    getRhoDot(
1248      matN      const & Rho,   //!< Previous time step Rho [m]
1249      real_type const & Time,  //!< Time step [s]
1250      matN            & RhoDot //!< Penetration derivative [m/s]
1251    ) const override;
1252
1253    //! Get approximed contact area [m^2] with triangles intersection
1254    void
1255    getContactAreaTri(
1256      RDF::TriangleRoad_list const & intersectionTriPtr, //!< Shadow/MeshSurface intersected
          triangles
1257      real_type                    & Area                //!< Approximed contact area
1258    ) const override;
1259
1260    //! Get approximed Disk contact area [m^2] with triangles intersection.
1261    void
1262    getContactAreaTriDisk(
1263      int_type               const   i,                  //!< i-th Disk
1264      RDF::TriangleRoad_list const & intersectionTriPtr, //!< Shadow/MeshSurface intersected
          triangles
1265      real_type                    & Area                //!< Approximed contact area
1266    ) const;
1267
1268    //! Get approximed contact area [m^2] with local plane intersection
1269    void
1270    getContactArea(
1271      real_type & Area //!< Approximed contact area
1272    ) const override;
1273
1274    //! Get Disk approximed contact area [m^2] with local plane intersection
1275    void
1276    getContactAreaDisk(
1277      int_type  const   i,   //!< i-th Disk
1278      real_type       & Area //!< Approximed contact area
1279    ) const;
1280
1281    //! Get approximed contact volume [m^3] with local plane intersection
1282    void
1283    getContactVolume(
1284      real_type & Volume //!< Approximed contact volume
1285    ) const override;
1286
1287    //! Get approximed Disk contact volume [m^3] with local plane intersection
1288    void
1289    getContactVolumeDisk(
```

```
1290      int_type  const   i,     //!< i-th Disk
1291      real_type       & Volume //!< Approximed contact volume
1292    ) const;
1293
1294    //! Update current tire position and find contact parameters
1295    void
1296    setup(
1297      RDF::MeshSurface       & Mesh,           //!< MeshSurface object (road)
1298      mat4             const & TM,             //!< 4x4 total transformation matrix
1299      real_type        const   DefaultH = 0.0, //!< Default height for out-of-mesh condition
1300      bool                     print = false   //!< Flag for printing results
1301    ) override;
1302
1303  private:
1304    //! Find offsets on Y-axis values for disks
1305    void
1306    offsetDisks(
1307       Eigen::VectorXd & OffsetVec //!< Disks offsets on Y-axis
1308    ) const;
1309
1310    //! Perform grid sampling
1311    void
1312    gridSampling(
1313      RDF::TriangleRoad_list const & intersectionTriPtr, //!< Shadow/MeshSurface intersected
              triangles
1314      real_type              const   DefaultH            //!< Default height for out-of-mesh
              condition
1315    );
1316
1317    //! Update current tire single contact point parameters
1318    void
1319    calcSingleContactPoint(
1320      real_type const DefaultH //!< Default height for out-of-mesh condition
1321    );
1322
1323    //! Perform contact point matrix 3D shift (in absolute reference frame)
1324    void
1325    shiftContactPoint(
1326      vec3 const & Shift //!< 3D shift vector
1327    );
1328
1329    //! Set contact normal versor
1330    void
1331    setContactNormal(
1332      vec3 const & Value //!< Contact normal versor
1333    );
1334
1335    //! Calculate mean vector for 3D vector matrix
1336    vec3
1337    mean(
1338      matN_vec3 const & Mat //!< 3D vector matrix
1339    ) const;
1340
1341  };
1342
1343 }  // namespace PatchTire
1344
1345 ///
1346 /// eof: PatchTire.hh
1347 ///
```

B.5 PatchTire.cc

```
#include "PatchTire.hh"

namespace PatchTire {

  using namespace TireGround;

  /*\
   |   ____  _     _
   |  |  _ \(_)___| | __
   |  | | | | / __| |/ /
   |  | |_| | \__ \   <
   |  |____/|_|___/_|\_\
  \*/

  bool
  Disk::isPointInside(
    vec2 const & Point
  ) const {
    // If output bool is true the point is inside the circumfererence,
    // otherwise it is outside.
    vec2 PO( Point - OriginXZ );
    return PO.dot(PO) <= Radius*Radius;
  }

  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

  int_type
  Disk::intersectSegment(
    vec2      const & Point_1,
    vec2      const & Point_2,
    vec2            & Intersect_1,
    vec2            & Intersect_2
  ) const {
    real_type t_param;
    vec2      d( Point_2 - Point_1 );
    vec2      P1O( Point_1 - OriginXZ );
    real_type A   = d.dot(d);
    real_type B   = 2 * d.dot(P1O);
    real_type C   = P1O.dot(P1O) - Radius*Radius;
    real_type discriminant = B*B - 4 * A * C;
    if ( A <= epsilon || discriminant < 0 ) {
      // No real solutions
      Intersect_1 = vec2(quietNaN, quietNaN);
      Intersect_2 = vec2(quietNaN, quietNaN);
      return 0;
    } else if ( std::abs(discriminant) < epsilon ) {
      // One solution
      t_param     = -B / (2*A);
      Intersect_1 = Point_1 + t_param * d;
      Intersect_2 = vec2(quietNaN, quietNaN);
      return 1;
    } else {
      // Two solutions
      t_param = (-B + std::sqrt(discriminant)) / (2 * A);
      Intersect_1 = Point_1 + t_param * d;
      t_param = (-B - std::sqrt(discriminant)) / (2 * A);
      Intersect_2 = Point_1 + t_param * d;
      return 2;
    }
  }

