



UNIVERSITÀ 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

Candidato:

Davide Stocco

Co-relatore:

Dott. Ing. Matteo Ragni

Anno Accademico 2019 · 2020

Sommario

This dissertation details ...

Indice

1	Introduzione	1
1.1	Obiettivi della tesi	1
1.2	Il problema	1
2	La superficie stradale	5
2.1	Il formato RDF	6
2.1.1	Superfici semplici	6
2.1.2	Superfici complesse	8
2.2	Analisi sintattico-grammaticale del formato RDF	10
3	Lo pneumatico	13
3.1	Geometria	13
3.2	Modelli di pneumatico	14
3.2.1	Il modello di Pacejka	16
3.3	Contatto con la superficie stradale	17
3.3.1	Modello di pneumatico a disco singolo	18
3.3.1.1	Contatto di Rill	18
3.3.1.2	Contatto ponderato in base all'area d'intersezione	22
3.3.2	Modello di pneumatico a più dischi	24
3.3.2.1	Contatto ponderato in base all'area d'intersezione	24
3.3.2.2	Contatto tramite campionamento	26
4	Algoritmi	29
4.1	Struttura ad albero di tipo "Bounding Volume Hierarchy"	29
4.1.1	Struttura di tipo "Minimum Bounding Box"	30
4.1.1.1	Struttura di tipo "Axis Aligned Bounding Box"	30

4.1.1.2	Struttura di tipo " <i>Arbitrarily Oriented Bounding Box</i> "	31
4.1.1.3	Struttura di tipo " <i>Object Oriented Bounding Box</i> "	31
4.1.2	Intersezione tra alberi di tipo AABB	31
4.2	Algoritmi geometrici	32
4.2.1	Intersezione tra entità geometriche	34
4.2.1.1	Intersezione punto-segmento	34
4.2.1.2	Intersezione punto-cerchio	34
4.2.1.3	Intersezione segmento-circonferenza	36
4.2.1.4	Intersezione piano-piano	38
4.2.1.5	Piano-Segmento e Piano-Raggio	41
4.2.1.6	Intersezione piano-triangolo	42
4.2.1.7	Intersezione raggio-triangolo	43
5	La libreria TireGround	49
5.1	Organizzazione	49
5.1.1	Gestione della superficie stradale	49
5.1.2	Gestione dei modelli di pneumatico	51
5.2	Librerie Esterne	55
5.2.1	Eigen3	56
5.2.2	Clothoids	56
5.3	Utilizzo	56
5.4	Prestazioni	60
6	Conclusioni e Lavoro Futuro	65
A	Convenzioni e notazioni	67
A.0.1	Sistemi di riferimento	67
A.0.2	Matrice di trasformazione	68
B	Documentazione della libreria TireGround	71
C	Codice dei test	163
C.1	Test geometrici	163
C.1.1	Geometry-test1.cc	163
C.1.2	Geometry-test2.cc	164

C.1.3	Geometry-test3.cc	166
C.1.4	Geometry-test4.cc	167
C.2	Tests per il modello a singolo disco	168
C.2.1	MagicFormula-test1.cc	168
C.2.2	MagicFormula-test2.cc	169
C.3	Tests per il modello a più dischi	170
C.3.1	MultiDisk-test1.cc	170
C.3.2	MultiDisk-test2.cc	172
Bibliografia		175

Elenco delle figure

2.1	Esempio di superficie rappresentata tramite <i>mesh</i> triangolare.	11
2.2	Intersezione stradale rappresentata tramite <i>mesh</i> triangolare.	11
3.1	Esempio di misure, secondo la notazione ETRTO, riportate sulla spalla dello pneumatico.	15
3.2	Forze e coppie generate dal contatto pneumatico/strada.	15
3.3	Curve caratteristiche generiche degli pneumatici derivate con il metodo della <i>Magic Formula</i>	17
3.4	Geometria del contatto pneumatico-strada.	19
3.5	Punti campionati nel piano locale della superficie stradale.	20
3.6	Punti di contatto P_{PL} e P_{MF} in relazione alla normale $e_{n_{XZ}}$ e al tipo di terreno.	21
3.7	Dato un generico triangolo che, intersecando il piano in cui giace il disco, crea il segmento dato dai punti A e B , l'area di intersezione è la regione racchiusa dai segmenti $B'B$, AB , AA' e dall'arco di circonferenza $A'B'$	22
3.8	I versori normali \vec{N}_A , \vec{N}_B , \vec{N}_C e \vec{N}_D vengono ponderati in base all'area delle rispettive regioni d'intersezione A , B , C e D	23
3.9	Disposizione dei dischi.	24
3.10	Pneumatico rappresentato da dischi a raggio uniforme. Notare il disco fittizio giacente sul piano XZ in linea tratteggiata.	25
3.11	Momento creato dalla morfologia del terreno.	25
3.12	Normali associate ai vari dischi dello pneumatico.	26
3.13	Campionamento della <i>mesh</i> triangolare in corrispondenza del piano in cui giace l' i -esimo disco. I raggi partono dall'asse x_C in direzione z_C	27
4.1	Esempio di albero di tipo AABB.	30

4.2	Schema del problema di intersezione punto-segmento	34
4.3	Schemi per l' <i>output</i> dell'intersezione punto-segmento.	34
4.4	Schema del codice per l'intersezione punto-segmento.	35
4.5	Schema del problema di intersezione punto-cerchio.	35
4.6	Schemi per l' <i>output</i> dell'intersezione punto-cerchio.	36
4.7	Schemi del codice per l'intersezione punto-cerchio.	36
4.8	Schema del problema di intersezione punto-circonferenza.	37
4.9	Schemi per l' <i>output</i> dell'intersezione segmento-cerchio.	38
4.10	Schema dello pseudocodice per l'intersezione segmento-cerchio. . . .	39
4.11	Schemi del problema di intersezione piano-piano.	40
4.12	Vettori dei piani P_1 , P_2 e della retta L	40
4.13	Schema dello pseudocodice per l'intersezione piano-piano.	41
4.14	Vettori dei piani P_1 , P_2 e della retta L	42
4.15	Schema dello pseudocodice per l'intersezione piano-segmento.	43
4.16	Schema dello pseudocodice per l'intersezione piano-triangolo.	43
4.17	Schema del problema di intersezione raggio-triangolo.	44
4.18	Cambiamento di coordinate nell'algoritmo di Möller-Trumbore. . . .	45
4.19	Schemi per l' <i>output</i> dell'intersezione punto-cerchio.	47
4.20	Schema dello pseudocodice per l'intersezione raggio-triangolo con <i>back-face culling</i>	47
6.1	Schema strutturale del modello "a spazzola" (<i>brush model</i>).	66
A.1	Rappresentazione degli assi del sistema di riferimento della vettura secondo la convenzione ISO-V.	67
A.2	Rappresentazione degli assi del sistema di riferimento dello pneumatico secondo la convenzione ISO-C.	68

Elenco delle tabelle

5.1	Attributi della classe BBox2D.	50
5.2	Attributi della classe Triangle3D.	50
5.3	Attributi della classe TriangleRoad.	51
5.4	Attributi della classe MeshSurface.	51
5.5	Attributi della classe Disk.	52
5.6	Attributi della classe ETRTO.	53
5.7	Attributi della classe ReferenceFrame.	53
5.8	Attributi della classe Shadow.	54
5.9	Attributi della classe SamplingGrid.	54
5.10	Attributi della classe Tire.	54
5.11	Attributi della classe MagicFormula.	55
5.12	Attributi della classe MultiDisk.	55
5.13	Tempi per il modello di pneumatico a singolo disco Tire::MagicFormula nel caso di <i>mesh</i> densa.	61
5.14	Tempi per il modello di pneumatico a singolo disco Tire::MagicFormula nel caso di <i>mesh</i> poco densa.	61
5.15	Tempi per il modello di pneumatico a più dischi Tire::MultiDisk nel caso di <i>mesh</i> densa.	62
5.16	Tempi per il modello di pneumatico a più dischi Tire::MultiDisk nel caso di <i>mesh</i> poco densa.	63

Elenco degli acronimi

AABB Axis Aligned Bounding Box	30
ADAS Advanced Driver-Assistance Systems	2
AOBB Arbitrarily Oriented Bounding Box	31
BB Bounding Box	23
BVH Bounding Volume Hierarchy	29
CAD Computer-Aided Design	32
CAE Computer-Aided Engineering	33
CAGD Computer-Aided Geometric Design	33
CAM Computer-Aided Manufacturing	32
ETRTO European Tyre and Rim Technical Organisation	3
GIS Geographic Information Systems	33
HIL Hardware in the Loop	2
ISO International Organization for Standardization	67
MBB Minimum Bounding Box	30
RDF Road Data File	5

1.1 Obiettivi della tesi

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* nel 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 uno o più dischi indeformabili) 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 sicurez-

za. Infatti, il modello digitale può essere facilmente riconfigurato e analizzato, al contrario invece del 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 a caratterizzare il comportamento del veicolo con un livello di dettaglio sufficiente. 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 di 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 però un compito estremamente arduo e a talvolta anche impossibile. Esistono quindi modelli empirici come il modello della *Magic Formula* di Hans Pacejka, che cercano di imitare il comportamento reale 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 è stato 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 pneumatico scelto da [4] definisce l'obiettivo di questo lavoro. In particolare, lo scopo è quello di implementare una libreria in linguaggio C++ che con alcuni *input*, come la denominazione *European Tyre and Rim Technical Organisation* (ETRTO) e la posizione nello spazio dello pneumatico, calcoli i dati relativi all'interazione dello stesso con strada.

Oltre allo pneumatico, la superficie stradale rappresenta il secondo importante elemento che definisce il contatto. Perché una superficie stradale possa essere facilmente utilizzata da un calcolatore deve essere prima discretizzata. La discretizzazione in questo caso avviene mediante la rappresentazione della superficie stessa in una *mesh* triangolare. La *mesh*, è contenuta in un *file* di 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 triangolo.

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

2.1 Il formato RDF

2.1.1 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 composta 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*. La sezione [MODEL] viene invece utilizzata per specificare la morfologia della superficie stradale, che può essere 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 i parametri generali e specifici per il tipo di superficie stradale. Possono essere:

- 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 mo-

dello 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 dell'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 sotto blocco chiamato (XZ_DATA) e costituito da tre colonne di dati numerici:

- La colonna 1 è un insieme di valori X in ordine crescente.
- Le colonne 2 e 3 sono insiemi di rispettivi valori Z per la traccia sinistra e destra.

Esempio:

```
1  [PARAMETERS]
2  MU = 1.0
3  OFFSET = 0.0
4  ROTATION_ANGLE_XY_PLANE = 0.0
5
6  { X_road  Z_left  Z_right }
7  (XZ_DATA)
8  -1.0e04  0  0
9  0.0500   0  0
10 0.1000   0  0
```

```

11  0.1500  0 0
12  ...  ...  ...

```

- Sinusoide:

La strada a superficie sinusoidale è implementata come:

$$z(x) = \frac{H}{2} \left(1 - \cos \left(\frac{2\pi \cdot (x - x_i)}{L} \right) \right) \quad (2.1)$$

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 dall'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.

2.1.2 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 la superficie stradale si utilizzerà dunque una *mesh* poligonale. Quest'ultima può essere rappresentata utilizzando diversi metodi per memorizzare i dati dei vertici, bordi e facce. Nel caso specifico si andrà ad utilizzare una rappresentazione del tipo faccia-vertice. La *mesh* faccia-vertice rappresenta un oggetto come un insieme di facce e un insieme di vertici. Questa rappresentazione è generalmente

la più utilizzata infatti permette una ricerca esplicita dei vertici di una faccia e delle facce che circondano un vertice.

Per descrivere una superficie stradale composta da una *mesh* di triangoli si utilizzerà quindi la seguente struttura dati:

- [NODES] (Vertici): presenti nella prima sezione, vengono descritti sotto forma di una quartina (id, x, y, z) data dal numero di identificazione e dalle coordinate nello spazio.
- [ELEMENTS] (Facce): presenti nella seconda sezione, 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 coefficiente di attrito presente nella faccia.

Esempio:

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

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

- X_SCALE: riscalda i punti delle coordinate dei nodi lungo l'asse X .
- Y_SCALE: riscalda i punti delle coordinate dei nodi lungo l'asse Y .
- Z_SCALE: riscalda i punti delle coordinate dei nodi lungo l'asse Z .