  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

  bool
  Disk::intersectPlane(
```

```
66    vec3 const & Plane_Normal,
67    vec3 const & Plane_Point,
68    vec3       & Line_Direction,
69    vec3       & Line_Point
70  ) const {
71    // Plane(Point,Normal) and Disk intersection -> Parametric rect
72    vec3 Disk_Point( getOriginXYZ() );
73    vec3 Disk_Normal( 0.0, 1.0, 0.0 );
74    // Rect direction
75    Line_Direction = Plane_Normal.cross(Disk_Normal);
76    // If the two plane are parallel they do not intersects
77    if ( Line_Direction.norm() > epsilon) {
78      // Given the plane ax+by+cz=d
79      real_type d_Disk = - Disk_Point.dot(Disk_Normal);
80      real_type d_Plane = - Plane_Point.dot(Plane_Normal);
81      // Find a point on the line, which is also on both planes
82      // choose simplest plane where d=0: ax + by + cz = 0
83      vec3 u1( d_Disk * Plane_Normal );
84      vec3 u2( -d_Plane * Disk_Normal );
85      Line_Point = (u1 + u2).cross(Line_Direction) / Line_Direction.dot(Line_Direction);
86      return true;
87    } else {
88      return false;
89    }
90  }
91
92  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
93
94  real_type
95  Disk::getPatchLength(
96    RDF::TriangleRoad_list const & intersectionTriPtr,
97    ReferenceFrame         const & RF
98  ) const {
99    // Disk point and vector in absolute reference frame
100   vec3 Disk_Point( RF.getOrigin() + RF.getRotationMatrix()*getOriginXYZ() );
101   vec3 Disk_Normal( RF.getY() );
102   real_type PatchLength = 0.0;
103   std::vector<vec3> IntersectionPts;
104   for ( unsigned i = 0; i < intersectionTriPtr.size(); ++i ) {
105     if( (*intersectionTriPtr[i]).intersectPlane(Disk_Normal, Disk_Point, IntersectionPts) ) {
106       // Transform in disk relative reference frame
107       vec3 P1_rel( RF.getRotationMatrixInverse(IntersectionPts[0] - RF.getOrigin()) );
108       vec3 P2_rel( RF.getRotationMatrixInverse(IntersectionPts[1] - RF.getOrigin()) );
109       // Transfer only the XZ part (Y part must be = to OffsetY, so useless)
110       PatchLength += getPatchLength( vec2(P1_rel.x(),P1_rel.z()), vec2(P2_rel.x(),P2_rel.z()), RF
              );
111     }
112   }
113   return PatchLength;
114 }
115
116 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
117
118 real_type
119 Disk::getPatchLength(
120     vec3           const & Plane_Normal,
121     vec3           const & Plane_Point,
122     ReferenceFrame const & RF
123 ) const {
124   // Change reference frame for local road plane
125   vec3 Plane_Normal_rel( RF.getRotationMatrixInverse(Plane_Normal).normalized() ) ;
126   vec3 Plane_Point_rel( RF.getRotationMatrixInverse(Plane_Point - RF.getOrigin()) );
127   // Check if two plane intersects and find the intersecting rect.
128   vec3 P, T;
129   if(intersectPlane( Plane_Normal_rel, Plane_Point_rel, T, P)){
```

```
130       // Make a segment on the intersection (on relative Disk rerence frame)
131       vec3 P1( P - 200.0*Radius*T );
132       vec3 P2( P + 200.0*Radius*T );
133       return getPatchLength(vec2(P1.x(),P1.z()), vec2(P2.x(),P2.z()), RF);
134     } else {
135       RDF_ERROR("Cannot handle planes intersection");
136       return quietNaN;
137     }
138   }
139
140   // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
141
142   real_type
143   Disk::getPatchLength(
144     vec2           const & PointXZ_1,
145     vec2           const & PointXZ_2,
146     ReferenceFrame const & RF
147   ) const {
148     vec2 Intersection_1, Intersection_2;
149     int_type Type = this->intersectSegment(
150       PointXZ_1, PointXZ_2, Intersection_1, Intersection_2
151     );
152     if ( Type == 0 ) {
153       // No contact points, the line segment is not into the Disk
154       return 0.0;
155     } else if (Type == 1) {
156       // Tangent, no length added
157       return 0.0;
158     } else if (Type == 2) {
159       // Check whether the two segment points are into the circle
160       bool Pose_pt1 = this->isPointInside( PointXZ_1 );
161       bool Pose_pt2 = this->isPointInside( PointXZ_2 );
162       // Check whether the two intersection points are onto the line segment
163       bool Pose_int1 = algorithms::intersectPointSegment(PointXZ_1, PointXZ_2, Intersection_1);
164       bool Pose_int2 = algorithms::intersectPointSegment(PointXZ_1, PointXZ_2, Intersection_2);
165       // Cases
166       // Line segment PointXZ_1 and line segment PointXZ_2 outside the circle,
167       // intersection points oustide line segment
168       if ( !Pose_pt1 && !Pose_pt2 && !Pose_int1 && !Pose_int2 ) {
169         return 0.0;
170       }
171       // Line segment PointXZ_1 and line segment PointXZ_2 into the circle,
172       // intersection points oustide line segment
173       if ( Pose_pt1 && Pose_pt2 && !Pose_int1 && !Pose_int2 ) {
174         return (PointXZ_2 - PointXZ_1).norm();
175       }
176       // Intersection points into the line segment and line segment points
177       // outside the circle
178       else if ( !Pose_pt1 && !Pose_pt2 && Pose_int1 && Pose_int2 ) {
179         return (Intersection_2 - Intersection_1).norm();
180       }
181       // Line segment Point_1 outside the circle, line segment Point_2
182       // inside the circle
183       else if ( !Pose_pt1 && Pose_pt2 ) {
184         if ( Pose_int1 && !Pose_int2 ) {
185           // Add length from Intersection_1 to Point_2
186           return (Intersection_1 - PointXZ_2).norm();
187         } else if ( Pose_int2 && !Pose_int1 ) {
188           // Add length from Intersection_2 to Point_2
189           return (Intersection_2 - PointXZ_2).norm();
190         }
191       }
192       // Line segment Point_1 inside the circle, line segment Point_2
193       // outside the circle
194       else if ( Pose_pt1 && !Pose_pt2 ) {
```

```
195       if ( Pose_int1 && !Pose_int2 ) {
196         // Add length from Intersection_1 to Point_1
197         return (Intersection_1 - PointXZ_1).norm();
198       } else if ( !Pose_int1 && Pose_int2 ) {
199         // Add length from Intersection_2 to Point_1
200         return (Intersection_2 - PointXZ_1).norm();
201       }
202     }
203   }
204   RDF_ERROR("Invalid intersection on Disk::getPatchLength");
205   return quietNaN;
206 }
207
208 /*\
209  |   ____       __                                 _____
210  |  |  _ \ ___ / _| ___ _ __ ___ _ __   ___ ___|  ___| __ __ _ _ __ ___   ___
211  |  | |_) / _ \ |_ / _ \ '__/ _ \ '_ \ / __/ _ \ |_ | '__/ _` | '_ ` _ \ / _ \
212  |  |  _ <  __/  _|  __/ | |  __/ | | | (_|  __/  _|| | | (_| | | | | | |  __/
213  |  |_| \_\___|_|  \___|_|  \___|_| |_|\___\___|_|  |_|  \__,_|_| |_| |_|\___|
214  |
215 \*/
216
217 real_type
218 ReferenceFrame::getEulerAngleX(
219   void
220 ) const {
221   real_type r21 = RotationMatrix(2, 1);
222   if (r21 < 1.0) {
223     if (r21 > -1.0) {
224       return std::asin(r21);
225     } else {  // r21 == -1.0
226       // Not a unique solution : thetaY - thetaZ = atan2( r02 , r00 )
227       return -G2lib::m_pi / 2.0;
228     }
229   } else {  // r21 == 1.0
230     // Not a unique solution : thetaY + thetaZ = atan2( r02 , r00 )
231     return G2lib::m_pi / 2.0;
232   }
233 }
234
235 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
236
237 real_type
238 ReferenceFrame::getEulerAngleY(
239   void
240 ) const {
241   real_type r20 = RotationMatrix(2, 0);
242   real_type r21 = RotationMatrix(2, 1);
243   real_type r22 = RotationMatrix(2, 2);
244   if (r21 < 1.0) {
245     if (r21 > -1.0) {
246       return std::atan2(-r20, r22);
247     } else {  // r21 == -1.0
248       // Not a unique solution : thetaY - thetaZ = atan2( r02 , r00 )
249       return 0.0;
250     }
251   } else {  // r21 == 1.0
252     // Not a unique solution : thetaY + thetaZ = atan2( r02 , r00 )
253     return 0.0;
254   }
255 }
256
257 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
258
259 real_type
```