- **ORIGIN**: definisce la posizione dell'origine del 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
```

2.2 Analisi sintattico-grammaticale del formato RDF

L'analisi sintattico-grammaticale è un processo che analizza un flusso continuo di dati in ingresso (letti per esempio da un *file*) in modo da determinare la correttezza della sua struttura grazie ad una data grammatica formale. Il programma che esegue questo compito viene chiamato *parser*. Nella maggior parte dei casi l'analisi sintattica opera su una sequenza di *tokens* in cui l'analizzatore lessicale spezzetta l'*input*.

Nel lavoro svolto è stato creato un algoritmo per eseguire l'analisi sintattico-grammaticale dei *file* di tipo RDF. Purtroppo, come precedentemente affermato, 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 programma che rilevi solo i nodi (**[NODES]**), li salvi temporanea-

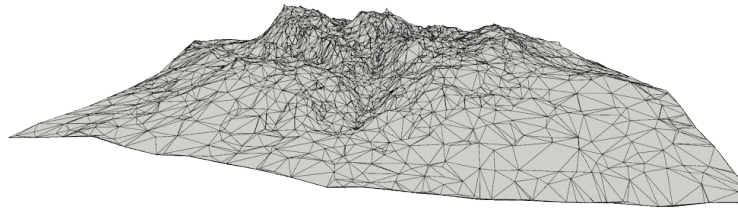


FIGURA 2.1: Esempio di superficie rappresentata tramite *mesh* triangolare.

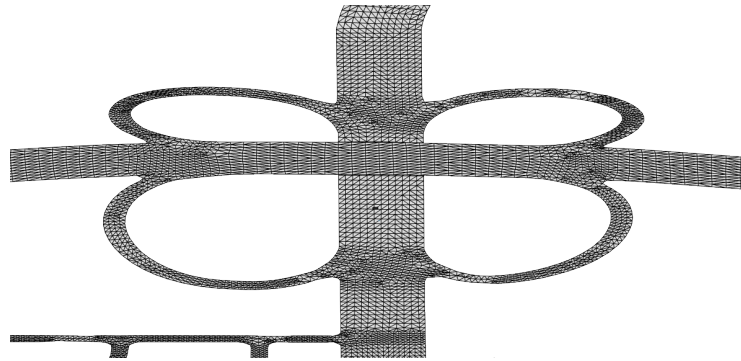


FIGURA 2.2: Intersezione stradale rappresentata tramite *mesh* triangolare.

mente e, dopo aver immagazzinato anche i dati relativi agli elementi ([ELEMENTS]), istanzi un oggetto *mesh*, composto dai nodi dichiarati nella sezione degli 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.

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

3.1 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 uno pneumatico generico la notazione che identificherà la geometria sarà del tipo $a / b R c$, 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 denominazione ETRTO: 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 una delle seguenti formule:

$$\phi_e = \frac{2ab}{25.4} + c \quad [\text{in}] \quad (3.1)$$

$$\phi_e = 2ab + 25.4c \quad [\text{mm}] \quad (3.2)$$

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

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

3.2 Modelli di pneumatico

Le forze di contatto tra la superficie stradale e lo pneumatico possono essere descritte da un vettore di forza risultante applicato in un punto specifico dell'impronta di contatto e da una coppia risultante, come illustrato nella Figura 3.2.

Come componenti cruciali per la movimentazione dei veicoli e il comportamento di guida, le forze degli pneumatici richiedono particolare attenzione soprattutto perché dev'essere considerato anche il comportamento non stazionario. Attualmente, è possibile suddividere i modelli di pneumatico in tre gruppi:

- modelli matematici;



FIGURA 3.1: Esempio di misure, secondo la notazione ETRTO, riportate sulla spalla dello pneumatico.

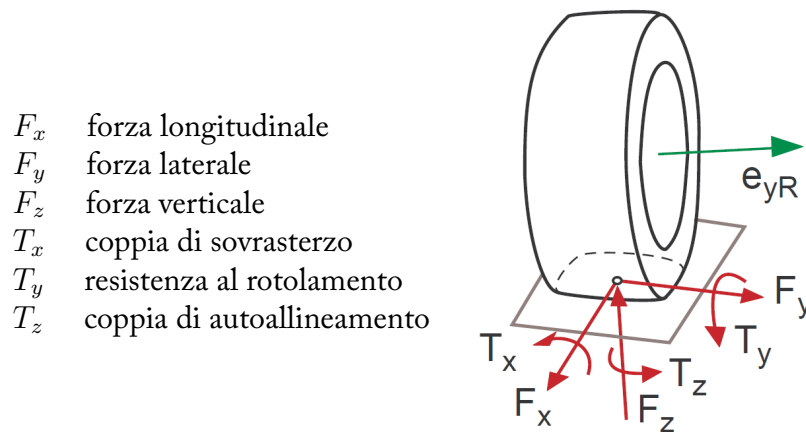


FIGURA 3.2: Forze e coppie generate dal contatto pneumatico/strada.

Da: Rill, *Road Vehicle Dynamics - Fundamentals and Modeling*.

- modelli fisici;
- combinazione dei precedenti.

La prima tipologia di modello tenta di rappresentare le caratteristiche fisiche dello pneumatico attraverso una descrizione puramente matematica. Pertanto, questo tipo di modelli parte da un curve caratteristiche ricavate sperimentalmente e cercano di derivare un comportamento approssimativo dall'interpolazione di un grande insieme di dati. Un esempio ben noto di questo approccio è il **modello di Pacejka**

o *Magic Formula* [8]. Questo tipo di modellazione è adatta per la simulazione di guida in cui il comportamento di interesse è per lo più la manovrabilità 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 preelaborazione 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 basano sulla *Magic Formula* 6.2.

3.2.1 Il modello di Pacejka

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 più volte 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. La forma generale della funzione può essere scritta come:

$$y(x) = D \sin\{C \arctan[B(x + S_h) - E(B(x + S_h) - \arctan(B(x + S_h)))]\} + S_v \quad (3.3)$$

dove i fattori rappresentano:

- B la rigidità;
- C la forma;
- D il valore massimo della forza o coppia;
- E la curvatura in corrispondenza del valore massimo;
- S_v lo spostamento in verticale della curva caratteristica;
- S_h lo spostamento in orizzontale della curva caratteristica.

e dove $y(x)$ può rappresentare 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 3.3 sono illustrate le curve caratteristiche generiche degli pneumatici derivate

con il metodo della *Magic Formula*.

Per poter utilizzare la *Magic Formula* è necessario conoscere:

- la geometria dello pneumatico;
- lo slittamento (o *slip*);
- la forza verticale applicata allo pneumatico;
- la penetrazione in corrispondenza del punto di contatto e la sua derivata nel tempo;
- l'inclinazione tra piano strada e sistema di riferimento del centro ruota (angolo di camber relativo).

È proprio nell'inclinazione tra piano strada e sistema di riferimento del centro ruota che si porrà una maggiore attenzione in quanto elemento fondamentale per ricavare l'effettivo punto di contatto dell'interazione pneumatico/strada.

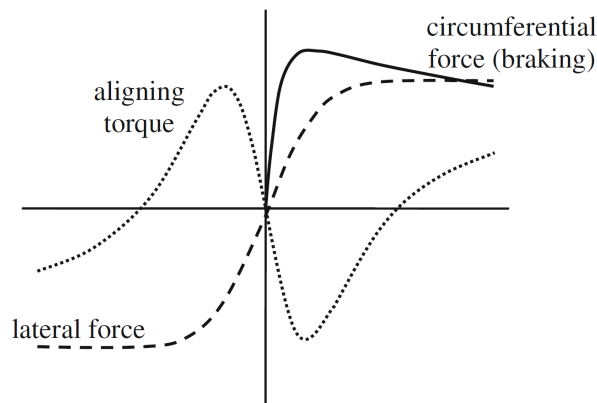


FIGURA 3.3: Curve caratteristiche generiche degli pneumatici derivate con il metodo della *Magic Formula*.

Da: Schramm, Hiller e Bardini, *Vehicle Dynamics: Modeling and Simulation*.

3.3 Contatto con la superficie stradale

Si analizzeranno ora le quattro metodologie di complessità crescente per ricavare l'inclinazione del piano locale e i punti di contatto sulla superficie stradale P_{PL} , nonché sulla circonferenza del disco indeformabile P_{MF} dove effettivamente agiranno le forze ricavate mediante la *Magic Formula*. Dapprima si utilizzerà un metodo a disco singolo presentato in [9], successivamente si passerà ad un modello a più dischi,

così da coprire una superficie stradale maggiore e avere quindi risultati più precisi, soprattutto in prossimità di variazioni repentine del manto stradale.

3.3.1 Modello di pneumatico a disco singolo

3.3.1.1 Contatto di Rill

Piano locale 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} , che viene calcolata istante per istante risolvendo le equazioni dinamiche del sistema ottenuto nel Capitolo 2 in [4]. Supponendo che il profilo stradale sia rappresentato da una funzione arbitraria a due coordinate spaziali del tipo:

$$z = z(x, y) \quad (3.4)$$

su una superficie irregolare, il punto di contatto con il piano locale P_{PL} non può essere calcolato direttamente. Nel metodo a disco singolo presentato in [9] da Rill, come prima approssimazione si identifica un punto di contatto P^* come una semplice traslazione del centro ruota M :

$$P^* = M - R_0 \mathbf{e}_{zC} \begin{bmatrix} x^* \\ y^* \\ z^* \end{bmatrix} \quad (3.5)$$

dove R_0 è il raggio dello pneumatico indeformato ed \mathbf{e}_{zC} è il vettore unitario che definisce l'asse z_C del sistema di riferimento della ruota.

La prima stima del sistema di riferimento del punto di contatto RF_{P^*} è una terna con origine in P^* e la medesima orientazione degli assi del sistema di riferimento della ruota. Si noti dunque che l'origine di RF_{P^*} corrisponde alla proiezione lungo l'asse z_C del sistema di riferimento della ruota.

$$RF_{P^*} = \left[\begin{array}{ccc|c} [R_{RF_{wh}}] & x^* \\ & y^* \\ & z^* \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \quad (3.6)$$

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

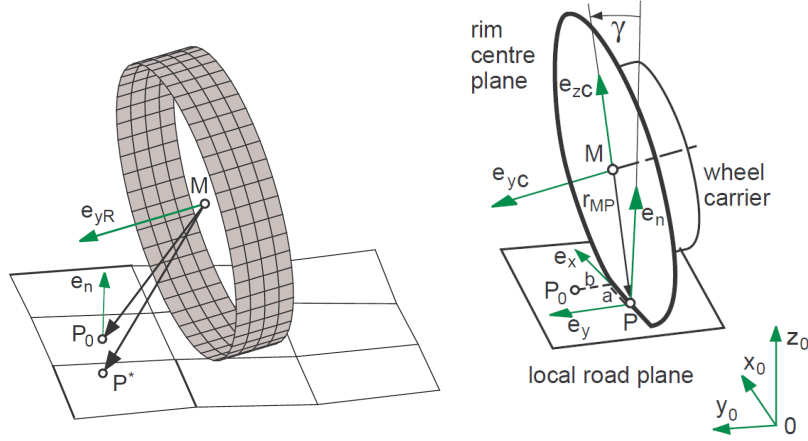


FIGURA 3.4: Geometria del contatto pneumatico-strada.
Da: Rill, *Road Vehicle Dynamics - Fundamentals and Modeling*.

campionamento $(Q_1^*, Q_2^*, Q_3^*, Q_4^*)$, rappresentati graficamente in Figura 3.5. I punti di campionamento sono definiti nel sistema di riferimento temporaneo del punto di contatto RF_{P^*} ; lo spostamento longitudinale e laterale sono definiti dall'origine, ovvero dallo stesso P^* . I vettori di spostamento sono definiti come:

$$\begin{aligned} r_{Q_{1,2}^*} &= \pm \Delta x e_{xP^*} = \pm \Delta x e_{xC} \\ r_{Q_{3,4}^*} &= \pm \Delta y e_{yP^*} = \pm \Delta y e_{yC} \end{aligned} \quad (3.7)$$

e quindi, i quattro punti di campionamento sono:

$$\begin{aligned} Q_{1,2}^* &= P^* \pm r_{Q_{1,2}^*} = P^* \pm \Delta x e_{xC} \\ Q_{3,4}^* &= P^* \pm r_{Q_{3,4}^*} = P^* \pm \Delta y e_{yC} \end{aligned} \quad (3.8)$$

Al fine di campionare il terreno nel modo più efficace possibile, le distanze di Δx e Δy , dell'equazione precedente, vengono regolate in base al raggio indeformato R_0 e alla larghezza B dello pneumatico. I valori di queste due quantità possono essere trovate in [9] e sono $\Delta x = 0.1R_0$ e $\Delta y = 0.3B$. Attraverso questa definizione, si può ottenere un comportamento sufficientemente realistico durante la simulazione.

Ora, la componente z in corrispondenza dei quattro punti campione, viene valutata attraverso la funzione $z(x, y)$ precedentemente definita. Quindi, aggiornando la terza coordinata dei punti di campionamento Q_i^* , si ottengono i corrispondenti punti campione Q_i sulla superficie. La linea fissata dai punti Q_1, Q_2 e Q_3, Q_4 , può ora essere utilizzata per definire la normale al piano strada locale (Figura 3.6).

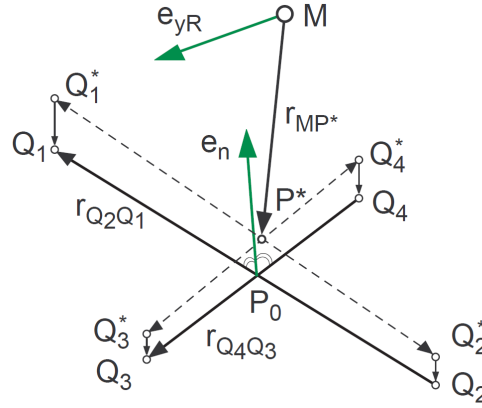


FIGURA 3.5: Punti campionati nel piano locale della superficie stradale.
Da: Rill, *Road Vehicle Dynamics – Fundamentals and Modeling*.

Pertanto, il vettore normale è definito come:

$$\mathbf{e}_n = \frac{\mathbf{r}_{Q_1Q_2} \times \mathbf{r}_{Q_4Q_3}}{|\mathbf{r}_{Q_1Q_2} \times \mathbf{r}_{Q_4Q_3}|} \quad (3.9)$$

Ora, i versori \mathbf{e}_x ed \mathbf{e}_y , che descrivono l'inclinazione del piano locale nel possono essere ottenuti dalle seguenti equazioni:

$$\mathbf{e}_x = \frac{\mathbf{e}_{yC} \times \mathbf{e}_n}{|\mathbf{e}_{yC} \times \mathbf{e}_n|} \quad \mathbf{e}_y = \mathbf{e}_n \times \mathbf{e}_x \quad (3.10)$$

dove sono $\mathbf{r}_{Q_2Q_1}$ e $\mathbf{r}_{Q_4Q_3}$ sono i vettori che puntano rispettivamente da Q_1 a Q_2 e da Q_3 a Q_4 . Applicando la (3.10) è ora possibile calcolare i vettori unitari \mathbf{e}_x e \mathbf{e}_y del piano locale di contatto. Per definire univocamente il piano strada, oltre alla normale calcolata in (3.9), viene utilizzato il punto P_n dato dal valore medio delle tre coordinate spaziali dei quattro punti campione.

$$P_n = \frac{1}{4} \begin{bmatrix} \sum_{i=1}^4 x_i \\ \sum_{i=1}^4 y_i \\ \sum_{i=1}^4 z_i \end{bmatrix} \quad (3.11)$$

Punti di contatto Infine, è necessario ricondursi alle condizioni tali per cui il modello di Pacejka è valido trovando il punto di contatto sul piano strada locale P_{PL} e il punto di contatto sulla circonferenza del disco indeformabile P_{MF} dove effettivamente agiranno le forze ricavate mediante la *Magic Formula*. Si troverà dapprima la componente della normale al piano strada $\mathbf{e}_{n_{XZ}}$ sul piano in cui giace il singolo

disco indeformabile. P_{MF} sarà dunque trovato a partire dal centro ruota M , moltiplicando scalarmente il versore $-e_{n_{XZ}}$ per il raggio del disco indeformabile R_0 , ovvero:

$$P_{MF} = -R_0 e_{n_{XZ}} \quad (3.12)$$

Come illustrato in Figura 3.6, il punto di contatto sul piano strada locale P_{PL} viene invece calcolato sfruttando un algoritmo di intersezione piano-raggio (che si tratterà nel Capitolo 4). P_{PL} giacerà dunque sulla proiezione in direzione $-e_{n_{XZ}}$ del punto M sulla retta individuata dal punto P_n e normale $e_{n_{XZ}}$.

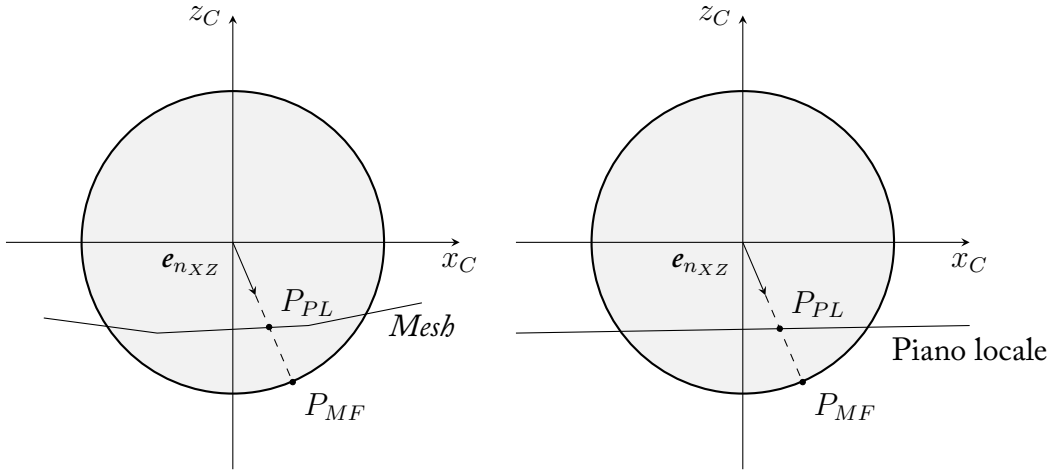


FIGURA 3.6: Punti di contatto P_{PL} e P_{MF} in relazione alla normale $e_{n_{XZ}}$ e al tipo di terreno.

Infine si possono unire tutte le componenti del piano di riferimento del punto di contatto P_{MF} ottenendo il relativo sistema di riferimento:

$$RF_{MF} = \left[\begin{array}{ccc|c} [e_x] & [e_y] & [e_z] & x_{P_{MF}} \\ & & & y_{P_{MF}} \\ & & & z_{P_{MF}} \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \quad (3.13)$$

Attraverso questo approccio, la normale del piano strada locale e_n insieme al punto di contatto sul piano strada locale P_{PL} e al punto di contatto sulla circonferenza del disco indeformabile P_{MF} , sono in grado di rappresentare l'irregolarità della strada in modo soddisfacente ma approssimativo, infatti, bordi taglienti o discontinuità del manto stradale saranno involontariamente filtrate da questo approccio.

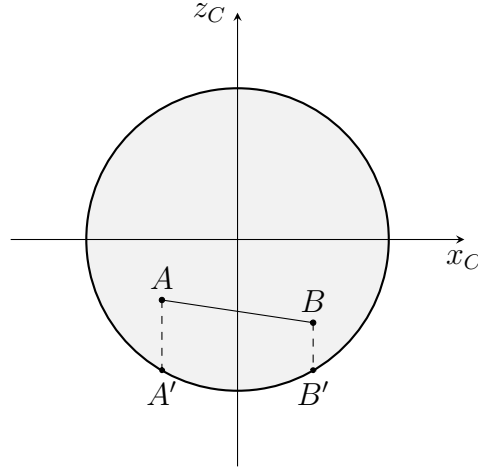


FIGURA 3.7: Dato un generico triangolo che, intersecando il piano in cui giace il disco, crea il segmento dato dai punti A e B , l'area di intersezione è la regione racchiusa dai segmenti $B'B$, AB , AA' e dall'arco di circonferenza $A'B'$.

Nel caso specifico di questo lavoro la superficie stradale non è rappresentata da una funzione del tipo $z(x, y)$ ma da una serie di triangoli. Questo comporta l'impossibilità di valutare la terza coordinata dei punti di campionamento Q_i^* . Per sopperire a questo problema si utilizzerà l'algoritmo per l'intersezione tra raggio e triangolo presentato nel Capitolo 4. Si definiranno dunque i punti di origine dei raggi direttamente nel sistema di riferimento della ruota RF_{wh} come:

$$\begin{aligned} Q_{1,2}^* &= M \pm r_{Q_{1,2}} = P^* \pm \Delta x e_{xC} \\ Q_{3,4}^* &= M \pm r_{Q_{3,4}} = P^* \pm \Delta y e_{yC} \end{aligned} \quad (3.14)$$

dai quali partiranno con direzione $-z_C$ e intersecheranno la *mesh* nei punti (Q_1, Q_2, Q_3, Q_4) .

3.3.1.2 Contatto ponderato in base all'area d'intersezione

Piano locale In alternativa, si può utilizzare un modello di contatto ponderato in base all'area di intersezione. In altre, parole si andrà a valutare triangolo per triangolo l'intersezione con il disco indeformabile. Prima di tutto si intersecherà il triangolo nello spazio con il piano in cui giace il disco trovando dunque un segmento. Successivamente si valuterà l'intersezione di questo segmento con il disco, calcolando l'area tra il segmento stesso e il semicerchio inferiore del disco.

Attraverso questa area si potrà pesare la normale alla faccia del triangolo considerato e quindi effettuare una media ponderata con tutti gli altri triangoli che

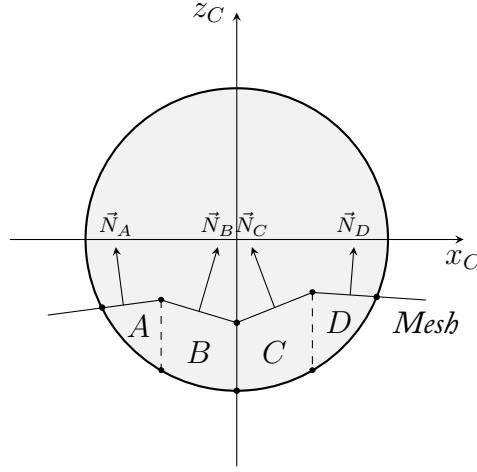


FIGURA 3.8: I versori normali \vec{N}_A , \vec{N}_B , \vec{N}_C e \vec{N}_D vengono ponderati in base all'area delle rispettive regioni d'intersezione A , B , C e D .

intersecano il disco, ovvero:

$$\mathbf{e}_n = \sum_{i=0}^{N_T} A_i \mathbf{e}_{n_i} \quad (3.15)$$

dove:

- N_T è il numero di triangoli all'interno della *Bounding Box* (BB) rappresentante l'ombra dello pneumatico;
- \mathbf{e}_n è il versore normale risultante;
- \mathbf{e}_{n_i} è il versore normale dell' i -esimo triangolo;
- A_i corrisponde all'area tra il segmento creato dall'intersezione piano-triangolo e il semicerchio inferiore del disco dell' i -esimo triangolo.

Questo metodo è ovviamente utilizzabile solo nel caso di strada rappresentata tramite *mesh* triangolare. A differenza dal modello di [9], permette di non approssimare la superficie stradale mediante soli quattro punti invece di avere una rappresentazione che sfrutta tutti i dati messi a disposizione dalla discretizzazione del manto stradale.

Punti di contatto Per trovare il punto di contatto con la *mesh* P_{PL} e il punto di contatto sulla circonferenza del disco indeformabile P_{MF} precedentemente definiti, si andrà a ripetere l'operazione in 3.3.1.1 per trovare la componente della normale al piano strada $\mathbf{e}_{n_{XZ}}$ sul piano in cui giace il singolo disco indeformabile. A differenza del modello di contatto di Rill, non si ha ora una definizione univoca del piano

strada locale. Infatti, l'unica componente ad ora conosciuta è il versore risultante normale al terreno e_n . Per ricavare il punto di contatto sulla circonferenza del disco indeformabile P_{MF} si utilizzerà la (3.12), mentre per il punto sulla *mesh* P_{PL} si andrà ad utilizzare un algoritmo d'intersezione tra raggio e la superficie stradale, dove l'origine del raggio sarà il centro ruota e la direzione $-e_{n_{XZ}}$.

3.3.2 Modello di pneumatico a più dischi

In questo modello lo pneumatico sarà rappresentato da più dischi rigidi indeformabili disposti uniformemente lungo la sezione dello stesso. Essi potranno avere raggio uguale o diverso l'uno dall'altro, in modo da rappresentare una forma specifica dello pneumatico.

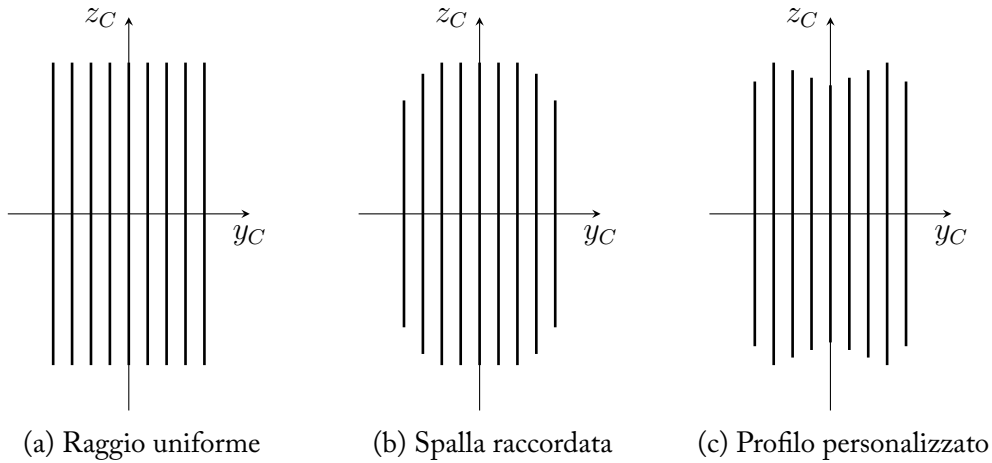


FIGURA 3.9: Disposizione dei dischi.

Anche se questo modello di pneumatico è costituito da più dischi, il punto di contatto P_{MF} utilizzato per valutare la formula di Pacejka verrà comunque considerato nel disco fittizio giacente sul piano XZ dello pneumatico. Equivalentemente, anche il punto di contatto con la *mesh* P_{PL} verrà considerato nel medesimo piano.

3.3.2.1 Contatto ponderato in base all'area d'intersezione

Piano locale Analogamente al modello di pneumatico a singolo disco, si può effettuare la stessa operazione su ogni disco per trovare il versore normale risultante

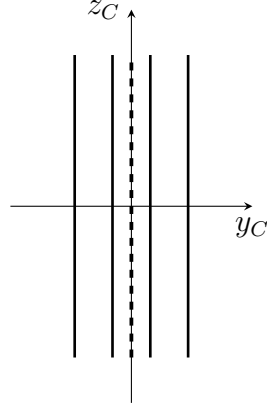


FIGURA 3.10: Pneumatico rappresentato da dischi a raggio uniforme. Notare il disco fittizio giacente sul piano XZ in linea tratteggiata.

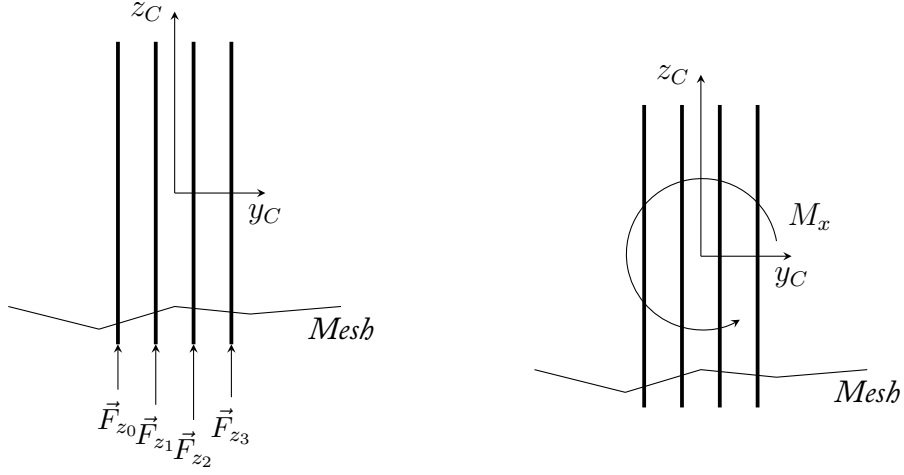


FIGURA 3.11: Momento creato dalla morfologia del terreno.

$e_{n_{D_j}}$ relativo al contatto del j -esimo disco. La (3.15) diventerà dunque:

$$e_{n_{D_j}} = \sum_{i=0}^{N_T} A_i e_{n_i} \quad (3.16)$$

Per combinare assieme i versori normali $e_{n_{D_j}}$ relativi ai dischi si effettuerà una nuova media ponderata pesata questa volta sull'area totale d'intersezione all'interno del disco. La formula sarà dunque:

$$e_n = \sum_{j=0}^{N_D} A_{D_j} e_{n_{D_j}} \quad (3.17)$$

dove:

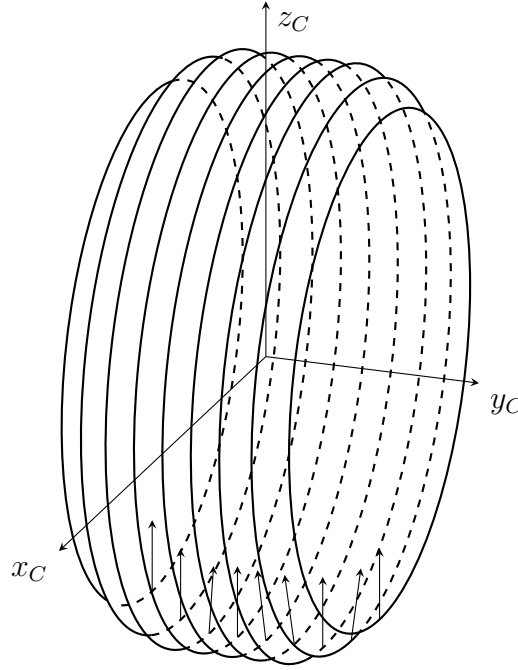


FIGURA 3.12: Normali associate ai vari dischi dello pneumatico.

- N_D è il numero di dischi totali rappresentanti lo pneumatico;
- e_n è il versore normale risultante;
- $e_{n_{D_j}}$ è il versore normale associato al j -esimo disco;
- A_{D_j} corrisponde all'area d'intersezione all'interno del j -disco e sotto la superficie della *mesh*.

Punti di contatto Avendo ora a disposizione il versore normale risultante e_n , si possono trovare i punti P_{MF} e P_{PL} adottando il medesimo metodo utilizzato in 3.3.1.1. Si noti che anche se lo pneumatico è rappresentato da più dischi, per ricondursi alle condizioni tali per cui il modello di Pacejka è valido, è necessario immaginare che ci sia sempre un disco fittizio giacente sul piano XZ della ruota.

3.3.2.2 Contatto tramite campionamento

Piano locale Nel caso in cui la densità della *mesh* sia troppo alta, effettuare il calcolo per il modello di contatto ponderato in base all'area d'intersezione precedentemente presentato, può essere molto dispendioso in termini di calcoli, e quindi di tempo. Per sopperire a questo problema, se il numero di triangoli è superiore ad una certa

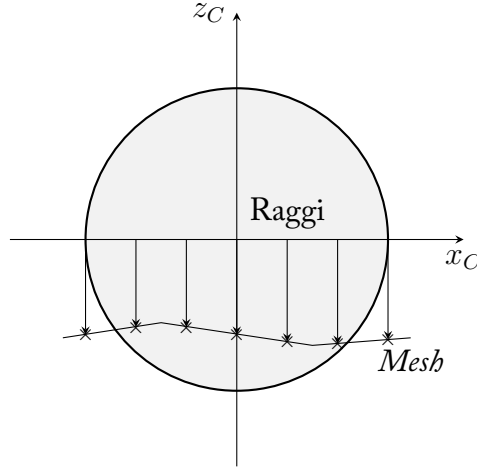


FIGURA 3.13: Campionamento della *mesh* triangolare in corrispondenza del piano in cui giace l' i -esimo disco. I raggi partono dall'asse x_C in direzione z_C .

misura, si andrà a campionare la *mesh* triangolare in corrispondenza del piano in cui giacciono i dischi. In particolare, per campionare la *mesh* si sfrutterà l'algoritmo di intersezione tra raggio e triangolo che verrà presentato nel Capitolo 4. Attraverso questo algoritmo, supponendo che il raggio abbia la stessa direzione dell'asse z_C , si andranno a memorizzare le normali alle facce dei triangoli campionati. Il versore normale risultante e_n non verrà più calcolato mediante una media ponderata ma bensì attraverso una semplice media aritmetica tra tutti i punti campionati lungo il disco.

Punti di contatto Avendo ora a disposizione il versore normale risultante e_n , si possono trovare i punti P_{MF} e P_{PL} adottando il medesimo metodo utilizzato in 3.3.1.1, sempre tenendo conto che entrambi i punti giaceranno sul disco fittizio corrispondente al piano XZ della ruota.

4.1 Struttura ad albero di tipo "*Bounding Volume Hierarchy*"

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 strutture BVH vengono utilizzate per supportare in modo efficiente diverse operazioni su insiemi di oggetti geometrici, come ad esempio il rilevamento delle collisioni.

Il confinamento degli oggetti nei volumi di delimitazione e l'esecuzione di test di collisione su di essi, prima di effettuare il test della geometria dell'oggetto stesso, comporta una semplificazione del problema e un miglioramento significativo delle prestazioni. Organizzando i volumi di delimitazione in una struttura BVH, la complessità temporale (il numero di test eseguiti) può essere ridotta logicamente nel numero di oggetti. Con una tale struttura, durante i test di collisione, i volumi secondari non devono essere esaminati se i loro volumi principali non sono intersecati.

4.1.1 Struttura di tipo "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 il rettangolo con la misura più piccola (area, volume o iper-volume 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.

4.1.1.1 Struttura di tipo "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 di oggetti, in primo esame si valuteranno le intersezioni tra i loro MBB. Questa operazione consente di escludere rapidamente i controlli delle coppie che sono molto distanti.

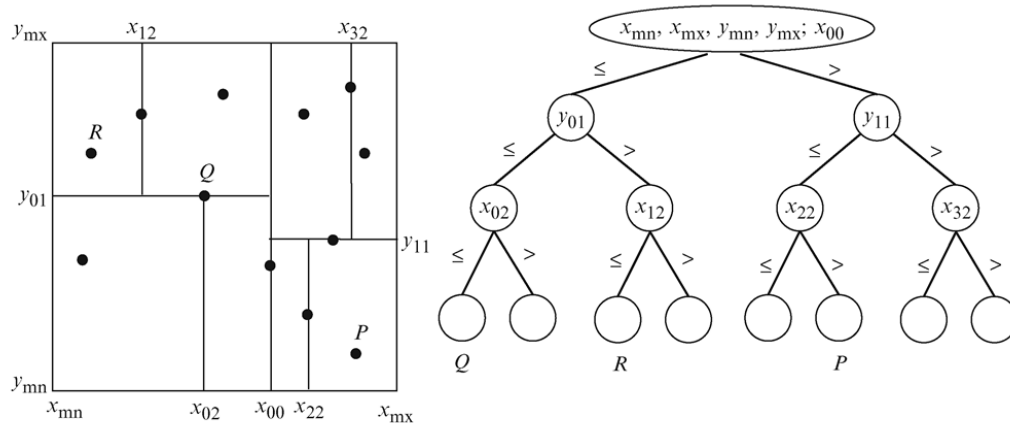


FIGURA 4.1: Esempio di albero di tipo AABB.

4.1.1.2 Struttura di tipo "Arbitrarily Oriented Bounding Box"

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

4.1.1.3 Struttura di tipo "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.

4.1.2 Intersezione tra alberi di tipo 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, vengono controllate le posizioni delle i -esime 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 rettangoli non ruotati può essere controllata solo tramite confronti logici. I riquadri ruotati richiedono invece 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, altrimenti 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 potrebbe portare i punti di campionamento del terreno fuori dall'ombra. La strada, contrariamente allo pneumatico, verrà tenuta come riferimento assoluto. In altre parole, una volta effettuata l'analisi sintattico-grammaticale del *file* RDF, verrà calcolato l'albero di tipo AABB. Lo pneumatico si muoverà all'interno della *mesh* e la sua ombra verrà ricalcolata e intersecata con l'albero AABB per ottenere tutti i triangoli in corrispondenza della stessa.

Volendo intersecare due semplici BB, quali

$$A = [A.minX, A.maxX, A.minY, A.maxY]$$

$$B = [B.minX, B.maxX, B.minY, B.maxY]$$

verrà usata la seguente funzione:

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

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

È importante notare che il metodo appena visto, presenta numerosi vantaggi:

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

4.2 Algoritmi geometrici

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. I progressi compiuti nei campi computer grafica, del *Computer-Aided Design* (CAD), del *Computer-Aided Manufacturing* (CAM) e nella visualizza-

zione matematica sono la ragione principale per lo sviluppo della geometria computazionale. Ad oggi, le sue applicazioni si trovano nella robotica, nella progettazione di circuiti integrati, nella visione artificiale, nel *Computer-Aided Engineering* (CAE) e nel *Geographic Information Systems* (GIS). I rami principali della geometria computazionale sono:

- *Calcolo combinatorio* (o *geometria algoritmica*), che si occupa di oggetti geometrici come entità discrete. Ad esempio, può essere utilizzato per determinare il poliedro o il poligono più piccolo che contiene tutti i punti forniti, o più formalmente, dato un insieme di punti, si deve determinare il più piccolo insieme convesso che li contenga tutti (problema dell'involuppo convesso).
- *Geometria di calcolo numerica* (o *Computer-Aided Geometric Design* (CAGD)), che si occupa principalmente di rappresentare oggetti del mondo reale in forme adatte per i calcoli informatici nei sistemi CAD e CAM. Questo ramo può essere considerato uno sviluppo della geometria descrittiva ed è spesso ritenuto 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 saranno trattati tutti gli algoritmi che verranno utilizzati in seguito durante l'analisi geometrica dell'intersezione tra pneumatico e superficie stradale. 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).

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

4.2.1 Intersezione tra entità geometriche

4.2.1.1 Intersezione punto-segmento

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

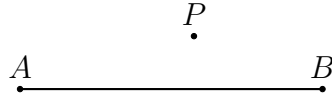


FIGURA 4.2: Schema del problema di intersezione punto-segmento

Per determinare se il punto P è interno al segmento si eseguiranno i seguenti step:

1. creazione di un vettore \overrightarrow{AB} e di un vettore \overrightarrow{AP} ;
2. calcolo del 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 del prodotto scalare tra \overrightarrow{AB} e \overrightarrow{AP} , se è nullo allora $P \equiv A$, se è pari al modulo di \overrightarrow{AB} allora il $P \equiv B$, se è compreso tra 0 il modulo di \overrightarrow{AB} , allora il punto P giace all'interno del segmento considerato.

Lo pseudocodice che esegue questo tipo di test è riportato in Figura 4.4

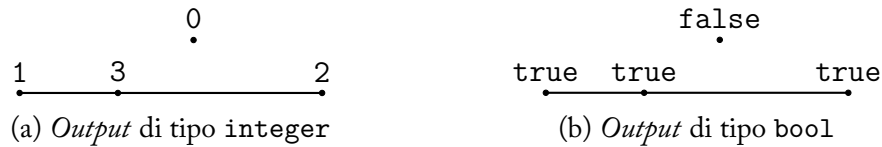


FIGURA 4.3: Schemi per l'output dell'intersezione punto-segmento.

4.2.1.2 Intersezione punto-cerchio

Data una circonferenza con centro $C = (x_c, y_c)$ e raggio r , il problema consiste nel trovare se un generico punto $P = (x_p, y_p)$ è all'interno, all'esterno o corrispondente alla circonferenza.

La soluzione al problema è semplice: la distanza tra il centro della circonferenza C e il punto P è data dal teorema di Pitagora, ovvero:

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

<i>Output di tipo integer</i>	<i>Output di tipo bool</i>
<pre> 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; </pre>	<pre> 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; </pre>

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

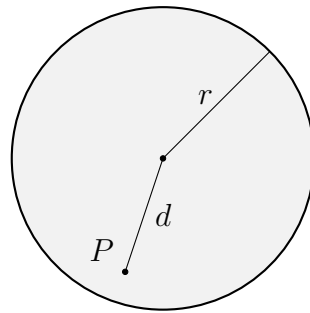


FIGURA 4.5: Schema del problema di intersezione punto-cerchio.

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

Gli *inputs* dell'algoritmo per l'intersezione punto-cerchio sono:

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

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

- 0 se il punto è esterno alla circonferenza;
- 1 se il punto è interno alla circonferenza;
- 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 alla circonferenza;
- true se il punto è interno o appartiene alla circonferenza.

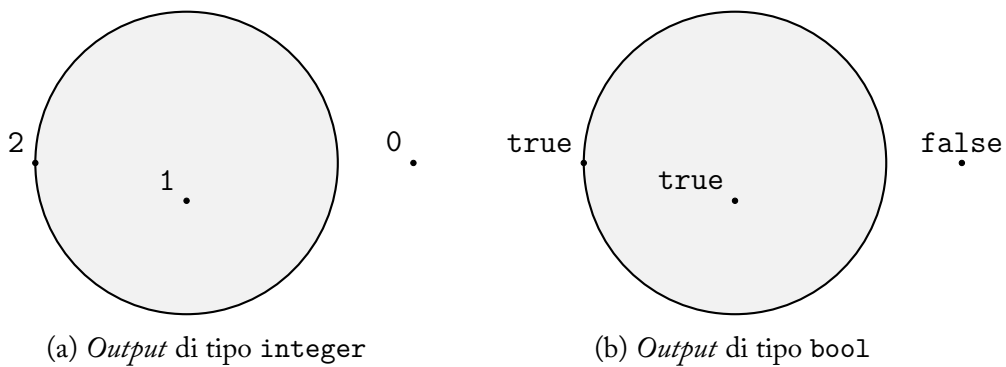


FIGURA 4.6: Schemi per l'*output* dell'intersezione punto-cerchio.

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

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

4.2.1.3 Intersezione segmento-circonferenza

Per l'intersezione di un segmento, avente punto iniziale e finale rispettivamente in P_0 e P_1 , con una circonferenza, avente centro $C = (x_c, y_c)$ e raggio r , è necessario prima di tutto riscrivere le equazioni di entrambe le entità. Il segmento viene momentaneamente riscritto come una retta del tipo:

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

mentre la circonferenza come:

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

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

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

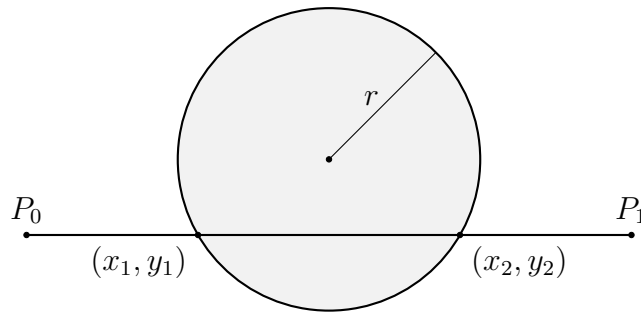


FIGURA 4.8: Schema del problema di intersezione punto-circonferenza.

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

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

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

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

I termini a , b e c saranno quindi pari a:

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

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

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

oppure:

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

Se $r^2(a^2 + b^2) - c^2 \geq 0$ vale come una disuguaglianza stretta, esistono due punti di intersezione. 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 risultare:

- 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.

Una volta ottenute il numero di intersezioni e le soluzioni (x_1, y_1) e (x_2, y_2) bisognerà controllare che queste siano all'interno del segmento individuato inizialmente dai punti P_0 e P_1 . Per questo motivo si utilizzerà l'algoritmo di intersezione punto-segmento visto precedentemente in 4.2.1.1.

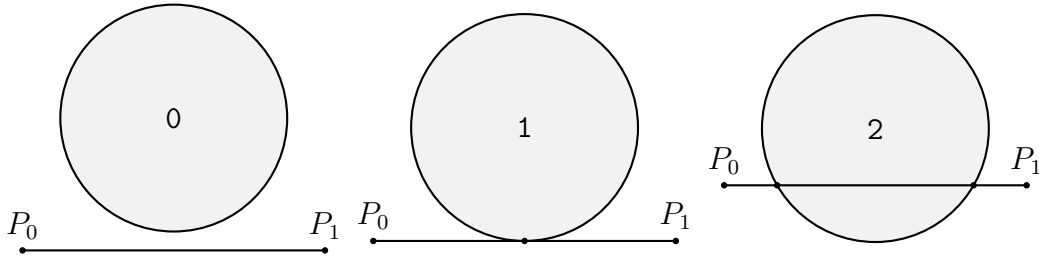


FIGURA 4.9: Schemi per l'*output* dell'intersezione segmento-cerchio.

4.2.1.4 Intersezione piano-piano

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

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

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