```
260  ReferenceFrame::getEulerAngleZ(
261    void
262  ) const {
263    real_type r00 = RotationMatrix(0, 0);
264    real_type r01 = RotationMatrix(0, 1);
265    real_type r02 = RotationMatrix(0, 2);
266    real_type r11 = RotationMatrix(1, 1);
267    real_type r21 = RotationMatrix(2, 1);
268    if (r21 < 1.0) {
269      if (r21 > -1.0) {
270        return std::atan2(-r01, r11);
271      } else {  // r21 == -1.0
272        // Not a unique solution : thetaY - thetaZ = atan2( r02 , r00 )
273        return -std::atan2(r02, r00);
274      }
275    } else {  // r21 == 1.0
276      // Not a unique solution : thetaY + thetaZ = atan2( r02 , r00 )
277      return std::atan2(r02, r00);
278    }
279  }
280
281  /*\
282  |   ____  _                 _
283  |  / ___|| |__   __ _  __| | _____      __
284  |  \___ \| '_ \ / _` |/ _` |/ _ \ \ /\ / /
285  |   ___) | | | | (_| | (_| | (_) \ V  V /
286  |  |____/|_| |_|\__,_|\__,_|\___/ \_/\_/
287  \*/
288
289  void
290  Shadow::update(
291      ETRTO const & TireGeometry,
292      ReferenceFrame const & RF
293  ) {
294    // Calculate maximum covered space
295    real_type diagonal =
296        hypot(TireGeometry.getSectionWidth(), TireGeometry.getTireDiameter()) /
297        2;
298
299    // Increment shadow to take in account camber angle
300    real_type inc = 1.1;
301
302    // Set new tire shadow domain
303    this->Xmax = RF.getOrigin()[0] + inc * diagonal;
304    this->Ymax = RF.getOrigin()[1] + inc * diagonal;
305    this->Xmin = RF.getOrigin()[0] - inc * diagonal;
306    this->Ymin = RF.getOrigin()[1] - inc * diagonal;
307  }
308
309  /*\
310   |         _                   _ _   _
311   |    __ _| | __ _  ___  _ __(_) |_| |__  _ __ ___  ___
312   |   / _` | |/ _` |/ _ \| '__| | __| '_ \| '_ ` _ \/ __|
313   |  | (_| | | (_| | (_) | |  | | |_| | | | | | | | \__ \
314   |   \__,_|_|\__, |\___/|_|  |_|\__|_| |_|_| |_| |_|___/
315   |           |___/
316  \*/
317
318  namespace algorithms {
319
320    bool
321    intersectPointSegment(
322      vec2 const & Point_1,
323      vec2 const & Point_2,
324      vec2 const & Point
```

```
325   ) {
326     // A and B are the extremities of the current segment C
327     // is the point to check.
328     // Create the vector AB
329     vec2 AB( Point_2 - Point_1 );
330     // Create the vector AC
331     vec2 AC( Point - Point_1 );
332     // Compute the cross product of AB and AC
333     // Check if the three points are aligned (cross product is null)
334     real_type z = AB.x() * AC.y() - AC.x() * AB.y();
335     if (z*z > epsilon) {
336       RDF_ERROR("Cannot handle point-segment intersection!");
337       return false;
338     }
339     // Compute the dot product of vectors
340     real_type KAC = AB.dot(AC);
341     if ( KAC      < -epsilon ) return false;
342     if ( abs(KAC) < epsilon  ) return true;
343     // Compute the square of the segment lenght
344     real_type KAB = AB.dot(AB);
345     if ( KAC > KAB ) return false;
346     if ( abs(KAC - KAB) < epsilon ) return true;
347     // The point is on the segment
348     return true;
349   }
350
351   // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
352
353   bool
354   intersectRayPlane(vec3 const & planeN,
355                     vec3 const & planeP,
356                     vec3 const & RayPoint,
357                     vec3 const & RayDirection,
358                     vec3       & IntersectionPt
359   ) {
360     // Check if the segment lays on the plane
361     RDF_ASSERT( std::abs(planeN.dot(RayDirection)) > epsilon &&
362                 std::abs((planeP - RayPoint).dot(planeN)) > epsilon,
363                 "Cannot find proper contact point (ray lays on the plane)." );
364     real_type d = - planeP.dot(planeN);
365     real_type t = -(RayPoint.dot(planeN) + d) / (RayDirection.dot(planeN));
366     if (t >= 0) {
367       IntersectionPt = RayPoint + t * RayDirection;
368       return true;
369     } else {
370       IntersectionPt = vec3_NaN;
371       return false;
372     }
373   }
374
375
376 }  // namespace algorithms
377
378   /*\
379    |   _____ _          ____
380    |  |_   _(_)_ __ ___| __ )  __ _ ___  ___
381    |    | | | '__/ _ \  _ \ / _` / __|/ _ \
382    |    | | | | |  __/ |_) | (_| \__ \  __/
383    |    |_| |_|_|  \___|____/ \__,_|___/\___|
384    |
385   \*/
386
387   int_type
388   Tire::pointSampling(
389     RDF::TriangleRoad_list const & intersectionTriPtr,
```

```
390     vec3                  const & RayOrigin,
391     vec3                  const & RayDirection,
392     vec3                        & SampledPt,
393     real_type                   & TriFriction,
394     vec3                        & TriNormal
395   ) const {
396     vec3 IntersectionPoint;
397     std::vector<vec3> IntersectionPointVec, TriNormalVec;
398     std::vector<real_type> TriFrictionVec;
399     bool IntersectBool = false;
400     for (unsigned t = 0; t < intersectionTriPtr.size(); ++t) {
401       if ( (*intersectionTriPtr[t]).intersectRay(
402               RayOrigin, RayDirection, IntersectionPoint) ) {
403         // Store results
404         IntersectionPointVec.push_back(IntersectionPoint);
405         TriFrictionVec.push_back((*intersectionTriPtr[t]).getFriction());
406         TriNormalVec.push_back((*intersectionTriPtr[t]).getNormal());
407         IntersectBool = true;
408       }
409     }
410     // Select the highest intersection point
411     if (IntersectionPointVec.size() == 1 && IntersectBool){
412       SampledPt   = IntersectionPointVec[0];
413       TriFriction = TriFrictionVec[0];
414       TriNormal   = TriNormalVec[0];
415       return 1;
416     } else if (IntersectionPointVec.size() > 1 && IntersectBool) {
417       SampledPt   = IntersectionPointVec[0];
418       TriFriction = TriFrictionVec[0];
419       TriNormal   = TriNormalVec[0];
420       for (unsigned j = 1; j < IntersectionPointVec.size(); ++j) {
421         if (IntersectionPointVec[j][2] > SampledPt[2]) {
422           SampledPt   = IntersectionPointVec[j];
423           TriFriction = TriFrictionVec[j];
424           TriNormal   = TriNormalVec[j];
425         }
426       }
427       return 1;
428     } else if (intersectionTriPtr.size() > 0 && !IntersectBool) {
429       // Flying over the mesh
430       //std::cout << "Warning: We are flying over the mesh capitain.\n\n";
431       return 2;
432     } else if (intersectionTriPtr.size() == 0) {
433       // Out of mesh
434       //std::cout << "Warning: Out of mesh dude.\n\n";
435       return 0;
436     } else {
437       RDF_ERROR("Not handled consition");
438     }
439   }
440
441   // . . . . . .
442
443   void
444   Tire::getRelativeCamber(
445     real_type & RelativeCamber
446   ) const {
447     if ( TirePose == 2 ) {
448         RelativeCamber = 0.0;
449     } else {
450       mat4 PointRF;
451       getContactPointRF(PointRF);
452       mat3 RF_pc_wh( RF.getRotationMatrix().transpose()*PointRF.block(0,0,3,3) );
453       real_type r21 = RF_pc_wh(2, 1);
454       if (r21 < 1.0) {
```

```
455       if (r21 > -1.0) {
456         RelativeCamber =  std::asin(r21);
457       } else {  // r21 == -1.0
458         // Not a unique solution : thetaY - thetaZ = atan2( r02 , r00 )
459         RelativeCamber =  -G2lib::m_pi / 2.0;
460       }
461     } else {  // r21 == 1.0
462       // Not a unique solution : thetaY + thetaZ = atan2( r02 , r00 )
463       RelativeCamber =  G2lib::m_pi / 2.0;
464     }
465   }
466 }
467
468 /*\
469  |   __  __             _      _____                          _
470  |  |  \/  | __ _  __ _(_) ___|  ___|__  _ __ _ __ ___  _   _| | __ _
471  |  | |\/| |/ _` |/ _` | |/ __| |_ / _ \| '__| '_ ` _ \| | | | |/ _` |
472  |  | |  | | (_| | (_| | | (__|  _| (_) | |  | | | | | | |_| | | (_| |
473  |  |_|  |_|\__,_|\__, |_|\___|_|  \___/|_|  |_| |_| |_|\__,_|_|\__,_|
474  |                |___/
475 \*/
476
477 void
478 MagicFormula::getContactPointRF(
479   mat4 & PointRF
480 ) const {
481   vec3 X_versor = (RF.getY().cross(ContactNormal)).normalized();
482   vec3 Y_versor = (ContactNormal.cross(X_versor)).normalized();
483   PointRF << X_versor, Y_versor, ContactNormal, ContactPoint,
484       vec4(0.0, 0.0, 0.0, 1.0).transpose();
485 }
486
487 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
488
489 void
490 MagicFormula::getContactVolume(
491   real_type & Volume
492 ) const {
493   if ( TirePose == 2 ) {
494     Volume = 0.0;
495   } else {
496     real_type R = TireGeometry.getTireRadius();
497     real_type c = SingleDisk.getPatchLength( ContactNormal, ContactPoint, RF );
498     real_type c_2R = c / (2*R);
499     Volume = R*R * (std::asin(c_2R)-c_2R*std::sqrt(1-c_2R*c_2R)) * TireGeometry.getSectionWidth()
          ;
500   }
501 }
502
503 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
504
505 void
506 MagicFormula::setup(
507   RDF::MeshSurface       & Mesh,
508   mat4             const & TM,
509   real_type        const   DefaultH,
510   bool                     print
511 ) {
512   // Set the new reference frame
513   RF.setTotalTransformationMatrix(TM);
514   // Shadow bounding box object
515   Shadow TireShadow(TireGeometry, RF);
516   // Local intersected triangles vector
517   RDF::TriangleRoad_list intersectionTriPtr =
518       Mesh.intersectAABBtree(TireShadow.getAABBPtr());
```

```
519   // Update the local road plane (normal, point and realtive camber)
520   fourPointsSampling(intersectionTriPtr, DefaultH);
521
522   if (print) {
523   // Print the results
524   mat4 ContactPointRF;
525   getContactPointRF(ContactPointRF);
526   real_type Rho = 0.0;
527   getRho(Rho);
528   real_type RelativeCamber = 0.0;
529   getRelativeCamber(RelativeCamber);
530   real_type ContactArea = 0.0;
531   getContactArea(ContactArea);
532   real_type ContactAreaTri = 0.0;
533   getContactAreaTri(intersectionTriPtr, ContactAreaTri);
534   real_type ContactVolume = 0.0;
535   getContactVolume(ContactVolume);
536     std::cout << "MAGIC FORMULA CONTACT PARAMETERS\n"
537               << "\tYaw angle\n"
538               << "\ t = " << getEulerAngleZ() / G2lib::m_pi << "pi rad\n"
539               << "\tRotation angle\n"
540               << "\tß = " << getEulerAngleY() / G2lib::m_pi << "pi rad\n"
541               << "\tCamber angle\n"
542               << "\Γt = " << getEulerAngleX() / G2lib::m_pi << "pi rad\n"
543               << "\tNormal contact point vector of the local track plane "
544                 "(absolute reference frame)\n"
545               << "\tN = [ " << ContactNormal.x() << ", " << ContactNormal.y()
546               << ", " << ContactNormal.z() << " ]\n"
547               << "\tLocal contact point (absolute reference frame)\n"
548               << "\tP = [ " << ContactPoint.x() << ", " << ContactPoint.y()
549               << ", " << ContactPoint.z() << " ]\n"
550               << "\tLocal contact point reference frame\n"
551               << ContactPointRF << "\n"
552               << "\tRelative camber angle\n"
553               << "\tß = " << RelativeCamber / G2lib::m_pi << "pi rad\n"
554               << "\tLocal contact point friction\n"
555               << "\tf = " << ContactFriction << "\n"
556               << "\tLocal contact depth (on center point) Rho\n"
557               << "\tD = " << Rho << " m\n"
558               << "\tLocal approximated contact area (calculated with local plane)\n"
559               << "\tA = " << ContactArea << " m^2\n"
560               << "\tLocal approximated contact area (calculated with triangles intersection)\n"
561               << "\tA = " << ContactAreaTri << " m^2\n"
562               << "\tLocal approximated contact volume (calculated with local plane)\n"
563               << "\tV = " << ContactVolume << " m^3\n"
564               << "\tTire position flag = " << TirePose << "\n\n";
565   }
566 }
567
568 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
569
570 void
571 MagicFormula::fourPointsSampling(
572   RDF::TriangleRoad_list const & intersectionTriPtr,
573   real_type              const   DefaultH
574 ) {
575   // Calculate Delta_X and Delta_Y
576   real_type Delta_X = 0.1 * TireGeometry.getTireRadius();
577   real_type Delta_Y = 0.3 * TireGeometry.getSectionWidth();
578   // Store the four sample positions
579   row_vec3 Qpos(4);
580   Qpos[0] = RF.getOrigin() + RF.getRotationMatrix() * vec3(Delta_X, 0.0, 0.0);
581   Qpos[1] = RF.getOrigin() - RF.getRotationMatrix() * vec3(Delta_X, 0.0, 0.0);
582   Qpos[2] = RF.getOrigin() + RF.getRotationMatrix() * vec3(0.0, Delta_Y, 0.0);
583   Qpos[3] = RF.getOrigin() - RF.getRotationMatrix() * vec3(0.0, Delta_Y, 0.0);
```