```

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 dello pseudocodice 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 . 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 di un terzo piano P_3 avente equazione implicita $\vec{n}_3 \cdot P = 0$ dove $\vec{n}_3 = \vec{n}_1 \times \vec{n}_2$ e $d_3 = 0$ (ovvero passa attraverso l'origine). Questo metodo è funzionante poiché:

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

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

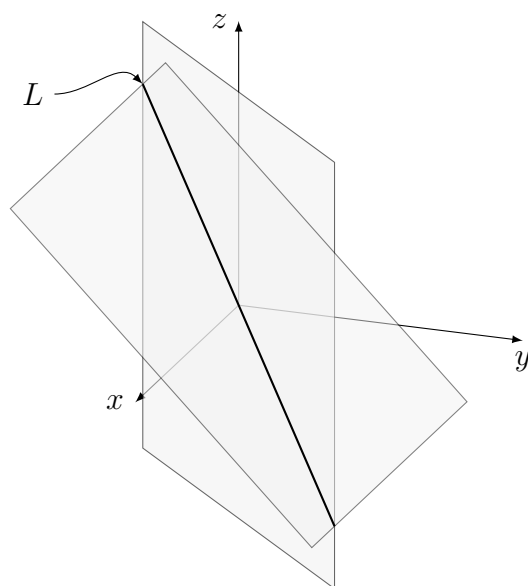


FIGURA 4.11: Schemi del problema di intersezione piano-piano.

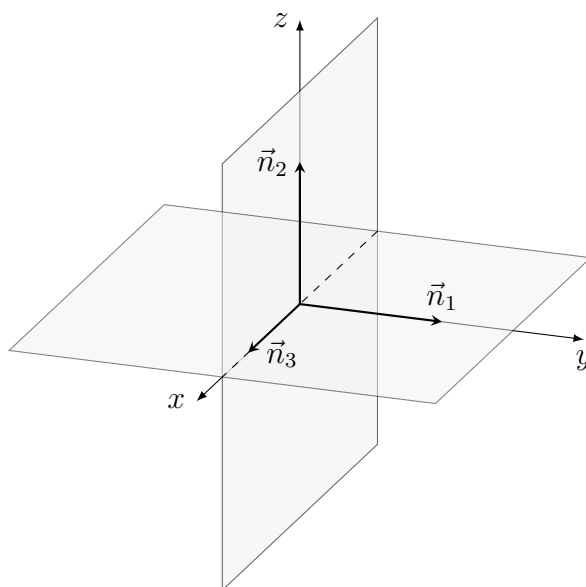


FIGURA 4.12: Vettori dei piani P_1 , P_2 e della retta L .

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

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

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

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

pertanto l'equazione parametrica per la retta L sarà:

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

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

```

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 dello pseudocodice per l'intersezione piano-piano.

4.2.1.5 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:

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

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

si intersecano, oppure L giace totalmente nel piano P . Disgiunzione o coincidenza possono essere determinate testando se in P esiste un punto specifico di L , per esempio P_0 , ovvero se soddisfa l'equazione di linea implicita:

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

Se la linea e il piano non sono paralleli, allora L e P si intersecano in un unico punto $P(t_I)$. 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:

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

Risolviendo si ottiene:

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

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 dimostrare che vi sia un'intersezione tra il segmento e il piano. Per raggio, c'è invece un'intersezione con il piano quando $t_I \geq 0$.

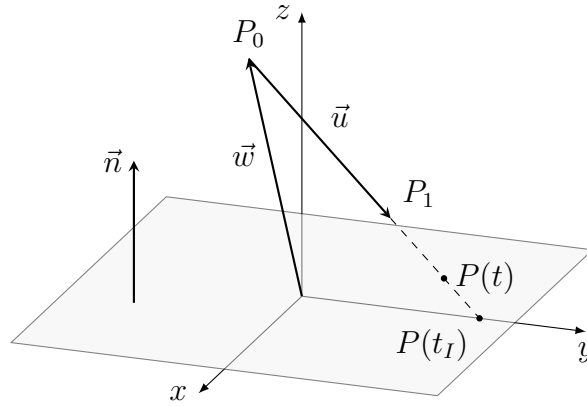


FIGURA 4.14: Vettori dei piani P_1 , P_2 e della retta L .

4.2.1.6 Intersezione piano-triangolo

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

```

1  u = P_1 - P_0;
2  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 dello pseudocodice 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 dello pseudocodice per l'intersezione piano-triangolo.

4.2.1.7 Intersezione 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:

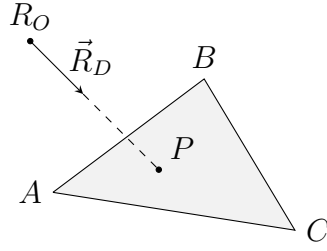


FIGURA 4.17: Schema del problema di intersezione raggio-triangolo.

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

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

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

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

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

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

e sviluppando si ottiene:

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

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

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

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

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

Sul membro a sinistra si hanno le tre incognite (t, u, v) moltiplicate per tre termini noti $(B - A, C - A, D)$. Si può riorganizzare questi termini e presentare l'equazione 4.22 usando la seguente notazione:

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

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

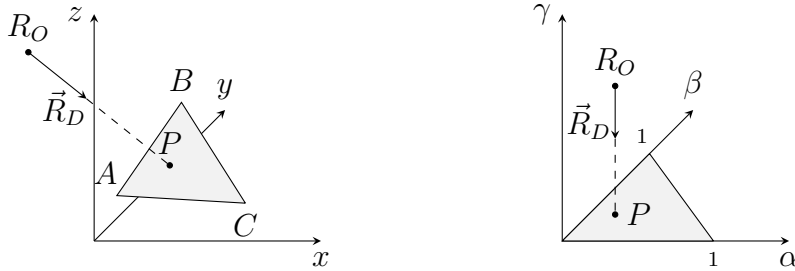


FIGURA 4.18: Cambiamento di coordinate nell'algoritmo di Möller-Trumbore.

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 sa inoltre che t esprime la distanza dall'origine del raggio a P , il punto di intersezione. Si è quindi creato un sistema di coordinate che consentirà di esprimere univocamente la posizione del punto d'intersezione P in termini di coordinate baricentriche e distanza dall'origine del raggio a quel punto sul triangolo.

Möller e Trumbore spiegano che la prima parte dell'equazione 4.23 (il termine $O - A$) può essere vista come una trasformazione che sposta il triangolo dalla sua posizione spaziale mondiale originale all'origine (il primo vertice del triangolo coincide con l'origine). L'altro lato dell'equazione ha l'effetto di trasformare il punto di intersezione dallo spazio (x, y, z) nello spazio (t, u, v) come spiegato precedentemente.

Per risolvere l'equazione 4.23, Möller e Trumbore hanno usato una tecnica conosciuta in matematica come regola di Cramer. La regola di Cramer fornisce la soluzione a un sistema di equazioni lineari mediante il determinante. La regola afferma che se la moltiplicazione di una matrice M per un vettore colonna X è uguale a un vettore colonna C , allora è possibile trovare X_i (l' i -esimo elemento del vettore colonna X) dividendo il determinante di M_i per il determinante di M . Dove M_i è la matrice formata sostituendo la sua colonna di M con il vettore colonna C . Usando questa regola si ottiene;

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

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

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

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

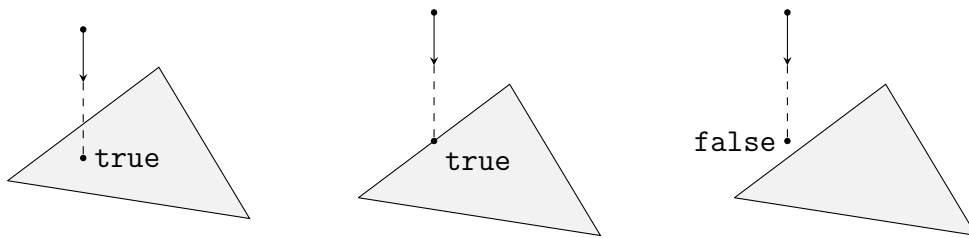


FIGURA 4.19: Schemi per l'output dell'intersezione punto-cerchio.

```

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_0 - A;
7    u = A · T;
8    if ( u < 0.0 || u > D ) return false;
9    B = T × E_1;
10   v = B · R_D;
11   if ( v < 0.0 || u + v > D ) return false;
12 } else if ( D < -epsilon ) {
13   T = R_0 - A;
14   u = A · T;
15   if ( u > 0.0 || u < D ) return false;
16   B = T × E_1;
17   v = B · R_D;
18   if ( v > 0.0 || u + v < D ) return false;
19 } else {
20   return false;
21 }
22 t = ( B · E_2 ) / D;
23 if ( t > 0.0 ) {
24   P = Q + D * t;
25   return true;
26 } else {
27   return false;
28 }

```

FIGURA 4.20: Schema dello pseudocodice per l'intersezione raggio-triangolo con *back-face culling*.

5.1 Organizzazione

La libreria TireGround è stata organizzata in due parti, la prima gestisce la superficie stradale mentre la seconda gestisce i modelli di pneumatico. Si sviluppa all'interno dell'omonimo *namespace* TireGround nel quale vengono inoltre dichiarati con `typedef` alcuni tipi che verranno utilizzati nelle due sottosezioni. Verranno ora riportate le informazioni di maggior rilievo per ognuna delle due parti della libreria.

5.1.1 Gestione della superficie stradale

La gestione della superficie stradale avviene all'interno del *namespace* RDF. In quest'ultimo 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 e processare la *mesh* a partire dal *file* in formato RDF.

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

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

- `updateBBBox2D` – aggiorna il dominio della BB settando i suoi valori secondo il massimo ingombro dato dai tre vertici nello spazio tridimensionale in *input*.

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

TABELLA 5.1: Attributi della classe BBox2D.

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

- `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 origine, 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	TriangleBBBox	•	•	BB del triangolo

TABELLA 5.2: Attributi della classe Triangle3D.

TriangleRoad Questa classe contiene tutte le informazioni geometriche e non geometriche per definire e manipolare un triangolo con vertici nello spazio tridimensionale rappresentante la superficie stradale. È derivato dalla classe `Triangle3D` e ha inoltre un attributo che permette 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 l'analisi sintattico-grammaticale del *file* RDF. Inoltre, contiene il vettore di puntatori alle BB di tipo `PtrBBBox`, necessario per calcolare l'albero di tipo `AABB`. Quest'ultimo esiste come ulteriore attributo della classe sotto forma di puntatore `PtrAABB`. I metodi più importanti di questa classe sono:

- `set` — copia la *mesh* da un'altra già esistente.
- `LoadFile` — effettua l'analisi sintattico-grammaticale del *file* dato come *input* e crea le istanze `TriangleRoad` che costituiscono la *mesh*.
- `updateIntersection` — interseca l'albero di tipo `AABB` della *mesh* con un altro albero esterno di tipo `AABB` e ne restituisce il vettore dei puntatori di tipo `std::shared_ptr` alle istanze della classe `TriangleRoad` che vengono intersecate.

Tipo	Nome	Getter	Setter	Descrizione
<code>TriangleRoad_list</code>	Friction	•		Vettore dei triangoli
<code>std::vector<PtrBBBox></code>	<code>PtrBBBoxVec</code>	•		Vettore delle BB
<code>PtrAABB</code>	<code>PtrTree</code>	•		Albero di tipo <code>AABB</code>

TABELLA 5.4: Attributi della classe MeshSurface.

5.1.2 Gestione dei modelli di pneumatico

La gestione dei modelli di pneumatico avviene nel *namespace* `TireGround`. Quest'ultimo contiene tutti le classi e le funzioni per gestire l'intersezione tra lo pneumatico e la *mesh* a partire dalla conoscenza di quest'ultima, della geometria e della posizione dello pneumatico.

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

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

- `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.
- `contactTriangles` – funzione in *overloading* che consente di ottenere il versore normale e coefficiente attrito medi ponderati sull'area, nonché l'area di contatto stessa all'interno del singolo disco a partire da una serie di triangoli.
- `contactPlane` – funzione in *overloading* che consente di ottenere l'area di contatto all'interno del singolo disco dato un piano.

Tipo	Nome	Getter	Setter	Descrizione
vec2	OriginXZ	•	•	Coordinate XZ del disco
real_type	OffsetY	•	•	Coordinata Y del disco
real_type	Radius	•	•	Raggio 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 3 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 posizio-

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 ETRT0.

namiento dello spazio del sistema di riferimento. I metodi più importanti di questa classe sono:

- `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 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 gli alberi di tipo AABB relativi all'ombra totale, della parte superiore e della parte inferiore del BB tridimensionale che racchiude lo pneumatico. I metodi più importanti di questa classe sono:

- `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
PtrAABB	PtrTree	•		Albero AABB totale
PtrAABB	PtrTree_U	•		Albero AABB parte superiore
PtrAABB	PtrTree_L	•		Albero AABB parte inferiore

TABELLA 5.8: Attributi della classe Shadow.

SamplingGrid Questa classe contiene tutti i parametri che riguardano la precisione dei calcoli che verranno effettuati nel calcolo della normale al terreno, punto e area di contatto.

Tipo	Nome	Getter	Setter	Descrizione
int_type	PointsN	•	•	N° di punti di campionamento
int_type	DisksN	•	•	N° di dischi
int_type	Switch	•	•	<i>Threshold</i> per il tipo contatto

TABELLA 5.9: Attributi della classe SamplingGrid.

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. È una classe virtuale in quanto viene definita con alcuni metodi puri virtuali. Questi metodi verranno poi sostituiti con nelle classi figlie.

Tipo	Nome	Getter	Setter	Descrizione
SamplingGrid	Precision			Precisione dei calcoli
ETRTO	TireGeometry			Geometria
ReferenceFrame	RF	•	•	Posizione
int_type	TirePose	•	•	<i>Flag</i> per la posizione

TABELLA 5.10: Attributi della classe Tire.

MagicFormula e MultiDisk Queste classi calcolano tutti i parametri necessari per valutare il contatto tra pneumatico a disco singolo e terreno attraverso la formula di Pacejka. I metodi più importanti di queste classi sono:

- `setup` — consente di riposizionare la ruota all'interno della *mesh*.
- `calculateRelativeCamber` — calcola il camber relativo.
- `getRho` — calcola l'affondamento del disco nel piano strada locale.
- `gettArea` — calcola l'area d'intersezione dei dischi.

Tipo	Nome	Getter	Descrizione
Disk	SingleDisk		Disco rigido
vec3	Normal	•	Vettore del piano strada
vec3	MeshPoint	•	Punto di contatto sulla <i>mesh</i>
vec3	MeshPoint	•	Punto di contatto sul disco
real_type	Friction	•	Coefficiente di attrito locale
real_type	Area	•	Area d'intersezione

TABELLA 5.11: Attributi della classe MagicFormula.

Tipo	Nome	Getter	Descrizione
Disk	DiskVec		Vettore dei dischi
vec3	NormalVec	•	Vettore dei vettori normali
vec3	MeshPointVec	•	Vettore dei punti di contatto sulla <i>mesh</i>
vec3	DiskPointVec	•	Vettore dei punti di contatto sul disco
real_type	FrictionVec	•	Vettore dei coefficienti di attrito locale
real_type	AreaVec	•	Vettore delle aree d'intersezione
vec3	Normal	•	Vettore del piano strada
vec3	MeshPoint	•	Punto di contatto singolo sulla <i>mesh</i>
vec3	MeshPoint	•	Punto di contatto singolo sul disco
real_type	Friction	•	Coefficiente di attrito locale
real_type	Area	•	Area totale d'intersezione

TABELLA 5.12: Attributi della classe MultiDisk.

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

5.3 Utilizzo

La libreria TireGround è stata pensata per essere semplice da utilizzare. Si vedranno ora i vari passi per utilizzarla in maniera appropriata.

Caricare la *mesh* Per caricare la superficie stradale, rappresentata dalla *mesh* triangolare contenuta nel *file* RDF, è sufficiente sfruttare il costruttore della classe `MeshSurface` che prende in *input* l'indirizzo al *file*.

```
1  RDF::MeshSurface Road("./file.rdf");
```

Creare lo pneumatico Per creare lo pneumatico a singolo disco è sufficiente utilizzare il costruttore di *default* della classe `MagicFormula`.

```

1  TireGround::Tire* SampleTire = new TireGround::MagicFormula(
2      SectionWidth, // Sezione laterale dello pneumatico [mm]
3      AspectRatio,  // Aspect ratio percentuale dello pneumatico
4      RimDiameter,  // Diametro del cerchio [in]
5      Threshold     // Threshold per passare dal modello di contatto
                      ponderato in base all'area di intersezione a quello di Rill
6  );

```

Nel caso invece si voglia creare uno pneumatico a più dischi si utilizzerà uno dei costruttori della classe MultiDisk.

Per il caso di pneumatico a più dischi con raggio uniforme si avrà:

```

1  TireGround::Tire* SampleTire = new TireGround::MultiDisk(
2      SectionWidth, // Sezione laterale dello pneumatico [mm]
3      AspectRatio,  // Aspect ratio percentuale dello pneumatico
4      RimDiameter,  // Diametro del cerchio [in]
5      PointsNumber, // Numero di punti di campionamento per ogni disco
6      DisksNumber,  // Numero di dischi totale
7      Threshold     // Threshold per passare dal modello di contatto
                      ponderato in base all'area di intersezione a quello di Rill
8  );

```

Nel caso di pneumatico a più dischi con raggio di raccordo sulla spalla si avrà invece:

```

1  TireGround::Tire* SampleTire = new TireGround::MultiDisk(
2      SectionWidth, // Sezione laterale dello pneumatico [mm]
3      AspectRatio,  // Aspect ratio percentuale dello pneumatico
4      RimDiameter,  // Diametro del cerchio [in]
5      SideRadius,   // Raggio di raccordo sulla spalla [mm]
6      PointsNumber, // Numero di punti di campionamento per ogni disco
7      DisksNumber,  // Numero di dischi totale
8      Threshold     // Threshold per passare dal modello di contatto
                      ponderato in base all'area di intersezione a quello di Rill
9  );

```

Infine, nel caso si voglia creare uno pneumatico a più dischi con forma personalizzata:

```

1  TireGround::Tire* SampleTire = new TireGround::MultiDisk(

```

```

2   SectionWidth, // Sezione laterale dello pneumatico [mm]
3   AspectRatio,  // Aspect ratio percentuale dello pneumatico
4   RimDiameter,  // Diametro del cerchio [in]
5   RadiusVec,    // Vettore dei raggi dei dischi [m]
6   PointsNumber, // Numero di punti di campionamento per ogni disco
7   Threshold     // Threshold per passare dal modello di contatto
                  ponderato in base all'area di intersezione a quello di Rill
8   );

```

Orientazione dello pneumatico e valutazione del contatto Per orientare lo pneumatico e valutarne il contatto con il manto stradale si utilizzerà il metodo setup della classe Tire.

```

1 bool Out = TireMD->setup(
2   Road,      // Superficie stradale
3   TransfMat // Matrice di trasformazione 4x4 per orientare lo pneumatico
4   );

```

Per estrarre i risultati si andranno dapprima a inizializzazione delle variabili reali o vettoriali come segue.

```

1 // Inizializzazione delle variabili
2 TireGround::vec3 N;
3 TireGround::vec3 P;
4 TireGround::real_type Friction;
5 TireGround::real_type Rho;
6 TireGround::real_type RhoDot;
7 TireGround::real_type RelativeCamber;
8 TireGround::real_type Area;
9 TireGround::real_type Volume;
10
11 // Estrazione della dimensione appropriata della struttura dati
12 TireGround::int_type size = TireSD->getDisksNumber();
13
14 // Inizializzazione dei vettori con dimensione appropriata
15 TireGround::row_vec3 NVec(size);
16 TireGround::row_vec3 PVec(size);
17 TireGround::row_vecN FrictionVec(size);

```

```

18 TireGround::row_vecN RhoVec(size);
19 TireGround::row_vecN RhoDotVec(size);
20 TireGround::row_vecN RelativeCamberVec(size);
21 TireGround::row_vecN AreaVec(size);
22 TireGround::row_vecN VolumeVec(size);

```

Successivamente verranno modificate dai metodi della classe Tire come:

```

1 // Estrazione dei dati
2 SampleTire->getNormal(N);
3 SampleTire->getMFpoint(P);
4 SampleTire->getFriction(Friction);
5 SampleTire->getRho(Rho);
6 SampleTire->getRhoDot(PreviousRho,TimeStep,RhoDot);
7 SampleTire->getRelativeCamber(RelativeCamber);
8 SampleTire->getArea(Area);
9 SampleTire->getVolume(Volume);
10
11 // Estrazione dei dati in vettori
12 SampleTire->getNormal(NVec);
13 SampleTire->getMFpoint(PVec);
14 SampleTire->getFriction(FrictionVec);
15 SampleTire->getRho(RhoVec);
16 SampleTire->getRhoDot(PreviousRho,TimeStep,RhoDotVec);
17 SampleTire->getRelativeCamber(RelativeCamberVec);
18 SampleTire->getArea(AreaVec);
19 SampleTire->getVolume(VolumeVec);

```

Casi particolari Nel caso in cui la variabile booleana in *output* dal metodo setup precedentemente chiamato sia falsa, si prospettano due casi.

Pneumatico fuori *mesh* Se, oltre alla condizione sulla variabile booleana in *output*, la lista di triangoli intersecati dall'ombra dello pneumatico è vuota. Per effettuare questo test si può intersecare l'albero della *mesh* con l'albero a una foglia dell'ombra dello pneumatico.

```

1 RDF::TriangleRoad_list TrianglesList;

```

```
2  bool List = Road.intersectAABBtree(TireSD->getAABBtree(),  
    TrianglesList);
```

Se la variabile booleana in *output* dal metodo `intersectAABBtree` è falsa allora la lista è vuota e lo pneumatico sarà quindi considerato fuori dalla superficie stradale descritta nel *file* RDF.

Pneumatico in volo Se, oltre alla condizione sulla variabile booleana in *output*, la lista di triangoli intersecati dall'ombra dello pneumatico non è vuota. Per effettuare questo test si può intersecare l'albero della *mesh* con l'albero a una foglia dell'ombra dello pneumatico. I parametri d'intersezione vanno settati dall'esterno della libreria TireGround a seconda della morfologia del terreno fuori dalla *mesh*.

```
1  RDF::TriangleRoad_list TrianglesList;  
2  bool List = Road.intersectAABBtree(TireSD->getAABBtree(),  
    TrianglesList);
```

Se la variabile booleana in *output* dal metodo `intersectAABBtree` è vera allora la lista non è vuota e lo pneumatico sarà quindi considerato in volo sopra la superficie stradale descritta nel *file* RDF. In questo caso i parametri d'intersezione vanno settati come intersezione nulla.

5.4 Prestazioni

I vari modelli di contatto sono stati preventivamente testati in una macchina avente le seguenti specifiche tecniche:

- memoria RAM:
 - dimensione: 8 GB;
 - frequenza: 1333 MHz;
- processore:
 - denominazione: Intel i5 3230M
 - numero di *core*: 2;
 - numero di thread: 4;
 - frequenza base: 2.60 GHz;
 - frequenza massima: 3.20 GHz;

	Modello di contatto		
	Rill	Ponderato sull'area	Mix
T_{totale} [ms]	116.543	121.249	106.058
T_{step} [ms]	0.0041621	0.00433017	0.00378765
σ [ms ²]	5.21719e-05	3.09234e-06	1.11754e-05
δ [ms]	0.00722301	0.00175851	0.00334297

Pneumatico 205/55R16
 Campionamenti = 28000
 Numero medio di triangoli sotto lo pneumatico = 11.4
Switch Rill ▷ Area a 10 triangoli

TABELLA 5.13: Tempi per il modello di pneumatico a singolo disco Tire::MagicFormula nel caso di *mesh* densa.

	Modello di contatto		
	Rill	Ponderato sull'area	Mix
T_{totale} [ms]	64.15	46.44	54.269
T_{step} [ms]	0.00229099	0.00165851	0.00193811
σ [ms ²]	3.92268e-06	4.39344e-06	5.825e-06
δ [ms]	0.00198058	0.00209605	0.0024135

Pneumatico 205/55R16
 Campionamenti = 28000
 Numero medio di triangoli sotto lo pneumatico = 3.2
Switch Rill ▷ Area a 3 triangoli

TABELLA 5.14: Tempi per il modello di pneumatico a singolo disco Tire::MagicFormula nel caso di *mesh* poco densa.

– *cache*: 3 MB.

I tempi rilevati per il modello di pneumatico a singolo disco Tire::MagicFormula sono riportati nelle Tabelle 5.13 e 5.14. Analogamente, i tempi rilevati per il modello di pneumatico a più dischi Tire::MultiDisk sono riportati nelle Tabelle ...