```
584   // Find intersection in the four positions
585   real_type iFriction;
586   real_type Friction = 0.0;
587   row_vec3 Qvec(4);
588   for (unsigned i = 0; i < 4; ++i) {
589     TirePose = pointSampling(intersectionTriPtr, Qpos[i], -RF.getZ(), Qvec[i], iFriction);
590     Friction += iFriction;
591     if ( TirePose == 0 ) {
592       // Out of mesh
593       ContactNormal   = vec3(0.0, 0.0, 1.0);
594       ContactFriction = 1.0;
595       calculateContactPoint(Qvec, DefaultH);
596       return;
597     } else if ( TirePose == 2 ) {
598       // Flying over the mesh
599       ContactNormal   = RF.getZ();
600       ContactFriction = 1.0;
601       calculateContactPoint(Qvec, DefaultH);
602       return;
603     }
604   }
605   // Calculate the contact point friction
606   ContactFriction = Friction / 4.0;
607   // Calculate normal of the local track plane
608   ContactNormal = ((Qvec[0] - Qvec[1]).cross(Qvec[2] - Qvec[3])).normalized();
609   // Calculate the real contact point (on the disk)
610   calculateContactPoint(Qvec, DefaultH);
611   return;
612 }
613
614 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
615
616 void
617 MagicFormula::calculateContactPoint(
618   row_vec3  const & Qvec,
619   real_type const   DefaultH
620 ) {
621   if ( TirePose == 0 ) {
622     // Out of mesh
623     vec3 P_star(0.0, 0.0, DefaultH);
624     algorithms::intersectRayPlane( ContactNormal, P_star, RF.getOrigin(), -RF.getZ(),
          ContactPoint );
625     return;
626   } else if ( TirePose == 2 ) {
627     // Flying over the mesh
628     ContactPoint = RF.getOrigin() - RF.getRotationMatrix() * vec3(0.0,0.0,TireGeometry.
          getTireRadius());
629     return;
630   } else {
631     // Calculate first guess of local contact point
632     vec3 P_star(0.0, 0.0, 0.0);
633     for (unsigned i = 0; i < 4; ++i)
634       P_star += Qvec[i];
635     P_star /= 4.0;
636     // Calculate the normal direction on the disk
637     vec3 NormalTireRF(RF.getRotationMatrixInverse(-ContactNormal));
638     vec3 Direction((RF.getRotationMatrix()*vec3( NormalTireRF.x(), 0.0, NormalTireRF.z())).
          normalized());
639     algorithms::intersectRayPlane( ContactNormal, P_star, RF.getOrigin(), Direction, ContactPoint
           );
640   }
641 }
642
643 /*\
644  |   __  __       _ _   _ ____  _     _
```

```
645   |  |  \/  |_   _| | |_(_)  _ \(_)___| | __
646   |  | |\/| | | | | __| | | | | / __| |/ /
647   |  | |  | | |_| | | |_| | |_| | \__ \   <
648   |  |_|  |_|\__,_|_|\__|_|____/|_|___/_|\_\
649   |
650   \*/
651
652   void
653   MultiDisk::getContactPointRF(
654     mat4 & PointRF
655   ) const {
656     vec3 X_versor = RF.getY().cross(SingleContactNormal).normalized();
657     vec3 Y_versor = (SingleContactNormal.cross(X_versor)).normalized();
658     PointRF << X_versor, Y_versor, SingleContactNormal, SingleContactPoint,
659                vec4(0.0, 0.0, 0.0, 1.0).transpose();
660   }
661
662   // . . . . . .
663
664   void
665   MultiDisk::getContactPointRF(
666     matN_mat4 & PointRF
667   ) const {
668     unsigned Xdiv = Precision.getXdiv();
669     unsigned Ydiv = Precision.getYdiv();
670     for (unsigned i = 0; i < Xdiv; ++i) {
671       for (unsigned j = 0; j < Ydiv; ++j) {
672         vec3 X_versor = RF.getY().cross(ContactNormal(i,j)).normalized();
673         vec3 Y_versor = (ContactNormal(i,j).cross(X_versor)).normalized();
674         PointRF(i,j) << X_versor, Y_versor, ContactNormal(i,j), ContactPoint(i,j),
675                         vec4(0.0, 0.0, 0.0, 1.0).transpose();
676       }
677     }
678   }
679
680   // . . . . . .
681
682   void
683   MultiDisk::getRho(
684     real_type & Rho
685   ) const {
686     if ( TirePose == 2 ) {
687       Rho = 0.0;
688     } else {
689       Rho = TireGeometry.getTireRadius()-(RF.getOrigin()-SingleContactPoint).norm();
690     }
691   }
692
693   // . . . . . .
694
695   void
696   MultiDisk::getRho(
697     matN & Rho
698   ) const {
699     unsigned Xdiv = Precision.getXdiv();
700     unsigned Ydiv = Precision.getYdiv();
701     if ( TirePose == 2 ) {
702       Rho = Eigen::MatrixXd::Constant( Xdiv, Ydiv, 0.0 );
703     } else {
704       for (unsigned i = 0; i < Xdiv; ++i) {
705         for (unsigned j = 0; j < Ydiv; ++j) {
706           getRho(Rho(i,j));
707         }
708       }
709     }
```

```
}

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

void
MultiDisk::getRhoDot(
  real_type const & Rho,
  real_type const & Time,
  real_type       & RhoDot
) const {
  if ( TirePose == 2 ) {
    RhoDot = 0.0;
  } else {
    real_type _Rho = 0.0;
    getRho(_Rho);
    RhoDot = (_Rho - Rho) / Time;
  }
}

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

void
MultiDisk::getRhoDot(
  matN      const & Rho,
  real_type const & Time,
  matN            & RhoDot
) const {
  unsigned Xdiv = Precision.getXdiv();
  unsigned Ydiv = Precision.getYdiv();
  if ( TirePose == 2 ) {
    RhoDot = Eigen::MatrixXd::Constant( Xdiv, Ydiv, 0.0 );
  } else {
    for (unsigned i = 0; i < Xdiv; ++i) {
      for (unsigned j = 0; j < Ydiv; ++j) {
        getRhoDot(Rho(i,j), Time, RhoDot(i,j));
      }
    }
  }
}

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

void
MultiDisk::getContactAreaTri(
  RDF::TriangleRoad_list const & intersectionTriPtr,
  real_type                    & Area
) const {
  if ( TirePose == 0 ) {
    Area = quietNaN;
  } else if ( TirePose == 2 ) {
    Area = 0.0;
  } else {
    Area = 0.0;
    for (unsigned i = 0; i < DiskVector.size(); ++i)
      Area += DiskVector[i].getPatchLength( intersectionTriPtr, RF ) * this->getDiskStep();
  }
}

// . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

void
MultiDisk::getContactAreaTriDisk(
  int_type               const   i,
  RDF::TriangleRoad_list const & intersectionTriPtr,
  real_type                    & Area
```

```
775  ) const {
776    if ( TirePose == 0 ) {
777      Area = quietNaN;
778    } else if ( TirePose == 2 ) {
779      Area = 0.0;
780    } else {
781      Area += DiskVector[i].getPatchLength( intersectionTriPtr, RF ) * this->getDiskStep();
782    }
783  }
784
785  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
786
787  void
788  MultiDisk::getContactArea(
789    real_type & Area
790  ) const {
791    if ( TirePose == 2 ) {
792      Area = 0.0;
793    } else {
794      Area = 0.0;
795      for (unsigned i = 0; i < DiskVector.size(); ++i)
796        Area += DiskVector[i].getPatchLength( SingleContactNormal, SingleContactPoint, RF ) * this
               ->getDiskStep();
797    }
798  }
799
800  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
801
802  void
803  MultiDisk::getContactAreaDisk(
804    int_type  const   i,
805    real_type       & Area
806  ) const {
807    if ( TirePose == 2 ) {
808      Area = 0.0;
809    } else {
810      Area = DiskVector[i].getPatchLength( SingleContactNormal, SingleContactPoint, RF ) * this->
           getDiskStep();
811    }
812  }
813
814  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
815
816  void
817  MultiDisk::getContactVolume(
818    real_type & Volume
819  ) const {
820    if ( TirePose == 2 ) {
821      Volume = 0.0;
822    } else {
823      Volume = 0.0;
824      real_type R = TireGeometry.getTireRadius();
825      for (unsigned i = 0; i < DiskVector.size(); ++i) {
826        real_type c = DiskVector[i].getPatchLength( SingleContactNormal, SingleContactPoint, RF );
827        real_type c_2R = c / (2*R);
828        Volume += R*R * ( std::asin(c_2R) - c_2R*std::sqrt(1-c_2R*c_2R)) * this->getDiskStep();
829      }
830    }
831  }
832
833  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
834
835  void
836  MultiDisk::getContactVolumeDisk(
837    int_type  const i,
```

```
838   real_type     & Volume
839 ) const {
840   if ( TirePose == 2 ) {
841     Volume = 0.0;
842   } else {
843     real_type R = TireGeometry.getTireRadius();
844     real_type c = DiskVector[i].getPatchLength( SingleContactNormal, SingleContactPoint, RF );
845     real_type c_2R = c / (2*R);
846     Volume = R*R * ( std::asin(c_2R) - c_2R*std::sqrt(1-c_2R*c_2R)) * this->getDiskStep();
847   }
848 }
849
850 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
851
852 void
853 MultiDisk::setup(
854   RDF::MeshSurface       & Mesh,
855   mat4             const & TM,
856   real_type        const   DefaultH,
857   bool                     print
858 ) {
859   // Set the new reference frame
860   RF.setTotalTransformationMatrix(TM);
861   // Shadow bounding box object
862   Shadow TireShadow(TireGeometry, RF);
863   // Local intersected triangles vector
864   RDF::TriangleRoad_list intersectionTriPtr =
865       Mesh.intersectAABBtree(TireShadow.getAABBPtr());
866   // Update the local road plane (normal, point and realtive camber)
867   // Perform the terrain sampling;
868   gridSampling(intersectionTriPtr, DefaultH);
869   calcSingleContactPoint(DefaultH);
870
871   // Print the results
872   real_type Rho = 0.0;
873   getRho(Rho);
874   real_type RelativeCamber = 0.0;
875   getRelativeCamber(RelativeCamber);
876   real_type ContactArea = 0.0;
877   getContactArea(ContactArea);
878   real_type ContactAreaTri = 0.0;
879   getContactAreaTri(intersectionTriPtr, ContactAreaTri);
880   real_type ContactVolume = 0.0;
881   getContactVolume(ContactVolume);
882   mat4 ContactPointRF;
883   getContactPointRF(ContactPointRF);
884   if (print) {
885     std::cout << "MULTIDISK CONTACT PARAMETERS\n"
886               << "\tYaw angle\n"
887               << "\ t = " << getEulerAngleZ() / G2lib::m_pi << "pi rad\n"
888               << "\tRotation angle\n"
889               << "\tß = " << getEulerAngleY() / G2lib::m_pi << "pi rad\n"
890               << "\tCamber angle\n"
891               << "\Γt = " << getEulerAngleX() / G2lib::m_pi << "pi rad\n"
892               << "\tNormal contact point vector of the local track plane "
893                 "(absolute reference frame)\n"
894               << "\tN = [ " << SingleContactNormal.x() << ", " << SingleContactNormal.y()
895               << ", " << SingleContactNormal.z() << " ]\n"
896               << "\tLocal contact point (absolute reference frame)\n"
897               << "\tP = [ " << SingleContactPoint.x() << ", " << SingleContactPoint.y()
898               << ", " << SingleContactPoint.z() << " ]\n"
899               << "\tLocal contact point reference frame\n"
900               << ContactPointRF << "\n"
901               << "\tRelative camber angle\n"
902               << "\tß = " << RelativeCamber / G2lib::m_pi << "pi rad\n"
```