Testando il modello al simulatore è stato possibile valutare la sua complessità computazionale. Le specifiche tecniche di questo simulatore sono:

- memoria RAM: 32 GB;
- processore: Intel Xeon(R) 3.40 GHz (16 *cores*);
- scheda grafica: Nvidia GeForce GTX 680.

	Precisione		Modello di contatto		
	Dischi	Punti	Campionamento	Ponderato sull'area	Mix
T_{totale} [ms]	10	10	11369.5	871.896	8869.17
T_{step} [ms]	10	10	0.40604	0.031138	0.316745
σ [ms ²]	10	10	0.0104055	0.000660589	0.0426783
δ [ms]	10	10	0.102007	0.0257019	0.206587
T_{totale} [ms]	10	20	43516.8	1576.57	32886.4
T_{step} [ms]	10	20	1.55412	0.0563041	1.17447
σ [ms ²]	10	20	0.142766	0.000214928	0.619357
δ [ms]	10	20	0.377844	0.0146604	0.786992
T_{totale} [ms]	20	10	23533	799.28	18245.3
T_{step} [ms]	20	10	0.840434	0.0285447	0.651595
σ [ms ²]	20	10	0.0398545	5.12437e-05	0.193016
δ [ms]	20	10	0.199636	0.00715847	0.439336
T_{totale} [ms]	20	20	96955.9	1644.38	76486.6
T_{step} [ms]	20	20	3.46259	0.0587258	2.73157
σ [ms ²]	20	20	0.692889	0.000257929	3.51893
δ [ms]	20	20	0.8324	0.0160602	1.87588

Pneumatico 205/55R16

Campionamenti = 28000

Numero medio di triangoli sotto lo pneumatico = 11.4

Switch Campionamento \triangleright Area a 10 triangoli

TABELLA 5.15: Tempi per il modello di pneumatico a più dischi `Tire::MultiDisk` nel caso di *mesh* densa.

	Precisione		Modello di contatto		
	Dischi	Punti	Campionamento	Ponderato sull'area	Mix
T_{totale} [ms]	10	10			
T_{step} [ms]	10	10			
σ [ms ²]	10	10			
δ [ms]	10	10			
T_{totale} [ms]	10	20			
T_{step} [ms]	10	20			
σ [ms ²]	10	20			
δ [ms]	10	20			
T_{totale} [ms]	20	10			
T_{step} [ms]	20	10			
σ [ms ²]	20	10			
δ [ms]	20	10			
T_{totale} [ms]	20	20			
T_{step} [ms]	20	20			
σ [ms ²]	20	20			
δ [ms]	20	20			

Pneumatico 205/55R16

Campionamenti = 28000

Numero medio di triangoli sotto lo pneumatico = 3.2

Switch Campionamento \triangleright Area a 3 triangoli

TABELLA 5.16: Tempi per il modello di pneumatico a più dischi `Tire::MultiDisk` nel caso di *mesh* poco densa.

Il modello sviluppato soddisfa il vincolo del tempo reale imposto per le simulazioni in tempo reale. Il tempo di calcolo sull'*hardware* del simulatore professionale si traduce in un ciclo di lavoro del 77%, garantendo quindi un ampio margine di sicurezza e dando la possibilità di aumentare la precisione del modello attraverso l'implementazione di modelli più complessi.

Dato che la rappresentazione dello pneumatico è basata sul modello di Pacejka, ovvero un modello semi-empirico, non tiene conto dei fenomeni transitori. Sarà quindi una scelta obbligata passare ad una rappresentazione dello pneumatico mediante un modello fisico. Quest'ultimo, infatti, a seconda del grado di complessità, può tenere in considerazione alcuni dei fenomeni transitori che maggiormente influenzano la manovrabilità del veicolo. I modelli fisici sono quindi anche molto complessi. Lo pneumatico è modellato da diversi anelli circolari con punti di massa accoppiati anche in direzione laterale. Si tiene quindi conto del contatto in più punti e della distribuzione della pressione su tutta la larghezza della cintura. Importante sarà quindi valutare la possibilità di poter parallelizzare i processi di calcolo per diminuire i tempi di esecuzione.

Un punto critico sul quale è opportuno porre particolare attenzione è la rappresentazione del terreno. I *file* RDF hanno una struttura dati molto poco formale e solida. La mancanza di uno standard universalmente riconosciuto per questo formato rende impossibile implementare un *parser* sufficientemente efficiente e stabile per tutte le occasioni.

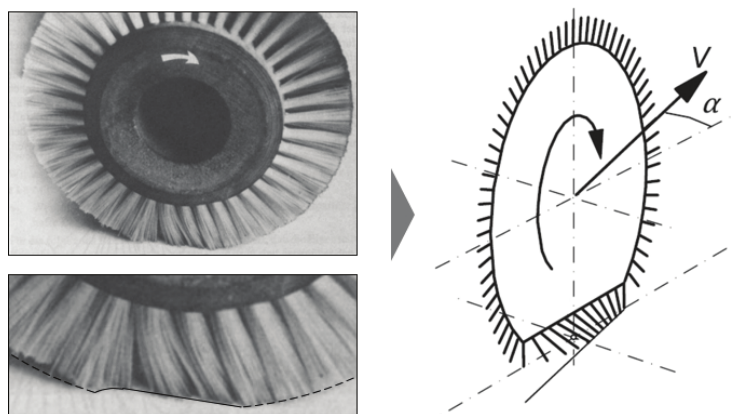


FIGURA 6.1: Schema strutturale del modello "a spazzola" (*brush model*).

Infine, considerando le finalità principali di questa tesi e i test effettuati sul modello sviluppato, possiamo affermare la sua validità per i campi di utilizzo previsti.

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*.

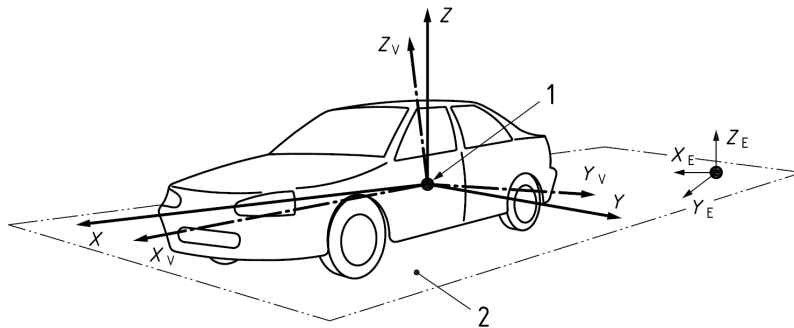


FIGURA A.1: Rappresentazione degli assi del sistema di riferimento della vettura secondo la convenzione ISO-V.

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].

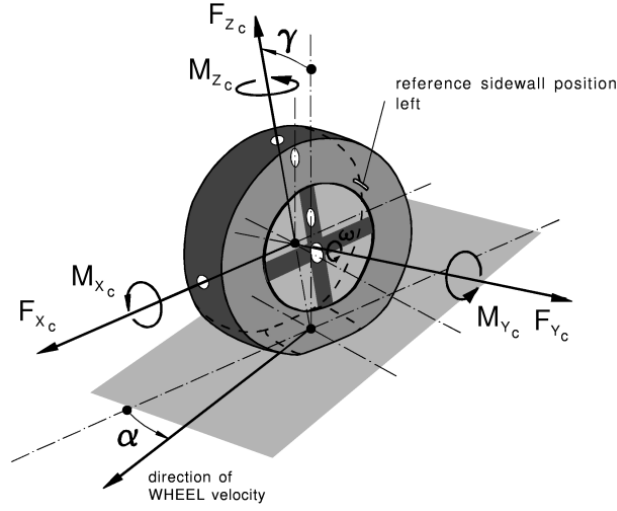


FIGURA A.2: Rappresentazione degli assi del sistema di riferimento dello pneumatico secondo la convenzione ISO-C.

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:

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

dove R_m è la matrice di rotazione 3×3 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.

Documentazione della libreria TireGround B

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 L^AT_EX) da una serie di *file* sorgente opportunamente 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 efficacemente 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.

TireGround

Generated by Doxygen 1.8.13

Contents

1	TireGround	1
2	Namespace Index	5
2.1	Namespace List	5
3	Hierarchical Index	7
3.1	Class Hierarchy	7
4	Class Index	9
4.1	Class List	9
5	Namespace Documentation	11
5.1	PatchTire Namespace Reference	11
5.1.1	Detailed Description	11
5.2	PatchTire::algorithms Namespace Reference	11
5.2.1	Detailed Description	12
5.2.2	Function Documentation	12
5.2.2.1	intersectPointSegment()	12
5.2.2.2	intersectRayPlane()	12
5.2.2.3	minmax_XY() [1/2]	13
5.2.2.4	minmax_XY() [2/2]	13
5.2.2.5	trapezoidArea()	13
5.2.2.6	weightedMean() [1/2]	14
5.2.2.7	weightedMean() [2/2]	14
5.3	RDF Namespace Reference	14
5.3.1	Detailed Description	15
5.4	RDF::algorithms Namespace Reference	15
5.4.1	Detailed Description	15
5.4.2	Function Documentation	15
5.4.2.1	firstToken()	15
5.4.2.2	getElement()	16
5.4.2.3	split()	16
5.4.2.4	tail()	16
5.5	TireGround Namespace Reference	16
5.5.1	Detailed Description	17

6	Class Documentation	19
6.1	RDF::BBox2D Class Reference	19
6.1.1	Detailed Description	19
6.1.2	Constructor & Destructor Documentation	20
6.1.2.1	BBox2D()	20
6.1.3	Member Function Documentation	20
6.1.3.1	print()	20
6.1.3.2	updateBBox2D()	20
6.2	PatchTire::Disk Class Reference	20
6.2.1	Detailed Description	21
6.2.2	Constructor & Destructor Documentation	21
6.2.2.1	Disk()	21
6.2.3	Member Function Documentation	21
6.2.3.1	contactPlane()	22
6.2.3.2	contactTriangles()	22
6.2.3.3	set()	22
6.2.3.4	setOriginXZ()	23
6.3	PatchTire::ETRTO Class Reference	23
6.3.1	Detailed Description	23
6.3.2	Constructor & Destructor Documentation	23
6.3.2.1	ETRTO()	23
6.3.3	Member Function Documentation	24
6.3.3.1	print()	24
6.4	PatchTire::MagicFormula Class Reference	24
6.4.1	Detailed Description	27
6.4.2	Constructor & Destructor Documentation	27
6.4.2.1	MagicFormula()	27
6.4.3	Member Function Documentation	27
6.4.3.1	evaluateContact()	27
6.4.3.2	fourPointsSampling()	28
6.4.3.3	getArea() [1/2]	28
6.4.3.4	getArea() [2/2]	28
6.4.3.5	getEulerAngleX()	28
6.4.3.6	getEulerAngleY()	28
6.4.3.7	getEulerAngleZ()	29
6.4.3.8	getFriction() [1/2]	29
6.4.3.9	getFriction() [2/2]	29
6.4.3.10	getMFpoint() [1/2]	29
6.4.3.11	getMFpoint() [2/2]	29
6.4.3.12	getMFpointRF() [1/2]	30
6.4.3.13	getMFpointRF() [2/2]	30
6.4.3.14	getNormal() [1/2]	30
6.4.3.15	getNormal() [2/2]	30
6.4.3.16	getRelativeCamber()	31

6.4.3.17	<code>getRho()</code> [1/2]	31
6.4.3.18	<code>getRho()</code> [2/2]	31
6.4.3.19	<code>getRhoDot()</code> [1/2]	31
6.4.3.20	<code>getRhoDot()</code> [2/2]	32
6.4.3.21	<code>getVolume()</code> [1/2]	32
6.4.3.22	<code>getVolume()</code> [2/2]	32
6.4.3.23	<code>pointSampling()</code>	33
6.4.3.24	<code>print()</code>	33
6.4.3.25	<code>printETRTOGeometry()</code>	33
6.4.3.26	<code>setOrigin()</code>	34
6.4.3.27	<code>setReferenceFrame()</code>	34
6.4.3.28	<code>setRotationMatrix()</code>	34
6.4.3.29	<code>setTotalTransformationMatrix()</code>	34
6.4.3.30	<code>setup()</code>	35
6.5	RDF::MeshSurface Class Reference	35
6.5.1	Detailed Description	36
6.5.2	Constructor & Destructor Documentation	36
6.5.2.1	<code>MeshSurface()</code> [1/2]	36
6.5.2.2	<code>MeshSurface()</code> [2/2]	36
6.5.3	Member Function Documentation	36
6.5.3.1	<code>intersectAABBtree()</code>	36
6.5.3.2	<code>intersectBBox()</code>	36
6.5.3.3	<code>LoadFile()</code>	37
6.5.3.4	<code>printData()</code>	37
6.5.3.5	<code>set()</code>	37
6.6	PatchTire::MultiDisk Class Reference	37
6.6.1	Detailed Description	40
6.6.2	Constructor & Destructor Documentation	40
6.6.2.1	<code>MultiDisk()</code> [1/3]	40
6.6.2.2	<code>MultiDisk()</code> [2/3]	41
6.6.2.3	<code>MultiDisk()</code> [3/3]	41
6.6.3	Member Function Documentation	42
6.6.3.1	<code>getArea()</code> [1/2]	42
6.6.3.2	<code>getArea()</code> [2/2]	42
6.6.3.3	<code>getDiskFriction()</code>	42
6.6.3.4	<code>getDiskMFpoint()</code>	43
6.6.3.5	<code>getDiskMFpointRF()</code>	43
6.6.3.6	<code>getDiskNormal()</code>	43
6.6.3.7	<code>getDiskOriginXYZ()</code> [1/2]	43
6.6.3.8	<code>getDiskOriginXYZ()</code> [2/2]	44
6.6.3.9	<code>getDiskRho()</code>	44
6.6.3.10	<code>getDiskRhoDot()</code>	44
6.6.3.11	<code>getEulerAngleX()</code>	44
6.6.3.12	<code>getEulerAngleY()</code>	45

6.6.3.13	getEulerAngleZ()	45
6.6.3.14	getFriction() [1/2]	45
6.6.3.15	getFriction() [2/2]	45
6.6.3.16	getMFpoint() [1/2]	45
6.6.3.17	getMFpoint() [2/2]	46
6.6.3.18	getMFpointRF() [1/2]	46
6.6.3.19	getMFpointRF() [2/2]	46
6.6.3.20	getNormal() [1/2]	46
6.6.3.21	getNormal() [2/2]	47
6.6.3.22	getRelativeCamber()	47
6.6.3.23	getRho() [1/2]	47
6.6.3.24	getRho() [2/2]	47
6.6.3.25	getRhoDot() [1/2]	48
6.6.3.26	getRhoDot() [2/2]	48
6.6.3.27	getVolume() [1/2]	48
6.6.3.28	getVolume() [2/2]	48
6.6.3.29	pointSampling()	50
6.6.3.30	print()	50
6.6.3.31	printETRTOGeometry()	50
6.6.3.32	setDiskOriginXZ() [1/2]	51
6.6.3.33	setDiskOriginXZ() [2/2]	51
6.6.3.34	setOrigin()	51
6.6.3.35	setReferenceFrame()	51
6.6.3.36	setRotationMatrix()	52
6.6.3.37	setTotalTransformationMatrix()	52
6.6.3.38	setup()	52
6.7	PatchTire::ReferenceFrame Class Reference	52
6.7.1	Detailed Description	53
6.7.2	Constructor & Destructor Documentation	53
6.7.2.1	ReferenceFrame()	53
6.7.3	Member Function Documentation	54
6.7.3.1	getEulerAngleX()	54
6.7.3.2	getEulerAngleY()	54
6.7.3.3	getEulerAngleZ()	54
6.7.3.4	set()	54
6.7.3.5	setOrigin()	54
6.7.3.6	setRotationMatrix()	54
6.7.3.7	setTotalTransformationMatrix()	55
6.8	PatchTire::SamplingGrid Class Reference	55
6.8.1	Detailed Description	55
6.8.2	Constructor & Destructor Documentation	56
6.8.2.1	SamplingGrid() [1/2]	56
6.8.2.2	SamplingGrid() [2/2]	56
6.8.3	Member Function Documentation	56

6.8.3.1	set() [1/2]	56
6.8.3.2	set() [2/2]	57
6.8.3.3	setSwitchNumber()	57
6.9	PatchTire::Shadow Class Reference	57
6.9.1	Detailed Description	57
6.9.2	Constructor & Destructor Documentation	57
6.9.2.1	Shadow()	58
6.9.3	Member Function Documentation	58
6.9.3.1	update()	58
6.10	TicToc Class Reference	58
6.11	PatchTire::Tire Class Reference	58
6.11.1	Detailed Description	61
6.11.2	Constructor & Destructor Documentation	61
6.11.2.1	Tire()	61
6.11.3	Member Function Documentation	61
6.11.3.1	evaluateContact()	61
6.11.3.2	getArea() [1/2]	62
6.11.3.3	getArea() [2/2]	62
6.11.3.4	getEulerAngleX()	62
6.11.3.5	getEulerAngleY()	62
6.11.3.6	getEulerAngleZ()	62
6.11.3.7	getFriction() [1/2]	62
6.11.3.8	getFriction() [2/2]	63
6.11.3.9	getMFpoint() [1/2]	63
6.11.3.10	getMFpoint() [2/2]	63
6.11.3.11	getMFpointRF() [1/2]	63
6.11.3.12	getMFpointRF() [2/2]	64
6.11.3.13	getNormal() [1/2]	64
6.11.3.14	getNormal() [2/2]	64
6.11.3.15	getRelativeCamber()	64
6.11.3.16	getRho() [1/2]	65
6.11.3.17	getRho() [2/2]	65
6.11.3.18	getRhoDot() [1/2]	65
6.11.3.19	getRhoDot() [2/2]	66
6.11.3.20	getVolume() [1/2]	66
6.11.3.21	getVolume() [2/2]	66
6.11.3.22	pointSampling()	66
6.11.3.23	print()	67
6.11.3.24	printETRTOGeometry()	67
6.11.3.25	setOrigin()	67
6.11.3.26	setReferenceFrame()	68
6.11.3.27	setRotationMatrix()	68
6.11.3.28	setTotalTransformationMatrix()	68
6.11.3.29	setup()	68

6.12	RDF::Triangle3D Class Reference	69
6.12.1	Detailed Description	70
6.12.2	Constructor & Destructor Documentation	70
6.12.2.1	Triangle3D()	70
6.12.3	Member Function Documentation	70
6.12.3.1	intersectEdgePlane()	70
6.12.3.2	intersectPlane()	71
6.12.3.3	intersectRay()	71
6.12.3.4	print()	71
6.12.3.5	setVertices() [1/2]	73
6.12.3.6	setVertices() [2/2]	73
6.13	RDF::TriangleRoad Class Reference	73
6.13.1	Detailed Description	75
6.13.2	Constructor & Destructor Documentation	75
6.13.2.1	TriangleRoad()	75
6.13.3	Member Function Documentation	75
6.13.3.1	intersectEdgePlane()	75
6.13.3.2	intersectPlane()	76
6.13.3.3	intersectRay()	76
6.13.3.4	print()	76
6.13.3.5	setFriction()	77
6.13.3.6	setVertices() [1/2]	77
6.13.3.7	setVertices() [2/2]	77
	Index	79

Chapter 1

TireGround

A repository for the code developed by Davide Stocco for his thesis.

Department of Industrial Engineering
Master Degree in Mechatronics Engineering

EN: Real-Time Computation of Tire/Road Contact using Tailored Algorithms

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

Academic Year 2019 · 2020

Author: **Davide Stocco**

Supervisor & Co-supervisor: **Prof. Enrico Bertolazzi & Dr.Eng. Matteo Ragni**

MagicFormula tire model usage

1. Load .rdf file.

```
RDF::MeshSurface Road(  
    "./file.rdf" // Path to the *.rdf file  
);
```

2. Initialize the MagicFormula tire model.

```
PatchTire::Tire* TireSD = new PatchTire::MagicFormula(  
    SectionWidth, // [mm]  
    AspectRatio,  // [%]  
    RimDiameter,  // [in]  
    SwitchNumber  // Maximum RoadTriangles in the Tire Shadow (switch to sampling)  
);
```

3. Contact evaluation.

```
bool Out = TireSD->setup( Road,      // Road mesh  
                          TransfMat // 4x4 total transformation matrix  
);
```

4. Data extraction.

```
// Variable initialization (for real numbers)  
PatchTire::vec3 N;  
PatchTire::vec3 P;  
PatchTire::real_type Friction;  
PatchTire::real_type Rho;  
PatchTire::real_type RhoDot;  
PatchTire::real_type RelativeCamber;  
PatchTire::real_type Friction;  
PatchTire::real_type Area;  
PatchTire::real_type Volume;  
PatchTire::real_type RelativeCamber;  
  
// Data extraction (for real numbers)  
TireSD->getNormal(N);  
TireSD->getPoint(P);
```

```

TireSD->getFriction(Friction);
TireSD->getRho(Rho);
TireSD->getRhoDot(PreviousRho,TimeStep,RhoDot);
TireSD->getRelativeCamber(RelativeCamber);
TireSD->getArea(Area);
TireSD->getVolume(Volume);
TireSD->getRelativeCamber(RelativeCamber)

// Extract data stucture size
PatchTire::int_type size = TireSD->getDisksNumber();

// Variable initialization (for vectors)
PatchTire::row_vec3 NVec(size);
PatchTire::row_vec3 PVec(size);
PatchTire::row_vecN FrictionVec(size);
PatchTire::row_vecN RhoVec(size);
PatchTire::row_vecN RhoDotVec(size);
PatchTire::row_vecN RelativeCamberVec(size);
PatchTire::row_vecN FrictionVec(size);
PatchTire::row_vecN AreaVec(size);
PatchTire::row_vecN VolumeVec(size);
PatchTire::row_vecN RelativeCamberVec(size);

// Data extraction (for vectors)
TireSD->getNormal(NVec);
TireSD->getPoint(PVec);
TireSD->getFriction(FrictionVec);
TireSD->getRho(RhoVec);
TireSD->getRhoDot(PreviousRho,TimeStep,RhoDotVec);
TireSD->getRelativeCamber(RelativeCamberVec);
TireSD->getArea(AreaVec);
TireSD->getVolume(VolumeVec);
TireSD->getRelativeCamber(RelativeCambeVecr)

```

MultiDisk tire model usage

1. Load .rdf file.

```

RDF::MeshSurface Road(
    "../file.rdf" // Path to the *.rdf file
);

```

2. Initialize the MultiDisk tire model:

- (a) MultiDisk tire without sidewall radius (uniform cylinder).

```

PatchTire::Tire* TireMD = new PatchTire::MultiDisk(
    SectionWidth, // [mm]
    AspectRatio,  // [%]
    RimDiameter,  // [in]
    PointsNumber, // Sampling points for each disk
    DisksNumber,  // Disks number
    SwitchNumber  // Maximum RoadTriangles in the Tire Shadow (switch to sampling)
);

```

- (b) MultiDisk tire with sidewall radius (uniform cylinder with filleted sidewall edge).

```

PatchTire::Tire* TireMD = new PatchTire::MultiDisk(
    SectionWidth, // [mm]
    AspectRatio,  // [%]
    RimDiameter,  // [in]
    SideRadius,   // Sidewall radius [mm]
    PointsNumber, // Sampling points for each disk
    DisksNumber,  // Disks number
    SwitchNumber  // Maximum RoadTriangles in the Tire Shadow (switch to sampling)
);

```

- (c) MultiDisk tire with custom disks radius.

```

PatchTire::Tire* TireMD = new PatchTire::MultiDisk(
    SectionWidth, // [mm]
    AspectRatio,  // [%]
    RimDiameter,  // [in]
    RadiusVec,    // Disks radius vector [m]
    PointsNumber, // Sampling points for each disk
    SwitchNumber  // Maximum RoadTriangles in the Tire Shadow (switch to sampling)
);

```

3. Contact evaluation.

```
bool Out = TireMD->setup( Road,      // Road mesh
                        TransfMat // 4x4 total transformation matrix
                        );
```

4. Data extraction for contact point(s).

```
// Variable initialization (for real numbers)
PatchTire::vec3 N;
PatchTire::vec3 P;
PatchTire::real_type Friction;
PatchTire::real_type Rho;
PatchTire::real_type RhoDot;
PatchTire::real_type RelativeCamber;
PatchTire::real_type Friction;
PatchTire::real_type Area;
PatchTire::real_type Volume;
PatchTire::real_type RelativeCamber;

// Data extraction (for real numbers)
TireMD->getNormal(N);
TireMD->getPoint(P);
TireMD->getFriction(Friction);
TireMD->getRho(Rho);
TireMD->getRhoDot(PreviousRho, TimeStep, RhoDot);
TireMD->getRelativeCamber(RelativeCamber);
TireMD->getArea(Area);
TireMD->getVolume(Volume);
TireMD->getRelativeCamber(RelativeCamber)

// Extract data stucture size
PatchTire::int_type size = TireSD->getDisksNumber();

// Variable initialization (for vectors)
PatchTire::row_vec3 NVec(size);
PatchTire::row_vec3 PVec(size);
PatchTire::row_vecN FrictionVec(size);
PatchTire::row_vecN RhoVec(size);
PatchTire::row_vecN RhoDotVec(size);
PatchTire::row_vecN RelativeCamberVec(size);
PatchTire::row_vecN FrictionVec(size);
PatchTire::row_vecN AreaVec(size);
PatchTire::row_vecN VolumeVec(size);
PatchTire::row_vecN RelativeCamberVec(size);

// Data extraction (for vectors)
TireMD->getNormal(NVec);
TireMD->getPoint(PVec);
TireMD->getFriction(FrictionVec);
TireMD->getRho(RhoVec);
TireMD->getRhoDot(PreviousRho, TimeStep, RhoDotVec);
TireMD->getRelativeCamber(RelativeCamberVec);
TireMD->getArea(AreaVec);
TireMD->getVolume(VolumeVec);
TireMD->getRelativeCamber(RelativeCambeVecr)
```


Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

PatchTire	
Tire computations routines	11
PatchTire::algorithms	
Algorithms for tire computations routine	11
RDF	
RDF mesh computations routines	14
RDF::algorithms	
Algorithms for RDF mesh computations routine	15
TireGround	
Typedefs for tire computations routine	16

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

RDF::BBox2D	19
PatchTire::Disk	20
PatchTire::ETRTO	23
RDF::MeshSurface	35
PatchTire::ReferenceFrame	52
PatchTire::SamplingGrid	55
PatchTire::Shadow	57
TicToc	58
PatchTire::Tire	58
PatchTire::MagicFormula	24
PatchTire::MultiDisk	37
RDF::Triangle3D	69
RDF::TriangleRoad	73

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

RDF::BBox2D	
2D Bounding Box class	19
PatchTire::Disk	
Tire disk	20
PatchTire::ETRTO	
Tire ETRTO denomination	23
PatchTire::MagicFormula	
Pacejka MagicFormula contact model	24
RDF::MeshSurface	
Mesh surface	35
PatchTire::MultiDisk	
Multi-disk tire contact model	37
PatchTire::ReferenceFrame	
Reference frame	52
PatchTire::SamplingGrid	
Patch evaluation precision	55
PatchTire::Shadow	
2D shadow (2D bounding box enhancement)	57
TicToc	58
PatchTire::Tire	
Base class for Tire models	58
RDF::Triangle3D	
3D triangle (pure geometrical description)	69
RDF::TriangleRoad	
3D triangles for road representation	73

Chapter 5

Namespace Documentation

5.1 PatchTire Namespace Reference

[Tire](#) computations routines.