```
903               << "\tLocal contact point friction\n"
904               << "\tf = " << SingleContactFriction << "\n"
905               << "\tContact point friction matrix\n"
906               << "\tf = " << ContactFriction << "\n"
907               << "\tLocal contact depth (on center point) Rho\n"
908               << "\tD = " << Rho << " m\n"
909               << "\tLocal approximated contact area (calculated with local plane)\n"
910               << "\tA = " << ContactArea << " m^2\n"
911               << "\tLocal approximated contact area (calculated with triangles intersection)\n"
912               << "\tA = " << ContactAreaTri << " m^2\n"
913               << "\tLocal approximated contact volume (calculated with local plane)\n"
914               << "\tV = " << ContactVolume << " m^3\n"
915               << "\tTire position flag = " << TirePose << "\n\n";
916   }
917 }
918
919 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
920
921 void
922 MultiDisk::offsetDisks(
923   Eigen::VectorXd & OffsetVec
924 ) const {
925   for ( unsigned i = 0; i < DiskVector.size(); ++i ) {
926     // Index from Y negative to Y positive
927     OffsetVec[i] = - TireGeometry.getSectionWidth() / 2.0 +
928                      this->getDiskStep() / 2.0 + i * this->getDiskStep();
929   }
930 }
931
932 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
933
934 void
935 MultiDisk::gridSampling(
936   RDF::TriangleRoad_list const & intersectionTriPtr,
937   real_type              const   DefaultH
938  ) {
939   // Storing dimensions
940   unsigned Xdiv = Precision.getXdiv();
941   unsigned Ydiv = Precision.getYdiv();
942   // Storing indexers
943   unsigned i = 0;
944   unsigned j = 0;
945   // Sample on grid pattern
946   real_type SectionWidth_2 = TireGeometry.getSectionWidth() / 2.0 -
947                              this->getDiskStep() / 2.0;
948   real_type Radius = TireGeometry.getTireRadius();
949   for ( real_type y = -SectionWidth_2;
950           y <= SectionWidth_2;
951           y += this->getDiskStep(), ++j ) {
952     for ( real_type x = -Radius;
953             x <=  Radius;
954             x += this->getXstep(), ++i ) {
955       // Find intersection and store results
956       TirePose = pointSampling( intersectionTriPtr,
957                                 RF.getOrigin() + RF.getRotationMatrix()*vec3(x,y,0.0),
958                                 -RF.getZ(),
959                                 ContactPoint(i,j),
960                                 ContactFriction(i,j),
961                                 ContactNormal(i,j) );
962         if ( TirePose == 0 ) {
963           // Out of mesh
964           shiftContactPoint( vec3(0.0,0.0,DefaultH) );
965           setContactNormal( vec3(0.0,0.0,1.0) );
966           ContactFriction = Eigen::MatrixXd::Constant(Xdiv,Ydiv,1.0);
967           return;
```

```
968         } else if ( TirePose == 2 ) {
969           // Flying over the mesh
970           shiftContactPoint( -RF.getRotationMatrix()*vec3(0.0,0.0,TireGeometry.getTireRadius()) )
                  ;
971           setContactNormal( RF.getZ() );
972           ContactFriction = Eigen::MatrixXd::Constant(Xdiv,Ydiv,1.0);
973           return;
974         }
975     }
976     // Update indexer
977     i = 0;
978   }
979   return;
980 }
981
982 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
983
984 void
985 MultiDisk::calcSingleContactPoint(
986   real_type const DefaultH
987 ) {
988   if ( TirePose == 0 ) {
989     // Out of mesh
990     SingleContactNormal   = vec3(0.0, 0.0, 1.0);
991     SingleContactFriction = 1.0;
992     vec3 P_star(0.0, 0.0, DefaultH);
993     algorithms::intersectRayPlane( SingleContactNormal,
994                                    P_star,
995                                    RF.getOrigin(),
996                                    -RF.getZ(),
997                                    SingleContactPoint );
998     return;
999   } else if ( TirePose == 2 ) {
1000    // Flying over the mesh
1001    SingleContactNormal   = vec3(0.0,0.0,1.0);
1002    SingleContactFriction = 1.0;
1003    SingleContactPoint    = RF.getOrigin() -
1004                            RF.getRotationMatrix() * vec3(0.0,0.0,TireGeometry.getTireRadius());
1005    return;
1006  } else {
1007    SingleContactFriction = ContactFriction.mean();
1008    SingleContactNormal   = mean(ContactNormal);
1009    // Calculate first guess of local contact point and contact normal
1010    vec3 P_star = mean(ContactPoint);
1011    // Calculate the normal direction on the disk
1012    vec3 NormalTireRF( RF.getRotationMatrixInverse(-SingleContactNormal) );
1013    vec3 Direction( (RF.getRotationMatrix()*vec3(NormalTireRF.x(), 0.0, NormalTireRF.z())).
           normalized() );
1014    // Find the point
1015    algorithms::intersectRayPlane( SingleContactNormal,
1016                                   P_star,
1017                                   RF.getOrigin(),
1018                                   Direction,
1019                                   SingleContactPoint );
1020  }
1021 }
1022
1023 // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
1024
1025 void
1026 MultiDisk::shiftContactPoint(
1027   vec3 const & Shift
1028 ) {
1029   // Storing indexers
1030   unsigned i = 0;
```

```
1031    unsigned j = 0;
1032    // Sample on grid pattern
1033    real_type SectionWidth_2 = TireGeometry.getSectionWidth()/2.0 - this->getDiskStep()/2.0;
1034    real_type Radius = TireGeometry.getTireRadius();
1035    for ( real_type y = -SectionWidth_2;
1036            y <= SectionWidth_2;
1037            y += this->getDiskStep(), ++j ) {
1038      for ( real_type x = -Radius;
1039              x <=  Radius;
1040              x += this->getXstep(), ++i ) {
1041        ContactPoint(i,j) = RF.getOrigin() + RF.getRotationMatrix()*vec3(x, y, 0.0) + Shift;
1042      }
1043      // Update indexer
1044      i = 0;
1045    }
1046  }
1047
1048  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
1049
1050  void
1051  MultiDisk::setContactNormal(
1052    vec3 const & Value
1053  ) {
1054    // Storing indexers
1055    unsigned Xdiv = Precision.getXdiv();
1056    unsigned Ydiv = Precision.getYdiv();
1057    for (unsigned i = 0; i < Xdiv; ++i) {
1058      for (unsigned j = 0; j < Ydiv; ++j) {
1059        ContactNormal(i,j) = Value;
1060      }
1061    }
1062  }
1063
1064  // . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
1065
1066  vec3
1067  MultiDisk::mean(
1068    matN_vec3 const & Mat
1069  ) const {
1070    return Mat.sum() / (Mat.rows()*Mat.cols());
1071  }
1072
1073 }  // namespace PatchTire
```

Codice dei Tests C

C.1 Tests Geometrici

C.1.1 Geometry-test1.cc

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

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

C.1.2 Geometry-test2.cc

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

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

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

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

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

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

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

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

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

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

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

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

C.1.3 Geometry-test3.cc

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

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

C.1.4 Geometry-test4.cc

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

C.2 Tests per il Modello Magic Formula

C.2.1 MagicFormula-test1.cc

```
// PATCH EVALUATION TEST 1 - LOAD THE DATA FROM THE RDF FILE THEN PRINT IT INTO
// A FILE Out.txt. THEN CHARGE THE TIRE DATA AND ASSOCIATE THE CURRENT MESH TO
// IT.

#include <chrono>    // chrono - STD Time Measurement Library
#include <fstream>   // fStream - STD File I/O Library
#include <iostream>  // Iostream - STD I/O Library

#include "PatchTire.hh" // Tire Data Processing
#include "RoadRDF.hh"   // Tire Data Processing
#include "TicToc.hh"    // Processing Time Library

// Main function
int
main() {

  try {

    // Instantiate a TicToc object
    TicToc tictoc;

    std::cout
      << "MAGIC FORMULA TIRE TEST 1 - CHECK INTERSECTION ON UNKNOWN MESH.\n\n";

    // Load .rdf File
    RDF::MeshSurface Road("./RDF_files/Eight.rdf");

    // Print OutMesh.txt file
    // Road.printData("OutMesh.txt");

    // Initialize the Magic Formula Tire
    PatchTire::Tire* TireSD = new PatchTire::MagicFormula(205, 60, 15);

    // Display current tire data on command line
    TireSD->print(std::cout);

    // Orient the tire in the space
    RDF::real_type Yaw    = 0*G2lib::m_pi;
    RDF::real_type Camber = 0*G2lib::m_pi;

    // Transformation matrix for X and Z-axis rotation
    TireGround::mat3 Rot_Z;
    Rot_Z << cos(Yaw), -sin(Yaw), 0,
             sin(Yaw),  cos(Yaw), 0,
                    0,        0, 1;
    TireGround::mat3 Rot_X;
    Rot_X << 1,           0,            0,
             0, cos(Camber), -sin(Camber),
             0, sin(Camber),  cos(Camber);
    // Update Rotation Matrix
    TireGround::mat3 RotMat = Rot_Z * Rot_X;

    TireGround::vec3 Origin(-10.8, 19.0, 0.6); //0.8, 19.0, 0.26
    PatchTire::ReferenceFrame Pose(Origin, RotMat);

    // Start chronometer
    tictoc.tic();

    // Set an orientation and calculate parameters
```

```
60    TireSD->setup( Road, Pose.getTotalTransformationMatrix(), 0.235, true);
61
62    /* Example: Get results
63    |  // Variable initialization
64    |  PatchTire::vec3 N(vec3_NaN);
65    |  PatchTire::vec3 P(vec3_NaN);
66    |  PatchTire::real_type ContactFriction = 0.0;
67    |  PatchTire::real_type Rho = 0.0;
68    |  PatchTire::real_type RhoDot = 0.0;
69    |  PatchTire::real_type RelativeCamber = 0.0;
70    |  PatchTire::real_type ContactArea = 0.0
71    |  PatchTire::real_type ContactVolume = 0.0;
72    |  PatchTire::real_type RelativeCamber = 0.0;
73    |
74    |  // Variable calculation
75    |  TireSD.getContactNormal(N);
76    |  TireSD.getContactPoint(P);
77    |  TireSD.getContactFriction(ContactFriction);
78    |  TireSD.getRho(Rho);
79    |  TireSD.getRhoDot(PreviousRho,Time,RhoDot);
80    |  TireSD.getRelativeCamber(RelativeCamber);
81    |  TireSD.getContactArea(Area);
82    |  TireSD.getContactVolume(Volume);
83    |  TireSD.getRelativeCamber(RelativeCamber);
84    */
85
86    // Stop chronometer
87    tictoc.toc();
88
89    // This constructs a duration object using milliseconds
90    std::cout
91      << "Execution time = " << tictoc.elapsed_ms() << " ms\n"
92      << "\nCheck the results...\n"
93      << "\nMAGIC FORMULA TIRE TEST 1: Completed\n\n";
94
95  } catch ( std::exception const & exc ) {
96    std::cerr << exc.what() << '\n';
97  }
98  catch (...) {
99    std::cerr << "Unknown error\n";
100 }
101 }
```

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, 1.0);
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    // Initialize the Magic Formula Tire
35    PatchTire::Tire* TireSD = new PatchTire::MagicFormula(205, 60, 15);
36
37    // Display current tire data on command line
38    TireSD->print(std::cout);
39
40    // Orient the tire in the space
41    RDF::real_type Yaw    = 0.0*G2lib::m_pi;
42    RDF::real_type Camber = 0.0*G2lib::m_pi;
43
44    // Transformation matrix for X and Z-axis rotation
45    TireGround::mat3 Rot_Z;
46    Rot_Z << cos(Yaw), -sin(Yaw), 0,
47             sin(Yaw),  cos(Yaw), 0,
48                    0,        0, 1;
49    TireGround::mat3 Rot_X;
50    Rot_X << 1,           0,            0,
51             0, cos(Camber), -sin(Camber),
52             0, sin(Camber),  cos(Camber);
53    // Update Rotation Matrix
54    TireGround::mat3 RotMat = Rot_Z * Rot_X;
55
56    TireGround::vec3 Origin( 50.0, 10.0, 0.26+0.5 );
57    PatchTire::ReferenceFrame Pose(Origin, RotMat);
58
59    // Start chronometer
60    tictoc.tic();
61
62    // Set an orientation and calculate parameters (true = print results)
63    TireSD->setup( Road, Pose.getTotalTransformationMatrix(), 0.235, true);
64
65    /* Example: Get results
66    |  // Variable initialization
67    |  PatchTire::vec3 N(vec3_NaN);
68    |  PatchTire::vec3 P(vec3_NaN);
69    |  PatchTire::real_type ContactFriction = 0.0;
70    |  PatchTire::real_type Rho = 0.0;
71    |  PatchTire::real_type RhoDot = 0.0;
72    |  PatchTire::real_type ContactArea = 0.0
73    |  PatchTire::real_type ContactVolume = 0.0;
74    |  PatchTire::real_type RelativeCamber = 0.0;
75    |
76    |  // Variable calculation
77    |  TireSD.getContactNormal(N);
78    |  TireSD.getContactPoint(P);
79    |  TireSD.getContactFriction(ContactFriction);
80    |  TireSD.getRho(Rho);
81    |  TireSD.getRhoDot(PreviousRho,Time,RhoDot);
82    |  TireSD.getContactArea(Area);
83    |  TireSD.getContactVolume(Volume);
84    |  TireSD.getRelativeCamber(RelativeCamber);
```

```
85     */
86
87     // Stop chronometer
88     tictoc.toc();
89
90     // This constructs a duration object using milliseconds
91     std::cout
92       << "Execution time = " << tictoc.elapsed_ms() << " ms\n"
93       << "\nCheck the results...\n"
94       << "\nMAGIC FORMULA TIRE TEST 2: Completed\n";
95
96   } catch ( std::exception const & exc ) {
97     std::cerr << exc.what() << '\n';
98   }
99   catch (...) {
100    std::cerr << "Unknown error\n";
101  }
102 }
```

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] Organización Internacional de Normalización (Ginebra). *Road Vehicles, Vehicle Dynamics and Road-holdin Ability: Vocabulary*. ISO, 1991. ISBN: ISBN 9781439838983.

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

[9] Georg Rill. *Road Vehicle Dynamics - Fundamentals and Modeling*. Set. 2011. ISBN: ISBN 9781439838983.

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

[11] Dieter Schramm, Manfred Hiller e Roberto Bardini. *Vehicle Dynamics: Modeling and Simulation*. Springer Publishing Company, Incorporated, 2014. ISBN: 3540360441.