Namespaces

- [algorithms](#)
Algorithms for tire computations routine.

Classes

- class [Disk](#)
Tire disk.
- class [ETRTO](#)
Tire ETRTO denomination.
- class [MagicFormula](#)
Pacejka [MagicFormula](#) contact model.
- class [MultiDisk](#)
Multi-disk tire contact model.
- class [ReferenceFrame](#)
Reference frame.
- class [SamplingGrid](#)
Patch evaluation precision.
- class [Shadow](#)
2D shadow (2D bounding box enhancement)
- class [Tire](#)
Base class for [Tire](#) models.

5.1.1 Detailed Description

[Tire](#) computations routines.

file: [PatchTire.hh](#)

5.2 PatchTire::algorithms Namespace Reference

Algorithms for tire computations routine.

Functions

- `real_type weightedMean (row_vecN const &Values, row_vecN const &Weights)`
Calculate arithmetic weighted mean for real numbers.
- `vec3 weightedMean (row_vec3 const &Values, row_vecN const &Weights)`
Calculate arithmetic weighted mean for 3D vectors.
- `bool intersectPointSegment (vec2 const &Point1, vec2 const &Point2, vec2 const &PointQ)`
- `bool intersectRayPlane (vec3 const &planeN, vec3 const &planeP, vec3 const &RayPoint, vec3 const &RayDirection, vec3 &IntersectionPt)`
Check if a segment hits a plane and find the intersection point.
- `void minmax_XY (row_vec3 const &Points, vec2 &XYmin, vec2 &XYmax)`
Calculate minimum and maximum in XY plane for 3D vectors.
- `void minmax_XY (row_vec2 const &Points, vec2 &XYmin, vec2 &XYmax)`
Calculate minimum and maximum in XY plane for 2D vectors.
- `real_type trapezoidArea (real_type const Base_A, real_type const Base_B, real_type const Height)`
Calculate area of a trapezoid [m²].

5.2.1 Detailed Description

Algorithms for tire computations routine.

5.2.2 Function Documentation

5.2.2.1 intersectPointSegment()

```
bool PatchTire::algorithms::intersectPointSegment (
    vec2 const & Point1,
    vec2 const & Point2,
    vec2 const & PointQ )
```

Check if a point lays inside or outside a line segment

Warning: The point query point must be on the same rect of the line segment!

Parameters

<i>Point1</i>	Line segment point 1
<i>Point2</i>	Line segment point 2
<i>PointQ</i>	Query point

5.2.2.2 intersectRayPlane()

```
bool PatchTire::algorithms::intersectRayPlane (
    vec3 const & planeN,
    vec3 const & planeP,
    vec3 const & RayPoint,
    vec3 const & RayDirection,
    vec3 & IntersectionPt )
```

Check if a segment hits a plane and find the intersection point.

Parameters

<i>planeN</i>	Plane normal vector
<i>planeP</i>	Plane known point
<i>RayPoint</i>	Ray point
<i>RayDirection</i>	Ray direction
<i>IntersectionPt</i>	Intersection point

5.2.2.3 minmax_XY() [1/2]

```
void PatchTire::algorithms::minmax_XY (
    row_vec3 const & Points,
    vec2 & XYmin,
    vec2 & XYmax )
```

Calculate minumum and maximum in XY plane for 3D vectors.

Parameters

<i>Points</i>	3D points vector
<i>XYmin</i>	Minimum (X , Y) values
<i>XYmax</i>	Maximum (X , Y) values

5.2.2.4 minmax_XY() [2/2]

```
void PatchTire::algorithms::minmax_XY (
    row_vec2 const & Points,
    vec2 & XYmin,
    vec2 & XYmax )
```

Calculate minumum and maximum in XY plane for 2D vectors.

Parameters

<i>Points</i>	2D points vector
<i>XYmin</i>	Minimum (X , Y) values
<i>XYmax</i>	Maximum (X , Y) values

5.2.2.5 trapezoidArea()

```
real_type PatchTire::algorithms::trapezoidArea (
    real_type const Base_A,
    real_type const Base_B,
    real_type const Height ) [inline]
```

Calculate area of a trapeziod [m^2].

Parameters

<i>Base_A</i>	Base 1
<i>Base_B</i>	Base 2
<i>Height</i>	Heigth

5.2.2.6 `weightedMean()` [1/2]

```
real_type PatchTire::algorithms::weightedMean (
    row_vecN const & Values,
    row_vecN const & Weights )
```

Calculate arithmetic weighted mean for real numbers.

Parameters

<i>Values</i>	Values (real numbers)
<i>Weights</i>	Weights (real numbers)

5.2.2.7 `weightedMean()` [2/2]

```
vec3 PatchTire::algorithms::weightedMean (
    row_vec3 const & Values,
    row_vecN const & Weights )
```

Calculate arithmetic weighted mean for 3D vectors.

Parameters

<i>Values</i>	Values (3D vectors)
<i>Weights</i>	Weights (real numbers)

5.3 RDF Namespace Reference

[RDF](#) mesh computations routines.

Namespaces

- [algorithms](#)
Algorithms for [RDF](#) mesh computations routine.

Classes

- class [BBox2D](#)
2D Bounding Box class
- class [MeshSurface](#)

Mesh surface.

- class [Triangle3D](#)
3D triangle (pure geometrical description)
- class [TriangleRoad](#)
3D triangles for road representation

Typedefs

- 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.

5.3.1 Detailed Description

[RDF](#) mesh computations routines.

5.4 RDF::algorithms Namespace Reference

Algorithms for [RDF](#) mesh computations routine.

Functions

- void [split](#) (std::string const &in, std::vector< std::string > &out, std::string const &token)
Split a string into a string array at a given token.
- std::string [tail](#) (std::string const &in)
Get tail of string after first token and possibly following spaces.
- std::string [firstToken](#) (std::string const &in)
Get first token of string.
- template<typename T >
T const & [getElement](#) (std::vector< T > const &elements, std::string const &index)
Get element at given index position.

5.4.1 Detailed Description

Algorithms for [RDF](#) mesh computations routine.

5.4.2 Function Documentation

5.4.2.1 firstToken()

```
std::string RDF::algorithms::firstToken (
    std::string const & in )
```

Get first token of string.

Parameters

<i>in</i>	Input string
-----------	--------------

5.4.2.2 getElement()

```
template<typename T >
T const& RDF::algorithms::getElement (
    std::vector< T > const & elements,
    std::string const & index )
```

Get element at given index position.

Parameters

<i>elements</i>	Elements vector
<i>index</i>	Index position

5.4.2.3 split()

```
void RDF::algorithms::split (
    std::string const & in,
    std::vector< std::string > & out,
    std::string const & token )
```

Split a string into a string array at a given token.

Parameters

<i>in</i>	Input string
<i>out</i>	Output string vector
<i>token</i>	Token

5.4.2.4 tail()

```
std::string RDF::algorithms::tail (
    std::string const & in )
```

Get tail of string after first token and possibly following spaces.

Parameters

<i>in</i>	Input string
-----------	--------------

5.5 TireGround Namespace Reference

Typedefs for tire computations routine.

Typedefs

- typedef double [real_type](#)
Real number type.

- typedef int [int_type](#)
Integer number type.
- typedef Eigen::Vector2i [vec2_int](#)
2D vector type of real integer type
- typedef Eigen::Vector2d [vec2](#)
2D vector type of real number type
- typedef Eigen::Vector3d [vec3](#)
3D vector type of real number type
- typedef Eigen::Vector4d [vec4](#)
4D vector type of real number type
- typedef Eigen::Matrix3d [mat3](#)
3x3 matrix type of real number type
- typedef Eigen::Matrix4d [mat4](#)
4x4 matrix type of real number type
- typedef Eigen::Matrix< [real_type](#), 1, Eigen::Dynamic > [row_vecN](#)
Row vector type real number type.
- typedef Eigen::Matrix< [real_type](#), Eigen::Dynamic, 1 > [col_vecN](#)
Column vector type real number type.
- typedef Eigen::Matrix< [real_type](#), Eigen::Dynamic, Eigen::Dynamic > [matN](#)
Matrix type of real number type.
- typedef Eigen::Matrix< [vec2](#), 1, Eigen::Dynamic > [row_vec2](#)
Row vector type of 2D vector.
- typedef Eigen::Matrix< [vec2](#), Eigen::Dynamic, 1 > [col_vec2](#)
Column vector type of 2D vector.
- typedef Eigen::Matrix< [vec2](#), Eigen::Dynamic, Eigen::Dynamic > [mat_vec2](#)
Matrix type of 2D vector.
- typedef Eigen::Matrix< [vec3](#), 1, Eigen::Dynamic > [row_vec3](#)
Row vector type of 3D vector.
- typedef Eigen::Matrix< [vec3](#), Eigen::Dynamic, 1 > [col_vec3](#)
Column vector type of 3D vector.
- typedef Eigen::Matrix< [vec3](#), Eigen::Dynamic, Eigen::Dynamic > [matN_vec3](#)
Matrix type of 3D vector.
- typedef Eigen::Matrix< [mat4](#), 1, Eigen::Dynamic > [row_mat4](#)
Matrix type of 4x4 matrix.
- typedef std::basic_ostream< char > [ostream_type](#)
Output stream type.

Variables

- [real_type](#) const [epsilon](#) = std::numeric_limits<[real_type](#)>::epsilon()
Epsilon type.

5.5.1 Detailed Description

Typedefs for tire computations routine.

file: [TireGround.hh](#)

Chapter 6

Class Documentation

6.1 RDF::BBox2D Class Reference

2D Bounding Box class

```
#include <RoadRDF.hh>
```

Public Member Functions

- [BBox2D](#) ()
Default constructor.
- [BBox2D](#) ([vec3](#) const Vertices[3])
Variable set constructor.
- void [setXmin](#) ([real_type](#) const _Xmin)
Set X_{min} shadow domain.
- void [setYmin](#) ([real_type](#) const _Ymin)
Set Y_{min} shadow domain.
- void [setXmax](#) ([real_type](#) const _Xmax)
Set X_{max} shadow domain.
- void [setYmax](#) ([real_type](#) const _Ymax)
Set Y_{max} shadow domain.
- [real_type](#) [getXmin](#) (void) const
Get X_{min} shadow domain.
- [real_type](#) [getYmin](#) (void) const
Get Y_{min} shadow domain.
- [real_type](#) [getXmax](#) (void) const
Get X_{max} shadow domain.
- [real_type](#) [getYmax](#) (void) const
Get Y_{max} shadow domain.
- void [clear](#) (void)
Clear the bounding box domain.
- void [print](#) ([ostream_type](#) &stream) const
Print bounding box domain.
- void [updateBBox2D](#) ([vec3](#) const Vertices[3])
Update the bounding box domain with three input vertices.

6.1.1 Detailed Description

2D Bounding Box class

6.1.2 Constructor & Destructor Documentation

6.1.2.1 BBox2D()

```
RDF::BBox2D::BBox2D (
    vec3 const Vertices[3] ) [inline]
```

Variable set constructor.

Parameters

<i>Vertices</i>	Vertices reference vector
-----------------	---------------------------

6.1.3 Member Function Documentation

6.1.3.1 print()

```
void RDF::BBox2D::print (
    ostream_type & stream ) const [inline]
```

Print bounding box domain.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

6.1.3.2 updateBBox2D()

```
void RDF::BBox2D::updateBBox2D (
    vec3 const Vertices[3] )
```

Update the bounding box domain with three input vertices.

Parameters

<i>Vertices</i>	Vertices reference vector
-----------------	---------------------------

The documentation for this class was generated from the following file:

- include/RoadRDF.hh

6.2 PatchTire::Disk Class Reference

[Tire](#) disk.

```
#include <PatchTire.hh>
```

Public Member Functions

- [Disk](#) ([Disk](#) &&)=default

Enable `&&` operator.

- `Disk ()`

Default constructor.

- `Disk (vec2 const &_OriginXZ, real_type _OffsetY, real_type _Radius)`

Variable set constructor.

- `void set (Disk const &in)`

Copy the `Disk` object.

- `void setOriginXZ (vec2 const &_OriginXZ)`

Set origin on XZ plane.

- `vec2 const & getOriginXZ () const`

Get origin vector XZ -axes coordinates.

- `vec3 getOriginXYZ () const`

Get origin vector XYZ -axes coordinates.

- `real_type getOffsetY () const`

Get origin Y -axis coordinate.

- `real_type getRadius () const`

Get `Disk` radius.

- `void contactTriangles (RDF::TriangleRoad_list const &TriList, ReferenceFrame const &RF, vec3 &Normal, real_type &Friction, real_type &Area) const`

- `void contactPlane (vec3 const &Normal, vec3 const &Point, ReferenceFrame const &RF, real_type &Area) const`

- `void pointOnDisk (vec3 const &Normal, ReferenceFrame const &RF, vec3 &DiskPoint, vec3 &NormalOnDisk) const`

Get the points on `Disk` the circumference and on a given plane.

6.2.1 Detailed Description

`Tire` disk.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 Disk()

```
PatchTire::Disk::Disk (
    vec2 const &_OriginXZ,
    real_type _OffsetY,
    real_type _Radius ) [inline]
```

Variable set constructor.

Parameters

<code>_OriginXZ</code>	(X_0, Z_0) origin coordinate
<code>_OffsetY</code>	Y_0 origin coordinate (offset from center)
<code>_Radius</code>	Radius

6.2.3 Member Function Documentation

6.2.3.1 contactPlane()

```
void PatchTire::Disk::contactPlane (
    vec3 const & Normal,
    vec3 const & Point,
    ReferenceFrame const & RF,
    real_type & Area ) const
```

Get the contact area [m^2] inside the single [Disk](#) given a plane in absolute reference frame

Parameters

<i>Normal</i>	Plane normal in absolute reference frame
<i>Point</i>	Plane point in absolute reference frame
<i>RF</i>	Tire ReferenceFrame
<i>Area</i>	Contact area [m^2]

6.2.3.2 contactTriangles()

```
void PatchTire::Disk::contactTriangles (
    RDF::TriangleRoad_list const & TriList,
    ReferenceFrame const & RF,
    vec3 & Normal,
    real_type & Friction,
    real_type & Area ) const
```

Get area weighted mean road normal versor, area weighted mean friction and contact area [m^2] inside the single [Disk](#) of segments described by the intersection of triangles on XZ -plane

Parameters

<i>TriList</i>	Shadow / MeshSurface intersected triangles
<i>RF</i>	Tire ReferenceFrame
<i>Normal</i>	Area weighted mean road normal versor
<i>Friction</i>	Area weighted mean contact friction
<i>Area</i>	Contact area [m^2]

6.2.3.3 set()

```
void PatchTire::Disk::set (
    Disk const & in ) [inline]
```

Copy the [Disk](#) object.

Parameters

<i>in</i>	Disk object to be copied
-----------	--

6.2.3.4 setOriginXZ()

```
void PatchTire::Disk::setOriginXZ (
    vec2 const & _OriginXZ ) [inline]
```

Set origin on XZ plane.

Parameters

<code>_OriginXZ</code>	New origin on XZ plane
------------------------	--------------------------

The documentation for this class was generated from the following file:

- include/PatchTire.hh

6.3 PatchTire::ETRTO Class Reference

[Tire ETRTO](#) denomination.

```
#include <PatchTire.hh>
```

Public Member Functions

- [ETRTO](#) ()
Default constructor.
- [ETRTO](#) ([real_type](#) _SectionWidth, [real_type](#) _AspectRatio, [real_type](#) _RimDiameter)
Variable set constructor.
- [real_type](#) getSidewallHeight (void) const
Get sidewall height [m].
- [real_type](#) getTireDiameter (void) const
Get external tire diameter [m].
- [real_type](#) getTireRadius (void) const
Get external tire radius [m].
- [real_type](#) getSectionWidth (void) const
Get section width [m].
- void [print](#) ([ostream_type](#) &stream) const
Display tire data.

6.3.1 Detailed Description

[Tire ETRTO](#) denomination.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 ETRTO()

```
PatchTire::ETRTO::ETRTO (
    real\_type _SectionWidth,
    real\_type _AspectRatio,
    real\_type _RimDiameter ) [inline]
```

Variable set constructor.

Parameters

<i>_SectionWidth</i>	Tire section width [<i>mm</i>]
<i>_AspectRatio</i>	Tire aspect ratio [%]
<i>_RimDiameter</i>	Rim diameter [<i>in</i>]

6.3.3 Member Function Documentation

6.3.3.1 print()

```
void PatchTire::ETRTO::print (
    ostream_type & stream ) const [inline]
```

Display tire data.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

The documentation for this class was generated from the following file:

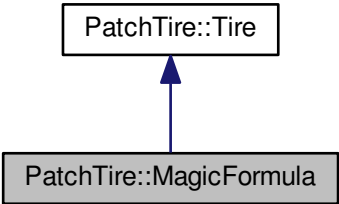
- include/PatchTire.hh

6.4 PatchTire::MagicFormula Class Reference

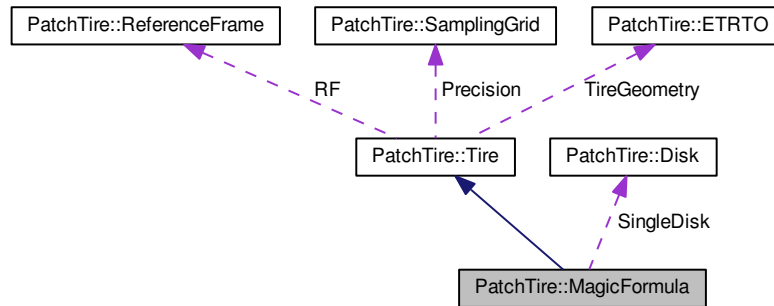
Pacejka [MagicFormula](#) contact model.

```
#include <PatchTire.hh>
```

Inheritance diagram for PatchTire::MagicFormula:



Collaboration diagram for PatchTire::MagicFormula:



Public Member Functions

- `~MagicFormula ()`
Default destructor.
- `MagicFormula (real_type const SectionWidth, real_type const AspectRatio, real_type const RimDiameter, int_type const SwitchN)`
Variable set constructor.
- `void getNormal (vec3 &_Normal) const override`
Get contact normal versor.
- `void getNormal (row_vec3 &_Normal) const override`
Get contact normal versors vector.
- `void getMFpoint (vec3 &_DiskPoint) const override`
Get Magic Formula contact point.
- `void getMFpoint (row_vec3 &_DiskPoint) const override`
Get Magic Formula contact point vector.
- `void getFriction (real_type &_Friction) const override`
Get contact point friction.
- `void getFriction (row_vecN &_Friction) const override`
Get contact point friction vector.
- `void getMFpointRF (mat4 &PointRF) const override`
Get Magic Formula contact point reference frame with 4x4 transformation matrix.
- `void getMFpointRF (row_mat4 &_MFpointRF) const override`
Get Magic Formula contact point reference frame vector with 4x4 transformation matrix.
- `void getRho (real_type &Rho) const override`
- `void getRho (row_vecN &Rho) const override`
- `void getRhoDot (real_type const &Rho, real_type const &Time, real_type &RhoDot) const override`
Get contact depth time derivative [m/s].
- `void getRhoDot (row_vecN const &Rho, real_type const &Time, row_vecN &RhoDot) const override`
Get contact depth time derivative vector [m/s].
- `void getArea (real_type &_Area) const override`
Get approximated contact area on Disk plane [m²].
- `void getArea (row_vecN &_Area) const override`
Get approximated contact area vector on Disk plane [m²].

- void `getVolume` (`real_type` &_Volume) const override
Get approximated contact volume [m³].
- void `getVolume` (`row_vecN` &Volume) const override
Get approximated contact volume vector [m³].
- bool `setup` (`RDF::MeshSurface` &Mesh, `mat4` const &TM) override
Update current tire position and find contact parameters.
- void `print` (`ostream_type` &stream) const override
Print contact parameters.
- void `printETRTOGeometry` (`ostream_type` &stream) const
Display Tire ETRTO geometry data.
- void `setReferenceFrame` (`ReferenceFrame` const &_RF)
- `ReferenceFrame` const & `getReferenceFrame` (void) const
Get tire ReferenceFrame object.
- void `setOrigin` (`vec3` const &Origin)
Set a new tire origin.
- void `setRotationMatrix` (`mat3` const &RotationMatrix)
- void `setTotalTransformationMatrix` (`mat4` const &TM)
- `real_type` `getEulerAngleX` (void) const
- `real_type` `getEulerAngleY` (void) const
- `real_type` `getEulerAngleZ` (void) const
- void `getRelativeCamber` (`real_type` &RelativeCamber) const
Get relative camber angle [rad].
- `int_type` `getDisksNumber` (void) const
Dimension of the contact points data structure (disks number)

Protected Member Functions

- `MagicFormula` (`MagicFormula` const &)=delete
Deleted copy constructor.
- `MagicFormula` const & `operator=` (`MagicFormula` const &)=delete
Deleted copy operator.
- void `evaluateContact` (`RDF::TriangleRoad_list` const &TriList) override
Evaluate contact with RoadTriangles.
- void `fourPointsSampling` (`RDF::TriangleRoad_list` const &TriList, `vec3` &P_star)
*Perform triangles sampling on 4 points at $\pm 0.1 * R$ along X and $\pm 0.3 * W$ along Y.*
- bool `pointSampling` (`RDF::TriangleRoad_list` const &TriList, `vec3` const &RayOrigin, `vec3` const &RayDirection, `vec3` &SampledPt, `real_type` &TriFriction=quietNaN, `vec3` &TriNormal=`vec3`↔NaN) const
Perform one point sampling (ray-triangle intersection)

Protected Attributes

- `Disk` `SingleDisk`
Single Disk.
- `vec3` `Normal`
Contact normal versor.
- `vec3` `MeshPoint`
Contact point on Mesh (not for Magic Formula)
- `vec3` `DiskPoint`
Contact point on undeformed Disk circumference (for Magic Formula)
- `real_type` `Friction`

- [Contact friction](#).
Contact friction.
- [real_type Area](#)
Contact area [m²].
- [SamplingGrid Precision](#)
Contact patch evaluating precision.
- [ETRTO TireGeometry](#)
Tire ETRTO denomination.
- [ReferenceFrame RF](#)
ReferenceFrame.

6.4.1 Detailed Description

Pacejka [MagicFormula](#) contact model.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 MagicFormula()

```
PatchTire::MagicFormula::MagicFormula (
    real_type const SectionWidth,
    real_type const AspectRatio,
    real_type const RimDiameter,
    int_type const SwitchN ) [inline]
```

Variable set constructor.

Parameters

<i>SectionWidth</i>	Tire section width [mm]
<i>AspectRatio</i>	Tire aspect ratio [%]
<i>RimDiameter</i>	Rim diameter [in]
<i>SwitchN</i>	Maximum RoadTriangles in the Tire Shadow (switch to sampling)

6.4.3 Member Function Documentation

6.4.3.1 evaluateContact()

```
void PatchTire::MagicFormula::evaluateContact (
    RDF::TriangleRoad_list const & TriList ) [override], [protected], [virtual]
```

Evaluate contact with RoadTriangles.

Parameters

<i>TriList</i>	Shadow/MeshSurface intersected triangles
----------------	--

Implements [PatchTire::Tire](#).

6.4.3.2 fourPointsSampling()

```
void PatchTire::MagicFormula::fourPointsSampling (
    RDF::TriangleRoad_list const & TriList,
    vec3 & P_star ) [protected]
```

Perform triangles sampling on 4 points at $\pm 0.1 \cdot R$ along X and $\pm 0.3 \cdot W$ along Y .

Parameters

<i>TriList</i>	Shadow/MeshSurface intersected triangles
----------------	--

6.4.3.3 getArea() [1/2]

```
void PatchTire::MagicFormula::getArea (
    real_type & _Area ) const [inline], [override], [virtual]
```

Get approximated contact area on [Disk](#) plane [m^2].

Parameters

<i>_Area</i>	Contact area [m^2]
--------------	------------------------

Implements [PatchTire::Tire](#).

6.4.3.4 getArea() [2/2]

```
void PatchTire::MagicFormula::getArea (
    row_vecN & _Area ) const [inline], [override], [virtual]
```

Get approximated contact area vector on [Disk](#) plane [m^2].

Parameters

<i>_Area</i>	Contact area vector [m^2]
--------------	-------------------------------

Implements [PatchTire::Tire](#).

6.4.3.5 getEulerAngleX()

```
real_type PatchTire::Tire::getEulerAngleX (
    void ) const [inline], [inherited]
```

Get current Euler angles [rad] for X -axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.4.3.6 getEulerAngleY()

```
real_type PatchTire::Tire::getEulerAngleY (
    void ) const [inline], [inherited]
```

Get current Euler angles [rad] for Y -axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.4.3.7 getEulerAngleZ()

```
real_type PatchTire::Tire::getEulerAngleZ (
    void ) const [inline], [inherited]
```

Get current Euler angles [*rad*] for *Z*-axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.4.3.8 getFriction() [1/2]

```
void PatchTire::MagicFormula::getFriction (
    real_type & _Friction ) const [inline], [override], [virtual]
```

Get contact point friction.

Parameters

<code>_Friction</code>	Contact point friction
------------------------	------------------------

Implements [PatchTire::Tire](#).

6.4.3.9 getFriction() [2/2]

```
void PatchTire::MagicFormula::getFriction (
    row_vecN & _Friction ) const [inline], [override], [virtual]
```

Get contact point friction vector.

Parameters

<code>_Friction</code>	Contact point friction vector
------------------------	-------------------------------

Implements [PatchTire::Tire](#).

6.4.3.10 getMFpoint() [1/2]

```
void PatchTire::MagicFormula::getMFpoint (
    vec3 & _DiskPoint ) const [inline], [override], [virtual]
```

Get Magic Formula contact point.

Parameters

<code>_DiskPoint</code>	Magic Formula contact point
-------------------------	-----------------------------

Implements [PatchTire::Tire](#).

6.4.3.11 getMFpoint() [2/2]

```
void PatchTire::MagicFormula::getMFpoint (
    row_vec3 & _DiskPoint ) const [inline], [override], [virtual]
```

Get Magic Formula contact point vector.

Parameters

<i><code>_DiskPoint</code></i>	Contact point vector on Disk
--------------------------------	--

Implements [PatchTire::Tire](#).

6.4.3.12 `getMFpointRF()` [1/2]

```
void PatchTire::MagicFormula::getMFpointRF (
    mat4 & PointRF ) const [override], [virtual]
```

Get Magic Formula contact point reference frame with 4x4 transformation matrix.

Parameters

<i><code>PointRF</code></i>	Magic Formula contact point reference frame
-----------------------------	---

Implements [PatchTire::Tire](#).

6.4.3.13 `getMFpointRF()` [2/2]

```
void PatchTire::MagicFormula::getMFpointRF (
    row\_mat4 & MFpointRF ) const [inline], [override], [virtual]
```

Get Magic Formula contact point reference frame vector with 4x4 transformation matrix.

Parameters

<i><code>MFpointRF</code></i>	Magic Formula onctact point reference frames vector
-------------------------------	---

Implements [PatchTire::Tire](#).

6.4.3.14 `getNormal()` [1/2]

```
void PatchTire::MagicFormula::getNormal (
    vec3 & _Normal ) const [inline], [override], [virtual]
```

Get contact normal versor.

Parameters

<i><code>_Normal</code></i>	Contact point normal versor
-----------------------------	-----------------------------

Implements [PatchTire::Tire](#).

6.4.3.15 `getNormal()` [2/2]

```
void PatchTire::MagicFormula::getNormal (
    row\_vec3 & _Normal ) const [inline], [override], [virtual]
```

Get contact normal versors vector.

Parameters

<i>_Normal</i>	Contact point normal direction vector
----------------	---------------------------------------

Implements [PatchTire::Tire](#).

6.4.3.16 getRelativeCamber()

```
void PatchTire::Tire::getRelativeCamber (
    real_type & RelativeCamber ) const [inherited]
```

Get relative camber angle [*rad*].

Parameters

<i>RelativeCamber</i>	Relative camber angle
-----------------------	-----------------------

6.4.3.17 getRho() [1/2]

```
void PatchTire::MagicFormula::getRho (
    real_type & Rho ) const [inline], [override], [virtual]
```

Get contact depth at center point [*m*]

Warning: (if negative the tire does not touch the ground)!

Parameters

<i>Rho</i>	Depth at center point
------------	-----------------------

Implements [PatchTire::Tire](#).

6.4.3.18 getRho() [2/2]

```
void PatchTire::MagicFormula::getRho (
    row_vecN & Rho ) const [inline], [override], [virtual]
```

Get contact depth matrix [*m*]

Warning: (if negative the tire does not touch the ground)!

Parameters

<i>Rho</i>	Depth matrix
------------	--------------

Implements [PatchTire::Tire](#).

6.4.3.19 getRhoDot() [1/2]

```
void PatchTire::MagicFormula::getRhoDot (
    real_type const & Rho,
```

```

    real_type const & Time,
    real_type & RhoDot ) const [inline], [override], [virtual]

```

Get contact depth time derivative [m/s].

Parameters

<i>Rho</i>	Previous time step Rho [m]
<i>Time</i>	Time step [s]
<i>RhoDot</i>	Penetration derivative [m/s]

Implements [PatchTire::Tire](#).

6.4.3.20 getRhoDot() [2/2]

```

void PatchTire::MagicFormula::getRhoDot (
    row_vecN const & Rho,
    real_type const & Time,
    row_vecN & RhoDot ) const [inline], [override], [virtual]

```

Get contact depth time derivative vector [m/s].

Parameters

<i>Rho</i>	Previous time step Rho [m]
<i>Time</i>	Time step [s]
<i>RhoDot</i>	Penetration derivative [m/s]

Implements [PatchTire::Tire](#).

6.4.3.21 getVolume() [1/2]

```

void PatchTire::MagicFormula::getVolume (
    real_type & _Volume ) const [inline], [override], [virtual]

```

Get approximated contact volume [m^3].

Parameters

<i>_Volume</i>	Contact volume [m^3]
----------------	--------------------------

Implements [PatchTire::Tire](#).

6.4.3.22 getVolume() [2/2]

```

void PatchTire::MagicFormula::getVolume (
    row_vecN & Volume ) const [inline], [override], [virtual]

```

Get approximated contact volume vector [m^3].

Parameters

<i>Volume</i>	Contact volume vector [m^3]
---------------	---------------------------------

Implements [PatchTire::Tire](#).

6.4.3.23 pointSampling()

```
bool PatchTire::Tire::pointSampling (
    RDF::TriangleRoad_list const & TriList,
    vec3 const & RayOrigin,
    vec3 const & RayDirection,
    vec3 & SampledPt,
    real_type & TriFriction = quietNaN,
    vec3 & TriNormal = vec3_NaN ) const [protected], [inherited]
```

Perform one point sampling (ray-triangle intersection)

Parameters

<i>TriList</i>	Shadow/MeshSurface intersected triangles
<i>RayOrigin</i>	Ray origin
<i>RayDirection</i>	Ray direction
<i>SampledPt</i>	Intersection point
<i>TriFriction</i>	Intersected triangle friction
<i>TriNormal</i>	Intersected triangle normal

6.4.3.24 print()

```
void PatchTire::MagicFormula::print (
    ostream_type & stream ) const [override], [virtual]
```

Print contact parameters.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

Implements [PatchTire::Tire](#).

6.4.3.25 printETRTOGeometry()

```
void PatchTire::Tire::printETRTOGeometry (
    ostream_type & stream ) const [inline], [inherited]
```

Display [Tire](#) ETRTO geometry data.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

6.4.3.26 setOrigin()

```
void PatchTire::Tire::setOrigin (
    vec3 const & Origin ) [inline], [inherited]
```

Set a new tire origin.

Parameters

<i>Origin</i>	Tire origin
---------------	-------------

6.4.3.27 setReferenceFrame()

```
void PatchTire::Tire::setReferenceFrame (
    ReferenceFrame const & _RF ) [inline], [inherited]
```

Copy the tire **ReferenceFrame** object

Warning: Rotation matrix must be orthonormal!

Parameters

<i>_RF</i>	ReferenceFrame object to be copied
------------	---

6.4.3.28 setRotationMatrix()

```
void PatchTire::Tire::setRotationMatrix (
    mat3 const & RotationMatrix ) [inline], [inherited]
```

Set a new 3x3 rotation matrix

Warning: Rotation matrix must be orthonormal!

Parameters

<i>RotationMatrix</i>	Rotation matrix
-----------------------	-----------------

6.4.3.29 setTotalTransformationMatrix()

```
void PatchTire::Tire::setTotalTransformationMatrix (
    mat4 const & TM ) [inline], [inherited]
```

Set 4x4 total transformation matrix

Warning: Rotation matrix must be orthonormal!

Parameters

<i>TM</i>	4x4 total transformation matrix
-----------	---------------------------------

6.4.3.30 setup()

```
bool PatchTire::MagicFormula::setup (
    RDF::MeshSurface & Mesh,
    mat4 const & TM ) [override], [virtual]
```

Update current tire position and find contact parameters.

Parameters

<i>Mesh</i>	MeshSurface object (road)
<i>TM</i>	4x4 total transformation matrix

Implements [PatchTire::Tire](#).

The documentation for this class was generated from the following file:

- include/PatchTire.hh

6.5 RDF::MeshSurface Class Reference

Mesh surface.

```
#include <RoadRDF.hh>
```

Public Member Functions

- [MeshSurface](#) ()
Default set constructor.
- [MeshSurface](#) ([TriangleRoad_list](#) const &_PtrTriangleVec)
Variable set constructor.
- [MeshSurface](#) (std::string const &Path)
Variable set constructor.
- [TriangleRoad_list](#) const & [getTrianglesList](#) (void) const
Get all triangles inside the mesh as a vector.
- [TriangleRoad_ptr](#) const [getTriangle](#) (unsigned i) const
Get i-th TriangleRoad.
- G2lib::AABBtree::PtrAABB const [getAABBPtr](#) (void) const
Get AABBtree object.
- void [printData](#) (std::string const &FileName) const
Print data in file.
- std::vector< G2lib::BBox::PtrBBox > const & [getPtrBBoxList](#) () const
Get the mesh G2lib bounding boxes pointers vector.
- void [set](#) ([MeshSurface](#) const &in)
Copy the MeshSurface object.
- bool [LoadFile](#) (std::string const &Path)
Load the RDF model and print information on a file.
- bool [intersectAABBtree](#) (G2lib::AABBtree::PtrAABB const &AABBTreePtr, [RDF::TriangleRoad_list](#) &TrianglesList) const
Update the local intersected TriangleRoad vector list.
- bool [intersectBBox](#) (std::vector< G2lib::BBox::PtrBBox > const &BBoxPtr, [RDF::TriangleRoad_list](#) &TrianglesList) const
Update the mesh AABBtree with an external G2lib::BBox object pointer vector.

6.5.1 Detailed Description

Mesh surface.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 MeshSurface() [1/2]

```
RDF::MeshSurface::MeshSurface (
    TriangleRoad_list const & _PtrTriangleVec ) [inline]
```

Variable set constructor.

Parameters

<i>_PtrTriangleVec</i>	Road triangles pointer vector list
------------------------	------------------------------------

6.5.2.2 MeshSurface() [2/2]

```
RDF::MeshSurface::MeshSurface (
    std::string const & Path ) [inline]
```

Variable set constructor.

Parameters

<i>Path</i>	Path to the RDF file
-------------	--------------------------------------

6.5.3 Member Function Documentation

6.5.3.1 intersectAABBtree()

```
bool RDF::MeshSurface::intersectAABBtree (
    G2lib::AABBtree::PtrAABB const & AABBTreePtr,
    RDF::TriangleRoad_list & TrianglesList ) const
```

Update the local intersected [TriangleRoad](#) vector list.

Parameters

<i>AABBTreePtr</i>	External AABBtree object pointer
<i>TrianglesList</i>	Intersected TriangleRoad vector list

6.5.3.2 intersectBBox()

```
bool RDF::MeshSurface::intersectBBox (
    std::vector< G2lib::BBox::PtrBBox > const & BBoxPtr,
    RDF::TriangleRoad_list & TrianglesList ) const
```

Update the mesh AABBtree with an external G2lib::BBox object pointer vector.

Parameters

<i>BBoxPtr</i>	External G2lib::BBox object pointer vector
<i>TrianglesList</i>	Intersected TriangleRoad vector list

6.5.3.3 LoadFile()

```
bool RDF::MeshSurface::LoadFile (
    std::string const & Path )
```

Load the [RDF](#) model and print information on a file.

Parameters

<i>Path</i>	Path to the RDF file
-------------	--------------------------------------

6.5.3.4 printData()

```
void RDF::MeshSurface::printData (
    std::string const & FileName ) const
```

Print data in file.

Parameters

<i>FileName</i>	File name in which print data
-----------------	-------------------------------

6.5.3.5 set()

```
void RDF::MeshSurface::set (
    MeshSurface const & in ) [inline]
```

Copy the [MeshSurface](#) object.

Parameters

<i>in</i>	MeshSurface object to be copied
-----------	---

The documentation for this class was generated from the following file:

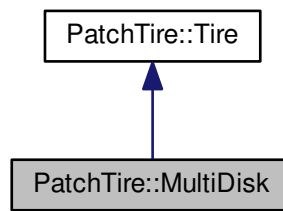
- include/RoadRDF.hh

6.6 PatchTire::MultiDisk Class Reference

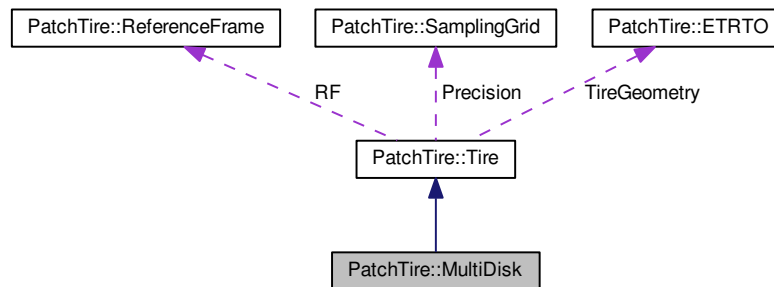
Multi-disk tire contact model.

```
#include <PatchTire.hh>
```

Inheritance diagram for PatchTire::MultiDisk:



Collaboration diagram for PatchTire::MultiDisk:



Public Member Functions

- [~MultiDisk](#) ()
Default destructor.
- [MultiDisk](#) ([real_type](#) const SectionWidth, [real_type](#) const AspectRatio, [real_type](#) const Rim↔Diameter, [int_type](#) const PointsN, [int_type](#) const DisksN, [int_type](#) const SwitchN)
Variable set constructor.
- [MultiDisk](#) ([real_type](#) const SectionWidth, [real_type](#) const AspectRatio, [real_type](#) const Rim↔Diameter, [real_type](#) const SideRadius, [int_type](#) const PointsN, [int_type](#) const DisksN, [int_type](#) const SwitchN)
Variable set constructor.
- [MultiDisk](#) ([real_type](#) const SectionWidth, [real_type](#) const AspectRatio, [real_type](#) const Rim↔Diameter, [row_vecN](#) const DisksRadius, [int_type](#) const PointsN, [int_type](#) const SwitchN)
Variable set constructor.
- [real_type](#) [getPointstep](#) (void) const
Get grid step on X-axis between sampling points [m].
- [real_type](#) [getDiskStep](#) (void) const
Get step on Y-axis between disks [m].
- void [getNormal](#) ([vec3](#) &_Normal) const override
Get contact normal mean versor.
- void [getDiskOriginXYZ](#) ([row_vec3](#) &Origin) const

- *Get disks origin (X,Y,Z).*
- void `getDiskOriginXYZ` (`int_type` const i, `vec3` &Origin) const
Get i-th Disk origin (X,Y,Z).
- void `setDiskOriginXZ` (`row_vec2` &Origin) const
Set disks origin (X,Y,Z).
- void `setDiskOriginXZ` (`int_type` const i, `vec2` &Origin) const
Set i-th Disk origin (X,Y,Z).
- void `getNormal` (`row_vec3` &_NormalVec) const override
Get contact normal versors vector.
- void `getDiskNormal` (`int_type` const i, `vec3` &_Normal) const
Get i-th Disk contact normal versor.
- void `getMFpoint` (`vec3` &_DiskPoint) const override
Get Magic Formula contact point.
- void `getMFpoint` (`row_vec3` &_DiskPointVec) const override
Get Magic Formula contact points vector.
- void `getDiskMFpoint` (`int_type` const i, `vec3` &_DiskPoint) const
Get i-th Disk Magic Formula contact point.
- void `getFriction` (`real_type` &_Friction) const override
Get area weighted mean contact friction.
- void `getFriction` (`row_vecN` &_Friction) const override
Get contact frictions vector.
- void `getDiskFriction` (`int_type` const i, `real_type` &_Friction) const
Get i-th Disk contact friction.
- void `getMFpointRF` (`mat4` &PointRF) const override
Get Magic Formula contact point reference frame with 4x4 transformation matrix.
- void `getMFpointRF` (`row_mat4` &PointRF) const override
Get Magic Formula contact point reference frames vector with 4x4 transformation matrix.
- void `getDiskMFpointRF` (`int_type` const i, `mat4` &PointRF) const
Get Disk Magic Formula contact point reference frame with 4x4 transformation matrix.
- void `getRho` (`real_type` &Rho) const override
- void `getRho` (`row_vecN` &Rho) const override
- void `getDiskRho` (`int_type` const i, `real_type` &Rho) const
- void `getRhoDot` (`real_type` const &Rho, `real_type` const &Time, `real_type` &RhoDot) const override
Get contact depth time derivative [m/s].
- void `getRhoDot` (`row_vecN` const &Rho, `real_type` const &Time, `row_vecN` &RhoDot) const override
Get contact depths derivative vector [m/s].
- void `getDiskRhoDot` (`int_type` const i, `real_type` const &Rho, `real_type` const &Time, `real_type` &RhoDot) const
Get i-th Disk contact depth derivative [m/s].
- void `getArea` (`real_type` &_Area) const override
Get approximated mean contact area on Disk plane [m²].
- void `getArea` (`row_vecN` &_AreaVec) const override
Get approximated contact areas vector on Disk plane [m²].
- void `getVolume` (`real_type` &Volume) const override
Get approximated contact volume [m³].
- void `getVolume` (`row_vecN` &Volume) const override
Get approximated contact volumes vector [m³].
- bool `setup` (`RDF::MeshSurface` &Mesh, `mat4` const &TM) override
Update current tire position and find contact parameters.

- void [print](#) ([ostream_type](#) &stream) const override
Print contact parameters.
- void [printETRTOGeometry](#) ([ostream_type](#) &stream) const
Display Tire ETRTO geometry data.
- void [setReferenceFrame](#) ([ReferenceFrame](#) const &_RF)
- [ReferenceFrame](#) const & [getReferenceFrame](#) (void) const
Get tire ReferenceFrame object.
- void [setOrigin](#) ([vec3](#) const &Origin)
Set a new tire origin.
- void [setRotationMatrix](#) ([mat3](#) const &RotationMatrix)
- void [setTotalTransformationMatrix](#) ([mat4](#) const &TM)
- [real_type](#) [getEulerAngleX](#) (void) const
- [real_type](#) [getEulerAngleY](#) (void) const
- [real_type](#) [getEulerAngleZ](#) (void) const
- void [getRelativeCamber](#) ([real_type](#) &RelativeCamber) const
Get relative camber angle [rad].
- [int_type](#) [getDisksNumber](#) (void) const
Dimension of the contact points data structure (disks number)

Protected Member Functions

- bool [pointSampling](#) ([RDF::TriangleRoad_list](#) const &TriList, [vec3](#) const &RayOrigin, [vec3](#) const &RayDirection, [vec3](#) &SampledPt, [real_type](#) &TriFriction=quietNaN, [vec3](#) &TriNormal=[vec3](#) ← NaN) const
Perform one point sampling (ray-triangle intersection)

Protected Attributes

- [SamplingGrid Precision](#)
Contact patch evaluating precision.
- [ETRTO TireGeometry](#)
Tire ETRTO denomination.
- [ReferenceFrame RF](#)
ReferenceFrame.

6.6.1 Detailed Description

Multi-disk tire contact model.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 MultiDisk() [1/3]

```
PatchTire::MultiDisk::MultiDisk (
    real\_type const SectionWidth,
    real\_type const AspectRatio,
    real\_type const RimDiameter,
    int\_type const PointsN,
    int\_type const DisksN,
    int\_type const SwitchN ) [inline]
```

Variable set constructor.

Parameters

<i>SectionWidth</i>	Tire section width [<i>mm</i>]
<i>AspectRatio</i>	Tire aspect ratio [%]
<i>RimDiameter</i>	Rim diameter [<i>in</i>]
<i>PointsN</i>	Sampling points for each Disk (divisions on <i>X</i> -axis)
<i>DisksN</i>	Number of Disks (divisions on <i>Y</i> -axis -1)
<i>SwitchN</i>	Maximum RoadTriangles in the Tire Shadow (switch to sampling)

6.6.2.2 MultiDisk() [2/3]

```
PatchTire::MultiDisk::MultiDisk (
    real_type const SectionWidth,
    real_type const AspectRatio,
    real_type const RimDiameter,
    real_type const SideRadius,
    int_type const PointsN,
    int_type const DisksN,
    int_type const SwitchN ) [inline]
```

Variable set constructor.

Parameters

<i>SectionWidth</i>	Tire section width [<i>mm</i>]
<i>AspectRatio</i>	Tire aspect ratio [%]
<i>RimDiameter</i>	Rim diameter [<i>in</i>]
<i>SideRadius</i>	Sidewall radius [<i>mm</i>]
<i>PointsN</i>	Sampling points for each Disk (divisions on <i>X</i> -axis)
<i>DisksN</i>	Number of Disks (divisions on <i>Y</i> -axis -1)
<i>SwitchN</i>	Maximum RoadTriangles in the Tire Shadow (switch to sampling)

6.6.2.3 MultiDisk() [3/3]

```
PatchTire::MultiDisk::MultiDisk (
    real_type const SectionWidth,
    real_type const AspectRatio,
    real_type const RimDiameter,
    row_vecN const DisksRadius,
    int_type const PointsN,
    int_type const SwitchN ) [inline]
```

Variable set constructor.

Parameters

<i>SectionWidth</i>	Tire section width [<i>mm</i>]
<i>AspectRatio</i>	Tire aspect ratio [%]
<i>RimDiameter</i>	Rim diameter [<i>in</i>]

Parameters

<i>DisksRadius</i>	Disks radius vector [<i>m</i>]
<i>PointsN</i>	Sampling points for each Disk (divisions on <i>X</i> -axis)
<i>SwitchN</i>	Maximum RoadTriangles in the Tire Shadow (switch to sampling)

6.6.3 Member Function Documentation

6.6.3.1 `getArea()` [1/2]

```
void PatchTire::MultiDisk::getArea (
    real\_type & _Area ) const [inline], [override], [virtual]
```

Get approximated mean contact area on [Disk](#) plane [*m*²].

Parameters

<i>_Area</i>	Contact area [<i>m</i> ²]
--------------	--

Implements [PatchTire::Tire](#).

6.6.3.2 `getArea()` [2/2]

```
void PatchTire::MultiDisk::getArea (
    row\_vecN & _AreaVec ) const [inline], [override], [virtual]
```

Get approximated contact areas vector on [Disk](#) plane [*m*²].

Parameters

<i>_AreaVec</i>	Contact areas vector [<i>m</i> ²]
-----------------	--

Implements [PatchTire::Tire](#).

6.6.3.3 `getDiskFriction()`

```
void PatchTire::MultiDisk::getDiskFriction (
    int\_type const i,
    real\_type & _Friction ) const [inline]
```

Get *i*-th [Disk](#) contact friction.

Parameters

<i>i</i>	<i>i</i> -th Disk
<i>_Friction</i>	Disk contact friction

6.6.3.4 getDiskMFpoint()

```
void PatchTire::MultiDisk::getDiskMFpoint (
    int_type const i,
    vec3 & _DiskPoint ) const [inline]
```

Get i -th [Disk](#) Magic Formula contact point.

Parameters

i	i -th Disk
$_DiskPoint$	Disk Magic Formula contact point

6.6.3.5 getDiskMFpointRF()

```
void PatchTire::MultiDisk::getDiskMFpointRF (
    int_type const i,
    mat4 & PointRF ) const [inline]
```

Get [Disk](#) Magic Formula contact point reference frame with 4x4 transformation matrix.

Parameters

i	i -th Disk
$PointRF$	Magic Formula contact point reference frame

6.6.3.6 getDiskNormal()

```
void PatchTire::MultiDisk::getDiskNormal (
    int_type const i,
    vec3 & _Normal ) const [inline]
```

Get i -th [Disk](#) contact normal versor.

Parameters

i	i -th Disk
$_Normal$	Contact normal versor

6.6.3.7 getDiskOriginXYZ() [1/2]

```
void PatchTire::MultiDisk::getDiskOriginXYZ (
    row_vec3 & Origin ) const [inline]
```

Get disks origin (X, Y, Z).

Parameters

$Origin$	Disks origin
----------	--------------

6.6.3.8 getDiskOriginXYZ() [2/2]

```
void PatchTire::MultiDisk::getDiskOriginXYZ (
    int_type const i,
    vec3 & Origin ) const [inline]
```

Get i -th [Disk](#) origin (X, Y, Z).

Parameters

i	i -th Disk
$Origin$	Disks origin

6.6.3.9 getDiskRho()

```
void PatchTire::MultiDisk::getDiskRho (
    int_type const i,
    real_type & Rho ) const [inline]
```

Get i -th [Disk](#) contact depth [m]

Warning: (if negative the tire does not touch the ground)!

Parameters

i	i -th Disk
Rho	Disk contact depth

6.6.3.10 getDiskRhoDot()

```
void PatchTire::MultiDisk::getDiskRhoDot (
    int_type const i,
    real_type const & Rho,
    real_type const & Time,
    real_type & RhoDot ) const [inline]
```

Get i -th [Disk](#) contact depth derivative [m/s].

Parameters

i	i -th Disk
Rho	Previous time step Rho [m]
$Time$	Time step [s]
$RhoDot$	Disk contact depth derivative [m/s]

6.6.3.11 getEulerAngleX()

```
real_type PatchTire::Tire::getEulerAngleX (
    void ) const [inline], [inherited]
```

Get current Euler angles [rad] for X -axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.6.3.12 getEulerAngleY()

```
real_type PatchTire::Tire::getEulerAngleY (
    void ) const [inline], [inherited]
```

Get current Euler angles [*rad*] for *Y*-axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.6.3.13 getEulerAngleZ()

```
real_type PatchTire::Tire::getEulerAngleZ (
    void ) const [inline], [inherited]
```

Get current Euler angles [*rad*] for *Z*-axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.6.3.14 getFriction() [1/2]

```
void PatchTire::MultiDisk::getFriction (
    real_type & _Friction ) const [inline], [override], [virtual]
```

Get area weighted mean contact friction.

Parameters

<i>_Friction</i>	Area weighted mean contact friction
------------------	-------------------------------------

Implements [PatchTire::Tire](#).

6.6.3.15 getFriction() [2/2]

```
void PatchTire::MultiDisk::getFriction (
    row_vecN & _Friction ) const [inline], [override], [virtual]
```

Get contact frictions vector.

Parameters

<i>_Friction</i>	Contact frictions vector
------------------	--------------------------

Implements [PatchTire::Tire](#).

6.6.3.16 getMFpoint() [1/2]

```
void PatchTire::MultiDisk::getMFpoint (
    vec3 & _DiskPoint ) const [inline], [override], [virtual]
```

Get Magic Formula contact point.

Parameters

<i>_DiskPoint</i>	Magic Formula contact point
-------------------	-----------------------------

Implements [PatchTire::Tire](#).

6.6.3.17 getMFpoint() [2/2]

```
void PatchTire::MultiDisk::getMFpoint (
    row_vec3 & _DiskPointVec ) const [inline], [override], [virtual]
```

Get Magic Formula contact points vector.

Parameters

<i>_DiskPointVec</i>	Magic Formula contact points vector
----------------------	-------------------------------------

Implements [PatchTire::Tire](#).

6.6.3.18 getMFpointRF() [1/2]

```
void PatchTire::MultiDisk::getMFpointRF (
    mat4 & PointRF ) const [inline], [override], [virtual]
```

Get Magic Formula contact point reference frame with 4x4 transformation matrix.

Parameters

<i>PointRF</i>	Magic Formula contact point reference frame
----------------	---

Implements [PatchTire::Tire](#).

6.6.3.19 getMFpointRF() [2/2]

```
void PatchTire::MultiDisk::getMFpointRF (
    row_mat4 & PointRF ) const [inline], [override], [virtual]
```

Get Magic Formula contact point reference frames vector with 4x4 transformation matrix.

Parameters

<i>PointRF</i>	Magic Formula contact point reference frames vector
----------------	---

Implements [PatchTire::Tire](#).

6.6.3.20 getNormal() [1/2]

```
void PatchTire::MultiDisk::getNormal (
    vec3 & _Normal ) const [inline], [override], [virtual]
```

Get contact normal mean versor.

Parameters

<i>_Normal</i>	Contact normal mean versor
----------------	----------------------------

Implements [PatchTire::Tire](#).

6.6.3.21 getNormal() [2/2]

```
void PatchTire::MultiDisk::getNormal (
    row_vec3 & _NormalVec ) const [inline], [override], [virtual]
```

Get contact normal versors vector.

Parameters

<i>_NormalVec</i>	Contact normal versors vector
-------------------	-------------------------------

Implements [PatchTire::Tire](#).

6.6.3.22 getRelativeCamber()

```
void PatchTire::Tire::getRelativeCamber (
    real_type & RelativeCamber ) const [inherited]
```

Get relative camber angle [*rad*].

Parameters

<i>RelativeCamber</i>	Relative camber angle
-----------------------	-----------------------

6.6.3.23 getRho() [1/2]

```
void PatchTire::MultiDisk::getRho (
    real_type & Rho ) const [inline], [override], [virtual]
```

Get contact depth at center point [*m*]

Warning: (if negative the tire does not touch the ground)!

Parameters

<i>Rho</i>	Depth at center point
------------	-----------------------

Implements [PatchTire::Tire](#).

6.6.3.24 getRho() [2/2]

```
void PatchTire::MultiDisk::getRho (
    row_vecN & Rho ) const [inline], [override], [virtual]
```

Get contact depths vector [*m*]

Warning: (if negative the tire does not touch the ground)!

Parameters

<i>Rho</i>	Contact depths vector
------------	-----------------------

Implements [PatchTire::Tire](#).

6.6.3.25 getRhoDot() [1/2]

```
void PatchTire::MultiDisk::getRhoDot (
    real_type const & Rho,
    real_type const & Time,
    real_type & RhoDot ) const [inline], [override], [virtual]
```

Get contact depth time derivative [m/s].

Parameters

<i>Rho</i>	Previous time step Rho [m]
<i>Time</i>	Time step [s]
<i>RhoDot</i>	Contact depth derivative [m/s]

Implements [PatchTire::Tire](#).

6.6.3.26 getRhoDot() [2/2]

```
void PatchTire::MultiDisk::getRhoDot (
    row_vecN const & Rho,
    real_type const & Time,
    row_vecN & RhoDot ) const [inline], [override], [virtual]
```

Get contact depths derivative vector [m/s].

Parameters

<i>Rho</i>	Previous time step Rho [m]
<i>Time</i>	Time step [s]
<i>RhoDot</i>	Contact depths derivative vector [m/s]

Implements [PatchTire::Tire](#).

6.6.3.27 getVolume() [1/2]

```
void PatchTire::MultiDisk::getVolume (
    real_type & Volume ) const [inline], [override], [virtual]
```

Get approximated contact volume [m^3].

Parameters

<i>Volume</i>	Contact volume [m^3]
---------------	--------------------------

Implements [PatchTire::Tire](#).

6.6.3.28 getVolume() [2/2]

```
void PatchTire::MultiDisk::getVolume (
    row_vecN & Volume ) const [inline], [override], [virtual]
```

Get approximated contact volumes vector [m^3].

Parameters

<i>Volume</i>	Contact volumes vector [m^3]
---------------	----------------------------------

Implements [PatchTire::Tire](#).

6.6.3.29 pointSampling()

```
bool PatchTire::Tire::pointSampling (
    RDF::TriangleRoad_list const & TriList,
    vec3 const & RayOrigin,
    vec3 const & RayDirection,
    vec3 & SampledPt,
    real_type & TriFriction = quietNaN,
    vec3 & TriNormal = vec3_NaN ) const [protected], [inherited]
```

Perform one point sampling (ray-triangle intersection)

Parameters

<i>TriList</i>	Shadow/MeshSurface intersected triangles
<i>RayOrigin</i>	Ray origin
<i>RayDirection</i>	Ray direction
<i>SampledPt</i>	Intersection point
<i>TriFriction</i>	Intersected triangle friction
<i>TriNormal</i>	Intersected triangle normal

6.6.3.30 print()

```
void PatchTire::MultiDisk::print (
    ostream_type & stream ) const [override], [virtual]
```

Print contact parameters.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

Implements [PatchTire::Tire](#).

6.6.3.31 printETRTOGeometry()

```
void PatchTire::Tire::printETRTOGeometry (
    ostream_type & stream ) const [inline], [inherited]
```

Display [Tire ETRTO](#) geometry data.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

6.6.3.32 setDiskOriginXZ() [1/2]

```
void PatchTire::MultiDisk::setDiskOriginXZ (
    row_vec2 & Origin ) const [inline]
```

Set disks origin (X, Y, Z).

Parameters

<i>Origin</i>	New Disks origin vector
---------------	-------------------------

6.6.3.33 setDiskOriginXZ() [2/2]

```
void PatchTire::MultiDisk::setDiskOriginXZ (
    int_type const i,
    vec2 & Origin ) const [inline]
```

Set i -th [Disk](#) origin (X, Y, Z).

Parameters

i	i -th Disk
<i>Origin</i>	New Disks origin vector

6.6.3.34 setOrigin()

```
void PatchTire::Tire::setOrigin (
    vec3 const & Origin ) [inline], [inherited]
```

Set a new tire origin.

Parameters

<i>Origin</i>	Tire origin
---------------	-----------------------------

6.6.3.35 setReferenceFrame()

```
void PatchTire::Tire::setReferenceFrame (
    ReferenceFrame const & _RF ) [inline], [inherited]
```

Copy the tire [ReferenceFrame](#) object

Warning: Rotation matrix must be orthonormal!

Parameters

$_RF$	ReferenceFrame object to be copied
--------	--

6.6.3.36 setRotationMatrix()

```
void PatchTire::Tire::setRotationMatrix (
    mat3 const & RotationMatrix ) [inline], [inherited]
```

Set a new 3x3 rotation matrix

Warning: Rotation matrix must be orthonormal!

Parameters

<i>RotationMatrix</i>	Rotation matrix
-----------------------	-----------------

6.6.3.37 setTotalTransformationMatrix()

```
void PatchTire::Tire::setTotalTransformationMatrix (
    mat4 const & TM ) [inline], [inherited]
```

Set 4x4 total transformation matrix

Warning: Rotation matrix must be orthonormal!

Parameters

<i>TM</i>	4x4 total transformation matrix
-----------	---------------------------------

6.6.3.38 setup()

```
bool PatchTire::MultiDisk::setup (
    RDF::MeshSurface & Mesh,
    mat4 const & TM ) [override], [virtual]
```

Update current tire position and find contact parameters.

Parameters

<i>Mesh</i>	MeshSurface object (road)
<i>TM</i>	4x4 total transformation matrix

Implements [PatchTire::Tire](#).

The documentation for this class was generated from the following file:

- include/PatchTire.hh

6.7 PatchTire::ReferenceFrame Class Reference

Reference frame.

```
#include <PatchTire.hh>
```

Public Member Functions

- [ReferenceFrame](#) ()
Default constructor.

- [ReferenceFrame](#) ([vec3](#) const &_Origin, [mat3](#) const &_RotationMatrix)
Variable set constructor.
- bool [isEmpty](#) (void)
Check if [ReferenceFrame](#) object is empty.
- [mat3](#) const & [getRotationMatrix](#) (void) const
Get current 3x3 rotation matrix.
- [mat3](#) [getRotationMatrixInverse](#) (void) const
Get current 3x3 rotation matrix inverse.
- [vec3](#) [getX](#) (void) const
Get current X-axis versor.
- [vec3](#) [getY](#) (void) const
Get current Y-axis versor.
- [vec3](#) [getZ](#) (void) const
Get current Z-axis versor.
- [vec3](#) const & [getOrigin](#) (void) const
Get origin position.
- void [setOrigin](#) ([vec3](#) const &_Origin)
Set origin position.
- void [setRotationMatrix](#) ([mat3](#) const &_RotationMatrix)
Set 3x3 rotation matrix.
- void [setTotalTransformationMatrix](#) ([mat4](#) const &TM)
Set 4x4 total transformation matrix.
- [mat4](#) [getTotalTransformationMatrix](#) (void)
Get 4x4 total transformation matrix.
- void [set](#) ([ReferenceFrame](#) const &in)
- [real_type](#) [getEulerAngleX](#) (void) const
- [real_type](#) [getEulerAngleY](#) (void) const
- [real_type](#) [getEulerAngleZ](#) (void) const

6.7.1 Detailed Description

Reference frame.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 ReferenceFrame()

```
PatchTire::ReferenceFrame::ReferenceFrame (
    vec3 const &_Origin,
    mat3 const &_RotationMatrix ) [inline]
```

Variable set constructor.

Parameters

_Origin	Origin position
_RotationMatrix	3x3 rotation matrix

6.7.3 Member Function Documentation

6.7.3.1 `getEulerAngleX()`

```
real_type PatchTire::ReferenceFrame::getEulerAngleX (
    void ) const
```

Get current Euler angles [*rad*] for *X*-axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.7.3.2 `getEulerAngleY()`

```
real_type PatchTire::ReferenceFrame::getEulerAngleY (
    void ) const
```

Get current Euler angles [*rad*] for *Y*-axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.7.3.3 `getEulerAngleZ()`

```
real_type PatchTire::ReferenceFrame::getEulerAngleZ (
    void ) const
```

Get current Euler angles [*rad*] for *Z*-axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.7.3.4 `set()`

```
void PatchTire::ReferenceFrame::set (
    ReferenceFrame const & in ) [inline]
```

Copy the tire [ReferenceFrame](#) object

Warning: Rotation matrix must be orthonormal!

Parameters

<i>in</i>	ReferenceFrame object to be copied
-----------	--

6.7.3.5 `setOrigin()`

```
void PatchTire::ReferenceFrame::setOrigin (
    vec3 const & _Origin ) [inline]
```

Set origin position.

Parameters

<i>_Origin</i>	Origin position
----------------	-----------------

6.7.3.6 `setRotationMatrix()`

```
void PatchTire::ReferenceFrame::setRotationMatrix (
```

```
mat3 const & _RotationMatrix ) [inline]
```

Set 3x3 rotation matrix.

Parameters

<code>_RotationMatrix</code>	3x3 rotation matrix
------------------------------	---------------------

6.7.3.7 setTotalTransformationMatrix()

```
void PatchTire::ReferenceFrame::setTotalTransformationMatrix (
    mat4 const & TM ) [inline]
```

Set 4x4 total transformation matrix.

Parameters

<code>TM</code>	4x4 total transformation matrix
-----------------	---------------------------------

The documentation for this class was generated from the following file:

- include/PatchTire.hh

6.8 PatchTire::SamplingGrid Class Reference

Patch evaluation precision.

```
#include <PatchTire.hh>
```

Public Member Functions

- [SamplingGrid](#) ()
Default constructor.
- [SamplingGrid](#) (int_type _PointsN, int_type _DisksN)
Variable set constructor.
- [SamplingGrid](#) (int_type _PointsN, int_type _DisksN, int_type _Switch)
Variable set constructor.
- int_type [getPointsNumber](#) (void) const
Get number of sampling points for each [Disk](#) (divisions on X-axis)
- int_type [getDisksNumber](#) (void) const
Get number of Disks (divisions on Y-axis -1)
- unsigned [getSwitchNumber](#) (void) const
Get number of maximum RoadTriangles in the [Tire Shadow](#) (switch to sampling)
- void [setSwitchNumber](#) (int_type const _Switch)
Set number of maximum RoadTriangles in the [Tire Shadow](#) (switch to sampling)
- void [set](#) (int_type _PointsN, int_type _DisksN, int_type _Switch)
Set number of divisions.
- void [set](#) ([SamplingGrid](#) const &in)
Copy the [SamplingGrid](#) object.

6.8.1 Detailed Description

Patch evaluation precision.

6.8.2 Constructor & Destructor Documentation

6.8.2.1 SamplingGrid() [1/2]

```
PatchTire::SamplingGrid::SamplingGrid (
    int_type _PointsN,
    int_type _DisksN ) [inline]
```

Variable set constructor.

Parameters

<code>_PointsN</code>	Sampling points for each Disk (divisions on <i>X</i> -axis)
<code>_DisksN</code>	Number of Disks (divisions on <i>Y</i> -axis −1)

6.8.2.2 SamplingGrid() [2/2]

```
PatchTire::SamplingGrid::SamplingGrid (
    int_type _PointsN,
    int_type _DisksN,
    int_type _Switch ) [inline]
```

Variable set constructor.

Parameters

<code>_PointsN</code>	Sampling points for each Disk (divisions on <i>X</i> -axis)
<code>_DisksN</code>	Number of Disks (divisions on <i>Y</i> -axis −1)
<code>_Switch</code>	Maximum RoadTriangles in the Tire Shadow (switch to sampling)

6.8.3 Member Function Documentation

6.8.3.1 set() [1/2]

```
void PatchTire::SamplingGrid::set (
    int_type _PointsN,
    int_type _DisksN,
    int_type _Switch ) [inline]
```

Set number of divisions.

Parameters

<code>_PointsN</code>	Sampling points for each Disk (divisions on <i>X</i> -axis)
<code>_DisksN</code>	Number of Disks (divisions on <i>Y</i> -axis −1)
<code>_Switch</code>	Maximum RoadTriangles in the Tire Shadow (switch to sampling)

6.8.3.2 set() [2/2]

```
void PatchTire::SamplingGrid::set (
    SamplingGrid const & in ) [inline]
```

Copy the [SamplingGrid](#) object.

Parameters

<i>in</i>	SamplingGrid object to be copied
-----------	--

6.8.3.3 setSwitchNumber()

```
void PatchTire::SamplingGrid::setSwitchNumber (
    int\_type const _Switch ) [inline]
```

Set number of maximum RoadTriangles in the [Tire Shadow](#) (switch to sampling)

Parameters

<i>_Switch</i>	New switch number
----------------	-------------------

The documentation for this class was generated from the following file:

- [include/PatchTire.hh](#)

6.9 PatchTire::Shadow Class Reference

2D shadow (2D bounding box enhacement)

```
#include <PatchTire.hh>
```

Public Member Functions

- [Shadow](#) ()
Default constructor.
- [Shadow](#) ([ETRTO](#) const &[TireGeometry](#), [ReferenceFrame](#) const &[RF](#))
- void [update](#) ([ETRTO](#) const &[TireGeometry](#), [ReferenceFrame](#) const &[RF](#))
- [G2lib::AABBtree::PtrAABB](#) const [getAABBtree](#) (void) const
Get total [Tire](#) [G2Lib::AABBtree](#) (3D projection on ground)
- [G2lib::AABBtree::PtrAABB](#) const [getUpperSideAABBtree](#) (void) const
Get upper side [Tire](#) [G2Lib::AABBtree](#) (3D projection on ground)
- [G2lib::AABBtree::PtrAABB](#) const [getLowerSideAABBtree](#) (void) const
Get lower side [Tire](#) [G2Lib::AABBtree](#) (3D projection on ground)

6.9.1 Detailed Description

2D shadow (2D bounding box enhacement)

6.9.2 Constructor & Destructor Documentation

6.9.2.1 Shadow()

```
PatchTire::Shadow::Shadow (
    ETRTO const & TireGeometry,
    ReferenceFrame const & RF ) [inline]
```

Variable set constructor

Warning: Rotation matrix must be orthonormal!

Parameters

<i>TireGeometry</i>	Tire ETRTO denomination
<i>RF</i>	Tire ReferenceFrame

6.9.3 Member Function Documentation

6.9.3.1 update()

```
void PatchTire::Shadow::update (
    ETRTO const & TireGeometry,
    ReferenceFrame const & RF )
```

Update the 2D tire shadow domain

Warning: Rotation matrix must be orthonormal!

Parameters

<i>TireGeometry</i>	Tire ETRTO denomination
<i>RF</i>	Tire ReferenceFrame

The documentation for this class was generated from the following file:

- include/PatchTire.hh

6.10 TicToc Class Reference

Public Member Functions

- void **tic** ()
- void **toc** ()
- real_type **elapsed_s** () const
- real_type **elapsed_ms** () const

The documentation for this class was generated from the following file:

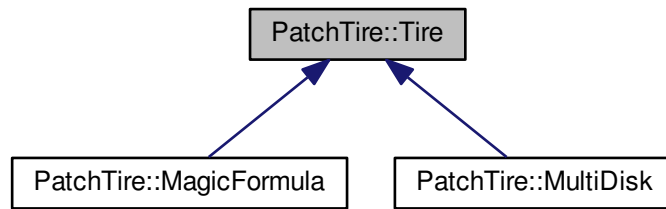
- include/TicToc.hh

6.11 PatchTire::Tire Class Reference

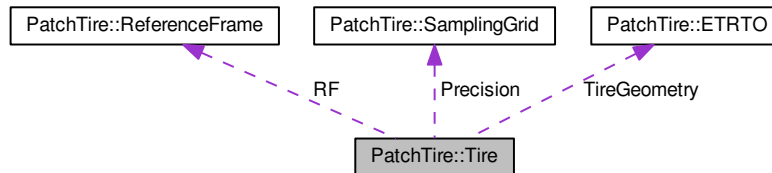
Base class for [Tire](#) models.

```
#include <PatchTire.hh>
```

Inheritance diagram for PatchTire::Tire:



Collaboration diagram for PatchTire::Tire:



Public Member Functions

- `~Tire ()`
Default destructor.
- `Tire (real_type const SectionWidth, real_type const AspectRatio, real_type const RimDiameter, int_type const PointsN, int_type const DisksN)`
Variable set constructor.
- `void printETRTOGeometry (ostream_type &stream) const`
Display Tire ETRTO geometry data.
- `void setReferenceFrame (ReferenceFrame const &_RF)`
- `ReferenceFrame const &getReferenceFrame () const`
Get tire ReferenceFrame object.
- `void setOrigin (vec3 const &Origin)`
Set a new tire origin.
- `void setRotationMatrix (mat3 const &RotationMatrix)`
- `void setTotalTransformationMatrix (mat4 const &TM)`
- `real_type getEulerAngleX () const`
- `real_type getEulerAngleY () const`
- `real_type getEulerAngleZ () const`
- `void getRelativeCamber (real_type &RelativeCamber) const`
Get relative camber angle [rad].
- `int_type getDisksNumber () const`
Dimension of the contact points data structure (disks number)
- `virtual void getRho (real_type &Rho) const =0`

- virtual void `getRho` (`row_vecN` &Rho) const =0
- virtual void `getRhoDot` (`real_type` const &Rho, `real_type` const &Time, `real_type` &RhoDot) const =0
Get contact depth time derivative [m/s].
- virtual void `getRhoDot` (`row_vecN` const &Rho, `real_type` const &Time, `row_vecN` &RhoDot) const =0
Get contact depth time derivative vector [m/s].
- virtual void `getNormal` (`vec3` &Normal) const =0
Get contact normal versor.
- virtual void `getNormal` (`row_vec3` &Normal) const =0
Get contact normal versors vector.
- virtual void `getMFpoint` (`vec3` &Point) const =0
Get Magic Formula contact point.
- virtual void `getMFpoint` (`row_vec3` &Point) const =0
Get Magic Formula contact point vector.
- virtual void `getFriction` (`real_type` &Friction) const =0
Get contact point friction.
- virtual void `getFriction` (`row_vecN` &Friction) const =0
Get contact frictions vector.
- virtual void `getMFpointRF` (`mat4` &PointRF) const =0
Get Magic Formula contact point reference frame with 4x4 transformation matrix.
- virtual void `getMFpointRF` (`row_mat4` &PointRF) const =0
Get Magic Formula contact point reference frame vector with 4x4 transformation matrix.
- virtual void `getArea` (`real_type` &_Area) const =0
*Get approximated contact area on *Disk* plane [m²].*
- virtual void `getArea` (`row_vecN` &Area) const =0
*Get approximated contact areas vector on *Disk* plane [m²].*
- virtual void `getVolume` (`real_type` &Volume) const =0
Get approximated contact volume [m³].
- virtual void `getVolume` (`row_vecN` &_Volume) const =0
Get approximated contact volume [m³].
- virtual void `evaluateContact` (`RDF::TriangleRoad_list` const &TriList)=0
Evaluate contact with RoadTriangles.
- virtual bool `setup` (`RDF::MeshSurface` &Mesh, `mat4` const &TM)=0
Update current tire position and find contact parameters.
- virtual void `print` (`ostream_type` &stream) const =0
Print contact parameters.

Protected Member Functions

- `Tire` (`Tire` const &)=delete
Deleted copy constructor.
- `Tire` const & `operator=` (`Tire` const &)=delete
Deleted copy operator.
- bool `pointSampling` (`RDF::TriangleRoad_list` const &TriList, `vec3` const &RayOrigin, `vec3` const &RayDirection, `vec3` &SampledPt, `real_type` &TriFriction=quietNaN, `vec3` &TriNormal=vec3_↔NaN) const
Perform one point sampling (ray-triangle intersection)

Protected Attributes

- [SamplingGrid Precision](#)
Contact patch evaluating precision.
- [ETRTO TireGeometry](#)
Tire ETRTO denomination.
- [ReferenceFrame RF](#)
ReferenceFrame.

6.11.1 Detailed Description

Base class for [Tire](#) models.

6.11.2 Constructor & Destructor Documentation

6.11.2.1 Tire()

```
PatchTire::Tire::Tire (
    real_type const SectionWidth,
    real_type const AspectRatio,
    real_type const RimDiameter,
    int_type const PointsN,
    int_type const DisksN ) [inline]
```

Variable set constructor.

Parameters

<i>SectionWidth</i>	Tire section width [<i>mm</i>]
<i>AspectRatio</i>	Tire aspect ratio [%]
<i>RimDiameter</i>	Rim diameter [<i>in</i>]
<i>PointsN</i>	Sampling points for each Disk (divisions on <i>X</i> -axis)
<i>DisksN</i>	Number of Disks (divisions on <i>Y</i> -axis -1)

6.11.3 Member Function Documentation

6.11.3.1 evaluateContact()

```
virtual void PatchTire::Tire::evaluateContact (
    RDF::TriangleRoad_list const & TriList ) [pure virtual]
```

Evaluate contact with RoadTriangles.

Parameters

<i>TriList</i>	Shadow/MeshSurface intersected triangles
----------------	--

Implemented in [PatchTire::MagicFormula](#).

6.11.3.2 getArea() [1/2]

```
virtual void PatchTire::Tire::getArea (
    real_type & _Area ) const [pure virtual]
```

Get approximated contact area on [Disk](#) plane [m^2].

Parameters

<code>_Area</code>	Contact area [m^2]
--------------------	------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.3 getArea() [2/2]

```
virtual void PatchTire::Tire::getArea (
    row_vecN & Area ) const [pure virtual]
```

Get approximated contact areas vector on [Disk](#) plane [m^2].

Parameters

<code>Area</code>	Contact areas vector [m^2]
-------------------	--------------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.4 getEulerAngleX()

```
real_type PatchTire::Tire::getEulerAngleX (
    void ) const [inline]
```

Get current Euler angles [rad] for X -axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.11.3.5 getEulerAngleY()

```
real_type PatchTire::Tire::getEulerAngleY (
    void ) const [inline]
```

Get current Euler angles [rad] for Y -axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.11.3.6 getEulerAngleZ()

```
real_type PatchTire::Tire::getEulerAngleZ (
    void ) const [inline]
```

Get current Euler angles [rad] for Z -axis

Warning: Factor as $[R_z][R_x][R_y]!$

6.11.3.7 getFriction() [1/2]

```
virtual void PatchTire::Tire::getFriction (
    real_type & Friction ) const [pure virtual]
```

Get contact point friction.

Parameters

<i>Friction</i>	Contact point friction
-----------------	------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.8 getFriction() [2/2]

```
virtual void PatchTire::Tire::getFriction (
    row_vecN & Friction ) const [pure virtual]
```

Get contact frictions vector.

Parameters

<i>Friction</i>	Contact frictions vector
-----------------	--------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.9 getMFpoint() [1/2]

```
virtual void PatchTire::Tire::getMFpoint (
    vec3 & Point ) const [pure virtual]
```

Get Magic Formula contact point.

Parameters

<i>Point</i>	Magic Formula contact point
--------------	-----------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.10 getMFpoint() [2/2]

```
virtual void PatchTire::Tire::getMFpoint (
    row_vec3 & Point ) const [pure virtual]
```

Get Magic Formula contact point vector.

Parameters

<i>Point</i>	Magic Formula Contact point vector
--------------	------------------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.11 getMFpointRF() [1/2]

```
virtual void PatchTire::Tire::getMFpointRF (
    mat4 & PointRF ) const [pure virtual]
```

Get Magic Formula contact point reference frame with 4x4 transformation matrix.

Parameters

<i>PointRF</i>	Magic Formula contact point reference frame
----------------	---

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.12 getMFpointRF() [2/2]

```
virtual void PatchTire::Tire::getMFpointRF (
    row_mat4 & PointRF ) const [pure virtual]
```

Get Magic Formula contact point reference frame vector with 4x4 transformation matrix.

Parameters

<i>PointRF</i>	Magic Formula ontact point reference frames vector
----------------	--

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.13 getNormal() [1/2]

```
virtual void PatchTire::Tire::getNormal (
    vec3 & Normal ) const [pure virtual]
```

Get contact normal versor.

Parameters

<i>Normal</i>	Contact point normal direction
---------------	--------------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.14 getNormal() [2/2]

```
virtual void PatchTire::Tire::getNormal (
    row_vec3 & Normal ) const [pure virtual]
```

Get contact normal versors vector.

Parameters

<i>Normal</i>	Contact point normal direction vector
---------------	---------------------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.15 getRelativeCamber()

```
void PatchTire::Tire::getRelativeCamber (
    real_type & RelativeCamber ) const
```

Get relative camber angle [rad].

Parameters

<i>RelativeCamber</i>	Relative camber angle
-----------------------	-----------------------

6.11.3.16 getRho() [1/2]

```
virtual void PatchTire::Tire::getRho (
    real_type & Rho ) const [pure virtual]
```

Get contact depth at center point [*m*]

Warning: (if negative the tire does not touch the ground)!

Parameters

<i>Rho</i>	Depth at center point
------------	-----------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.17 getRho() [2/2]

```
virtual void PatchTire::Tire::getRho (
    row_vecN & Rho ) const [pure virtual]
```

Get contact depth vector [*m*]

Warning: (if negative the tire does not touch the ground)!

Parameters

<i>Rho</i>	Depth vector [<i>m</i>]
------------	---------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.18 getRhoDot() [1/2]

```
virtual void PatchTire::Tire::getRhoDot (
    real_type const & Rho,
    real_type const & Time,
    real_type & RhoDot ) const [pure virtual]
```

Get contact depth time derivative [*m/s*].

Parameters

<i>Rho</i>	Previous time step Rho [<i>m</i>]
<i>Time</i>	Time step [<i>s</i>]
<i>RhoDot</i>	Penetration derivative [<i>m/s</i>]

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.19 getRhoDot() [2/2]

```
virtual void PatchTire::Tire::getRhoDot (
    row_vecN const & Rho,
    real_type const & Time,
    row_vecN & RhoDot ) const [pure virtual]
```

Get contact depth time derivative vector [m/s].

Parameters

<i>Rho</i>	Previous time step Rho [m]
<i>Time</i>	Time step [s]
<i>RhoDot</i>	Penetration derivative [m/s]

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.20 getVolume() [1/2]

```
virtual void PatchTire::Tire::getVolume (
    real_type & Volume ) const [pure virtual]
```

Get approximated contact volume [m^3].

Parameters

<i>Volume</i>	Contact volume [m^3]
---------------	--------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.21 getVolume() [2/2]

```
virtual void PatchTire::Tire::getVolume (
    row_vecN & _Volume ) const [pure virtual]
```

Get approximated contact volume [m^3].

Parameters

<i>_Volume</i>	Contact volume vector [m^3]
----------------	---------------------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.22 pointSampling()

```
bool PatchTire::Tire::pointSampling (
    RDF::TriangleRoad_list const & TriList,
    vec3 const & RayOrigin,
    vec3 const & RayDirection,
    vec3 & SampledPt,
    real_type & TriFriction = quietNaN,
    vec3 & TriNormal = vec3_NaN ) const [protected]
```

Perform one point sampling (ray-triangle intersection)

Parameters

<i>TriList</i>	Shadow/MeshSurface intersected triangles
<i>RayOrigin</i>	Ray origin
<i>RayDirection</i>	Ray direction
<i>SampledPt</i>	Intersection point
<i>TriFriction</i>	Intersected triangle friction
<i>TriNormal</i>	Intersected triangle normal

6.11.3.23 print()

```
virtual void PatchTire::Tire::print (
    ostream_type & stream ) const [pure virtual]
```

Print contact parameters.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

6.11.3.24 printETRTOGeometry()

```
void PatchTire::Tire::printETRTOGeometry (
    ostream_type & stream ) const [inline]
```

Display [Tire ETRTO](#) geometry data.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

6.11.3.25 setOrigin()

```
void PatchTire::Tire::setOrigin (
    vec3 const & Origin ) [inline]
```

Set a new tire origin.

Parameters

<i>Origin</i>	Tire origin
---------------	-----------------------------

6.11.3.26 setReferenceFrame()

```
void PatchTire::Tire::setReferenceFrame (
    ReferenceFrame const & _RF ) [inline]
```

Copy the tire [ReferenceFrame](#) object

Warning: Rotation matrix must be orthonormal!

Parameters

<i>_RF</i>	ReferenceFrame object to be copied
------------	--

6.11.3.27 setRotationMatrix()

```
void PatchTire::Tire::setRotationMatrix (
    mat3 const & RotationMatrix ) [inline]
```

Set a new 3x3 rotation matrix

Warning: Rotation matrix must be orthonormal!

Parameters

<i>RotationMatrix</i>	Rotation matrix
-----------------------	-----------------

6.11.3.28 setTotalTransformationMatrix()

```
void PatchTire::Tire::setTotalTransformationMatrix (
    mat4 const & TM ) [inline]
```

Set 4x4 total transformation matrix

Warning: Rotation matrix must be orthonormal!

Parameters

<i>TM</i>	4x4 total transformation matrix
-----------	---------------------------------

6.11.3.29 setup()

```
virtual bool PatchTire::Tire::setup (
    RDF::MeshSurface & Mesh,
    mat4 const & TM ) [pure virtual]
```

Update current tire position and find contact parameters.

Parameters

<i>Mesh</i>	MeshSurface object (road)
<i>TM</i>	4x4 total transformation matrix

Implemented in [PatchTire::MultiDisk](#), and [PatchTire::MagicFormula](#).

The documentation for this class was generated from the following file:

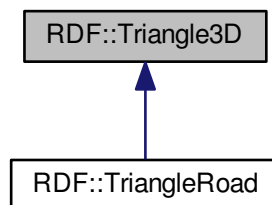
- `include/PatchTire.hh`

6.12 RDF::Triangle3D Class Reference

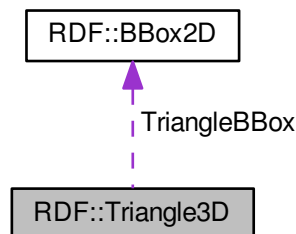
3D triangle (pure geometrical description)

```
#include <RoadRDF.hh>
```

Inheritance diagram for RDF::Triangle3D:



Collaboration diagram for RDF::Triangle3D:



Public Member Functions

- [Triangle3D](#) ()
Variable set constructor.
- [Triangle3D](#) ([vec3](#) const _Vertices[3])
Variable set constructor.
- void [setVertices](#) ([vec3](#) const _Vertices[3])
Set new vertices and update bounding box domain.
- void [setVertices](#) ([vec3](#) const &Vertex0, [vec3](#) const &Vertex1, [vec3](#) const &Vertex2)
Set new vertices then update bounding box domain and normal versor.
- [vec3](#) const & [getNormal](#) (void) const
Get normal versor.
- [vec3](#) const & [getVertex](#) (unsigned i) const

- *Get i -th vertex.*
- `BBox2D` `const & getBBox` (void) `const`
- *Get `Triangle3D` bounding box `BBox2D`.*
- `void print` (`ostream_type` &stream) `const`
- *Print vertices data.*
- `bool intersectRay` (`vec3` `const &RayOrigin`, `vec3` `const &RayDirection`, `vec3` &IntPt) `const`
- `int_type intersectEdgePlane` (`vec3` `const &PlaneN`, `vec3` `const &PlaneP`, `int_type` `const Edge`, `vec3` &IntPt1, `vec3` &IntPt2) `const`
- `bool intersectPlane` (`vec3` `const &PlaneN`, `vec3` `const &PlaneP`, `std::vector`< `vec3` > &IntPts) `const`

Protected Member Functions

- `Triangle3D` (`Triangle3D` `const &`)=delete
- *Deleted copy constructor.*
- `Triangle3D & operator=` (`Triangle3D` `const &`)=delete
- *Deleted copy operator.*

Protected Attributes

- `vec3 Vertices` [3]
- *Vertices reference vector.*
- `vec3 Normal`
- *Triangle normal versor.*
- `BBox2D TriangleBBox`
- *Triangle 2D bounding box (XY plane)*

6.12.1 Detailed Description

3D triangle (pure geometrical description)

6.12.2 Constructor & Destructor Documentation

6.12.2.1 Triangle3D()

```
RDF::Triangle3D::Triangle3D (
    vec3 const _Vertices[3] ) [inline]
```

Variable set constructor.

Parameters

<code>_Vertices</code>	Vertices reference vector
------------------------	---------------------------

6.12.3 Member Function Documentation

6.12.3.1 intersectEdgePlane()

```
int_type RDF::Triangle3D::intersectEdgePlane (
    vec3 const & PlaneN,
```

```

    vec3 const & PlaneN,
    int_type const Edge,
    vec3 & IntPt1,
    vec3 & IntPt2 ) const

```

Check if an edge of the [Triangle3D](#) object hits a and find the intersection point

Parameters

<i>PlaneN</i>	Plane normal vector
<i>PlaneP</i>	Plane known point
<i>Edge</i>	Triangle edge number (0:2)
<i>IntPt1</i>	Intersection point 1
<i>IntPt2</i>	Intersection point 2

6.12.3.2 intersectPlane()

```

bool RDF::Triangle3D::intersectPlane (
    vec3 const & PlaneN,
    vec3 const & PlaneP,
    std::vector< vec3 > & IntPts ) const

```

Check if a plane intersects a [Triangle3D](#) object and find the intersection points

Parameters

<i>PlaneN</i>	Plane normal vector
<i>PlaneP</i>	Plane known point
<i>IntPts</i>	Intersection points

6.12.3.3 intersectRay()

```

bool RDF::Triangle3D::intersectRay (
    vec3 const & RayOrigin,
    vec3 const & RayDirection,
    vec3 & IntPt ) const

```

Check if a ray hits a [Triangle3D](#) object through Möller-Trumbore intersection algorithm

Parameters

<i>RayOrigin</i>	Ray origin position
<i>RayDirection</i>	Ray direction vector
<i>IntPt</i>	Intersection point

6.12.3.4 print()

```

void RDF::Triangle3D::print (
    ostream_type & stream ) const [inline]

```

Print vertices data.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

6.12.3.5 setVertices() [1/2]

```
void RDF::Triangle3D::setVertices (
    vec3 const _Vertices[3] ) [inline]
```

Set new vertices and update bounding box domain.

Parameters

<i>_Vertices</i>	Vertices reference vector
------------------	---------------------------

6.12.3.6 setVertices() [2/2]

```
void RDF::Triangle3D::setVertices (
    vec3 const & Vertex0,
    vec3 const & Vertex1,
    vec3 const & Vertex2 ) [inline]
```

Set new vertices then update bounding box domain and normal versor.

Parameters

<i>Vertex0</i>	Vertex 1
<i>Vertex1</i>	Vertex 2
<i>Vertex2</i>	Vertex 3

The documentation for this class was generated from the following file:

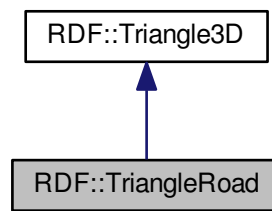
- include/RoadRDF.hh

6.13 RDF::TriangleRoad Class Reference

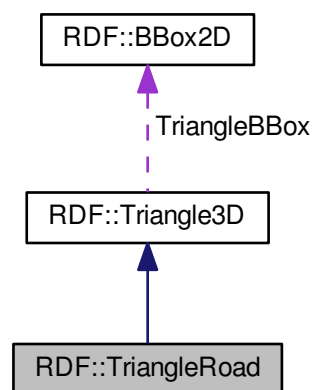
3D triangles for road representation

```
#include <RoadRDF.hh>
```

Inheritance diagram for RDF::TriangleRoad:



Collaboration diagram for RDF::TriangleRoad:



Public Member Functions

- [TriangleRoad](#) ()
Default set constructor.
- [TriangleRoad](#) ([vec3](#) const _Vertices[3], [real_type](#) _Friction)
Variable set constructor.
- void [setFriction](#) ([real_type](#) _Friction)
Set friction coefficient.
- [real_type](#) [getFriction](#) (void) const
Get friction coefficient on the face.
- void [setVertices](#) ([vec3](#) const _Vertices[3])
Set new vertices and update bounding box domain.
- void [setVertices](#) ([vec3](#) const &Vertex0, [vec3](#) const &Vertex1, [vec3](#) const &Vertex2)
Set new vertices then update bounding box domain and normal versor.
- [vec3](#) const & [getNormal](#) (void) const
Get normal versor.

- `vec3` const & `getVertex` (unsigned i) const
Get i -th vertex.
- `BBox2D` const & `getBBox` (void) const
*Get *Triangle3D* bonding box *BBox2D*.*
- void `print` (ostream_type &stream) const
Print vertices data.
- bool `intersectRay` (`vec3` const &RayOrigin, `vec3` const &RayDirection, `vec3` &IntPt) const
- int_type `intersectEdgePlane` (`vec3` const &PlaneN, `vec3` const &PlaneP, int_type const Edge, `vec3` &IntPt1, `vec3` &IntPt2) const
- bool `intersectPlane` (`vec3` const &PlaneN, `vec3` const &PlaneP, std::vector< `vec3` > &IntPts) const

Protected Attributes

- `vec3` `Vertices` [3]
Vertices reference vector.
- `vec3` `Normal`
Triangle normal versor.
- `BBox2D` `TriangleBBox`
*Triangle 2D bounding box (*XY* plane)*

6.13.1 Detailed Description

3D triangles for road representation

6.13.2 Constructor & Destructor Documentation

6.13.2.1 TriangleRoad()

```
RDF::TriangleRoad::TriangleRoad (
    vec3 const _Vertices[3],
    real_type _Friction ) [inline]
```

Variable set constructor.

Parameters

<code>_Vertices</code>	Vertices reference vector
<code>_Friction</code>	Friction coefficient

6.13.3 Member Function Documentation

6.13.3.1 intersectEdgePlane()

```
int_type RDF::Triangle3D::intersectEdgePlane (
    vec3 const & PlaneN,
    vec3 const & PlaneP,
    int_type const Edge,
    vec3 & IntPt1,
    vec3 & IntPt2 ) const [inherited]
```


Check if an edge of the [Triangle3D](#) object hits a and find the intersection point

Parameters

<i>PlaneN</i>	Plane normal vector
<i>PlaneP</i>	Plane known point
<i>Edge</i>	Triangle edge number (0:2)
<i>IntPt1</i>	Intersection point 1
<i>IntPt2</i>	Intersection point 2

6.13.3.2 intersectPlane()

```
bool RDF::Triangle3D::intersectPlane (
    vec3 const & PlaneN,
    vec3 const & PlaneP,
    std::vector< vec3 > & IntPts ) const [inherited]
```

Check if a plane intersects a [Triangle3D](#) object and find the intersection points

Parameters

<i>PlaneN</i>	Plane normal vector
<i>PlaneP</i>	Plane known point
<i>IntPts</i>	Intersection points

6.13.3.3 intersectRay()

```
bool RDF::Triangle3D::intersectRay (
    vec3 const & RayOrigin,
    vec3 const & RayDirection,
    vec3 & IntPt ) const [inherited]
```

Check if a ray hits a [Triangle3D](#) object through Möller-Trumbore intersection algorithm

Parameters

<i>RayOrigin</i>	Ray origin position
<i>RayDirection</i>	Ray direction vector
<i>IntPt</i>	Intersection point

6.13.3.4 print()

```
void RDF::Triangle3D::print (
    ostream_type & stream ) const [inline], [inherited]
```

Print vertices data.

Parameters

<i>stream</i>	Output stream type
---------------	--------------------

6.13.3.5 setFriction()

```
void RDF::TriangleRoad::setFriction (
    real_type _Friction ) [inline]
```

Set friction coefficient.

Parameters

<i>_Friction</i>	New friction coefficient
------------------	--------------------------

6.13.3.6 setVertices() [1/2]

```
void RDF::Triangle3D::setVertices (
    vec3 const _Vertices[3] ) [inline], [inherited]
```

Set new vertices and update bounding box domain.

Parameters

<i>_Vertices</i>	Vertices reference vector
------------------	---------------------------

6.13.3.7 setVertices() [2/2]

```
void RDF::Triangle3D::setVertices (
    vec3 const & Vertex0,
    vec3 const & Vertex1,
    vec3 const & Vertex2 ) [inline], [inherited]
```

Set new vertices then update bounding box domain and normal versor.

Parameters

<i>Vertex0</i>	Vertex 1
<i>Vertex1</i>	Vertex 2
<i>Vertex2</i>	Vertex 3

The documentation for this class was generated from the following file:

- include/RoadRDF.hh

Index

- BBox2D
 - RDF::BBox2D, [20](#)
- contactPlane
 - PatchTire::Disk, [21](#)
- contactTriangles
 - PatchTire::Disk, [22](#)
- Disk
 - PatchTire::Disk, [21](#)
- ETRTO
 - PatchTire::ETRTO, [23](#)
- evaluateContact
 - PatchTire::MagicFormula, [27](#)
 - PatchTire::Tire, [61](#)
- firstToken
 - RDF::algorithms, [15](#)
- fourPointsSampling
 - PatchTire::MagicFormula, [27](#)
- getArea
 - PatchTire::MagicFormula, [28](#)
 - PatchTire::MultiDisk, [42](#)
 - PatchTire::Tire, [61](#), [62](#)
- getDiskFriction
 - PatchTire::MultiDisk, [42](#)
- getDiskMFpoint
 - PatchTire::MultiDisk, [42](#)
- getDiskMFpointRF
 - PatchTire::MultiDisk, [43](#)
- getDiskNormal
 - PatchTire::MultiDisk, [43](#)
- getDiskOriginXYZ
 - PatchTire::MultiDisk, [43](#)
- getDiskRho
 - PatchTire::MultiDisk, [44](#)
- getDiskRhoDot
 - PatchTire::MultiDisk, [44](#)
- getElement
 - RDF::algorithms, [16](#)
- getEulerAngleX
 - PatchTire::MagicFormula, [28](#)
 - PatchTire::MultiDisk, [44](#)
 - PatchTire::ReferenceFrame, [54](#)
 - PatchTire::Tire, [62](#)
- getEulerAngleY
 - PatchTire::MagicFormula, [28](#)
 - PatchTire::MultiDisk, [44](#)
 - PatchTire::ReferenceFrame, [54](#)
 - PatchTire::Tire, [62](#)
- getEulerAngleZ
 - PatchTire::MagicFormula, [28](#)
 - PatchTire::MultiDisk, [45](#)
 - PatchTire::ReferenceFrame, [54](#)
 - PatchTire::Tire, [62](#)
- getFriction
 - PatchTire::MagicFormula, [29](#)
 - PatchTire::MultiDisk, [45](#)
 - PatchTire::Tire, [62](#), [63](#)
- getMFpoint
 - PatchTire::MagicFormula, [29](#)
 - PatchTire::MultiDisk, [45](#)
 - PatchTire::Tire, [63](#)
- getMFpointRF
 - PatchTire::MagicFormula, [30](#)
 - PatchTire::MultiDisk, [46](#)
 - PatchTire::Tire, [63](#), [64](#)
- getNormal
 - PatchTire::MagicFormula, [30](#)
 - PatchTire::MultiDisk, [46](#)
 - PatchTire::Tire, [64](#)
- getRelativeCamber
 - PatchTire::MagicFormula, [31](#)
 - PatchTire::MultiDisk, [47](#)
 - PatchTire::Tire, [64](#)
- getRho
 - PatchTire::MagicFormula, [31](#)
 - PatchTire::MultiDisk, [47](#)
 - PatchTire::Tire, [65](#)
- getRhoDot
 - PatchTire::MagicFormula, [31](#), [32](#)
 - PatchTire::MultiDisk, [47](#), [48](#)
 - PatchTire::Tire, [65](#)
- getVolume
 - PatchTire::MagicFormula, [32](#)
 - PatchTire::MultiDisk, [48](#)
 - PatchTire::Tire, [66](#)
- intersectAABBtree
 - RDF::MeshSurface, [36](#)
- intersectBBox
 - RDF::MeshSurface, [36](#)
- intersectEdgePlane
 - RDF::Triangle3D, [70](#)
 - RDF::TriangleRoad, [75](#)
- intersectPlane
 - RDF::Triangle3D, [71](#)
 - RDF::TriangleRoad, [76](#)

- intersectPointSegment
 - PatchTire::algorithms, 12
- intersectRay
 - RDF::Triangle3D, 71
 - RDF::TriangleRoad, 76
- intersectRayPlane
 - PatchTire::algorithms, 12
- LoadFile
 - RDF::MeshSurface, 37
- MagicFormula
 - PatchTire::MagicFormula, 27
- MeshSurface
 - RDF::MeshSurface, 36
- minmax_XY
 - PatchTire::algorithms, 13
- MultiDisk
 - PatchTire::MultiDisk, 40, 41
- PatchTire, 11
- PatchTire::Disk, 20
 - contactPlane, 21
 - contactTriangles, 22
 - Disk, 21
 - set, 22
 - setOriginXZ, 22
- PatchTire::ETRTO, 23
 - ETRTO, 23
 - print, 24
- PatchTire::MagicFormula, 24
 - evaluateContact, 27
 - fourPointsSampling, 27
 - getArea, 28
 - getEulerAngleX, 28
 - getEulerAngleY, 28
 - getEulerAngleZ, 28
 - getFriction, 29
 - getMFpoint, 29
 - getMFpointRF, 30
 - getNormal, 30
 - getRelativeCamber, 31
 - getRho, 31
 - getRhoDot, 31, 32
 - getVolume, 32
 - MagicFormula, 27
 - pointSampling, 33
 - print, 33
 - printETRTOGeometry, 33
 - setOrigin, 34
 - setReferenceFrame, 34
 - setRotationMatrix, 34
 - setTotalTransformationMatrix, 34
 - setup, 34
- PatchTire::MultiDisk, 37
 - getArea, 42
 - getDiskFriction, 42
 - getDiskMFpoint, 42
 - getDiskMFpointRF, 43
 - getDiskNormal, 43
 - getDiskOriginXYZ, 43
 - getDiskRho, 44
 - getDiskRhoDot, 44
 - getEulerAngleX, 44
 - getEulerAngleY, 44
 - getEulerAngleZ, 45
 - getFriction, 45
 - getMFpoint, 45
 - getMFpointRF, 46
 - getNormal, 46
 - getRelativeCamber, 47
 - getRho, 47
 - getRhoDot, 47, 48
 - getVolume, 48
 - MultiDisk, 40, 41
 - pointSampling, 50
 - print, 50
 - printETRTOGeometry, 50
 - setDiskOriginXZ, 51
 - setOrigin, 51
 - setReferenceFrame, 51
 - setRotationMatrix, 51
 - setTotalTransformationMatrix, 52
 - setup, 52
- PatchTire::ReferenceFrame, 52
 - getEulerAngleX, 54
 - getEulerAngleY, 54
 - getEulerAngleZ, 54
 - ReferenceFrame, 53
 - set, 54
 - setOrigin, 54
 - setRotationMatrix, 54
 - setTotalTransformationMatrix, 55
- PatchTire::SamplingGrid, 55
 - SamplingGrid, 56
 - set, 56
 - setSwitchNumber, 57
- PatchTire::Shadow, 57
 - Shadow, 57
 - update, 58
- PatchTire::Tire, 58
 - evaluateContact, 61
 - getArea, 61, 62
 - getEulerAngleX, 62
 - getEulerAngleY, 62
 - getEulerAngleZ, 62
 - getFriction, 62, 63
 - getMFpoint, 63
 - getMFpointRF, 63, 64
 - getNormal, 64
 - getRelativeCamber, 64
 - getRho, 65
 - getRhoDot, 65
 - getVolume, 66
 - pointSampling, 66
 - print, 67
 - printETRTOGeometry, 67

- setOrigin, [67](#)
 - setReferenceFrame, [67](#)
 - setRotationMatrix, [68](#)
 - setTotalTransformationMatrix, [68](#)
 - setup, [68](#)
 - Tire, [61](#)
- PatchTire::algorithms, [11](#)
 - intersectPointSegment, [12](#)
 - intersectRayPlane, [12](#)
 - minmax_XY, [13](#)
 - trapezoidArea, [13](#)
 - weightedMean, [14](#)
- pointSampling
 - PatchTire::MagicFormula, [33](#)
 - PatchTire::MultiDisk, [50](#)
 - PatchTire::Tire, [66](#)
- print
 - PatchTire::ETRTO, [24](#)
 - PatchTire::MagicFormula, [33](#)
 - PatchTire::MultiDisk, [50](#)
 - PatchTire::Tire, [67](#)
 - RDF::BBox2D, [20](#)
 - RDF::Triangle3D, [71](#)
 - RDF::TriangleRoad, [76](#)
- printData
 - RDF::MeshSurface, [37](#)
- printETRTOGeometry
 - PatchTire::MagicFormula, [33](#)
 - PatchTire::MultiDisk, [50](#)
 - PatchTire::Tire, [67](#)
- RDF::BBox2D, [19](#)
 - BBox2D, [20](#)
 - print, [20](#)
 - updateBBox2D, [20](#)
- RDF::MeshSurface, [35](#)
 - intersectAABBtree, [36](#)
 - intersectBBox, [36](#)
 - LoadFile, [37](#)
 - MeshSurface, [36](#)
 - printData, [37](#)
 - set, [37](#)
- RDF::Triangle3D, [69](#)
 - intersectEdgePlane, [70](#)
 - intersectPlane, [71](#)
 - intersectRay, [71](#)
 - print, [71](#)
 - setVertices, [73](#)
 - Triangle3D, [70](#)
- RDF::TriangleRoad, [73](#)
 - intersectEdgePlane, [75](#)
 - intersectPlane, [76](#)
 - intersectRay, [76](#)
 - print, [76](#)
 - setFriction, [77](#)
 - setVertices, [77](#)
 - TriangleRoad, [75](#)
- RDF::algorithms, [15](#)
 - firstToken, [15](#)
 - getElement, [16](#)
 - split, [16](#)
 - tail, [16](#)
- RDF, [14](#)
- ReferenceFrame
 - PatchTire::ReferenceFrame, [53](#)
- SamplingGrid
 - PatchTire::SamplingGrid, [56](#)
- set
 - PatchTire::Disk, [22](#)
 - PatchTire::ReferenceFrame, [54](#)
 - PatchTire::SamplingGrid, [56](#)
 - RDF::MeshSurface, [37](#)
- setDiskOriginXZ
 - PatchTire::MultiDisk, [51](#)
- setFriction
 - RDF::TriangleRoad, [77](#)
- setOrigin
 - PatchTire::MagicFormula, [34](#)
 - PatchTire::MultiDisk, [51](#)
 - PatchTire::ReferenceFrame, [54](#)
 - PatchTire::Tire, [67](#)
- setOriginXZ
 - PatchTire::Disk, [22](#)
- setReferenceFrame
 - PatchTire::MagicFormula, [34](#)
 - PatchTire::MultiDisk, [51](#)
 - PatchTire::Tire, [67](#)
- setRotationMatrix
 - PatchTire::MagicFormula, [34](#)
 - PatchTire::MultiDisk, [51](#)
 - PatchTire::ReferenceFrame, [54](#)
 - PatchTire::Tire, [68](#)
- setSwitchNumber
 - PatchTire::SamplingGrid, [57](#)
- setTotalTransformationMatrix
 - PatchTire::MagicFormula, [34](#)
 - PatchTire::MultiDisk, [52](#)
 - PatchTire::ReferenceFrame, [55](#)
 - PatchTire::Tire, [68](#)
- setVertices
 - RDF::Triangle3D, [73](#)
 - RDF::TriangleRoad, [77](#)
- setup
 - PatchTire::MagicFormula, [34](#)
 - PatchTire::MultiDisk, [52](#)
 - PatchTire::Tire, [68](#)
- Shadow
 - PatchTire::Shadow, [57](#)
- split
 - RDF::algorithms, [16](#)
- tail
 - RDF::algorithms, [16](#)
- TicToc, [58](#)
- Tire
 - PatchTire::Tire, [61](#)
- TireGround, [16](#)

- trapezoidArea
 - PatchTire::algorithms, [13](#)
- Triangle3D
 - RDF::Triangle3D, [70](#)
- TriangleRoad
 - RDF::TriangleRoad, [75](#)
- update
 - PatchTire::Shadow, [58](#)
- updateBBox2D
 - RDF::BBox2D, [20](#)
- weightedMean
 - PatchTire::algorithms, [14](#)

C.1 Test 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     TireGround::vec3 V1[3];
18     V1[0] = TireGround::vec3(1.0, 0.0, 0.0);
19     V1[1] = TireGround::vec3(0.0, 1.0, 0.0);
20     V1[2] = TireGround::vec3(-1.0, 0.0, 0.0);
21
22     TireGround::vec3 V2[3];
23     V2[0] = TireGround::vec3(-1.0, 0.0, 0.0);
24     V2[1] = TireGround::vec3(0.0, -1.0, 0.0);
25     V2[2] = TireGround::vec3(1.0, 0.0, 0.0);
26
27     // Initialize generic Triangle3D
28     TireGround::RDF::TriangleRoad Triangle1(V1, 0.0);
29     TireGround::RDF::TriangleRoad Triangle2(V2, 0.0);
30
31     // Initialize rotation matrix
32     TireGround::mat3 Rot_X;
33 }
```

```
34 // Initialize intersection point
35 TireGround::vec3 IntersectionPointTri1, IntersectionPointTri2;
36 bool IntersectionBoolTri1, IntersectionBoolTri2;
37
38 // Initialize Ray
39 TireGround::vec3 RayOrigin = TireGround::vec3(0.0, 0.0, 0.0);
40 TireGround::vec3 RayDirection = TireGround::vec3(0.0, 0.0, -1.0);
41
42 // Perform intersection at 0.5° step
43 for ( TireGround::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     TireGround::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 TireGround::vec3 N1 = Triangle1.getNormal();
86 TireGround::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\nGEOMETRY TEST 1: Completed\n";
91
92 // Exit the program
93 return 0;
94 }
```

C.1.2 Geometry-test2.cc

```

1 // GEOMETRY TEST 2 - SEGMENT CIRCLE INTERSECTION
2
3 #include <fstream>    // fStream - STD File I/O Library
4 #include <iostream>   // Iostream - STD I/O Library
5 #include <string>     // String - STD String Library
6
7 #include "PatchTire.hh" // Tire Data Processing
8 #include "RoadRDF.hh"  // Tire Data Processing
9
10 // Main function
11 int
12 main() {
13     // Initialize disk
14     TireGround::Disk NewDisk(TireGround::vec2(0.0, 0.0), 0.0, 1.0);
15
16     // Initialize segments points
17     TireGround::vec2 SegIn1PtA = TireGround::vec2(0.0, 0.0);
18     TireGround::vec2 SegIn1PtB = TireGround::vec2(0.0, 1.0);
19
20     TireGround::vec2 SegIn2PtA = TireGround::vec2(-2.0, 0.0);
21     TireGround::vec2 SegIn2PtB = TireGround::vec2(2.0, 0.0);
22
23     TireGround::vec2 SegOutPtA = TireGround::vec2(1.0, 2.0);
24     TireGround::vec2 SegOutPtB = TireGround::vec2(-1.0, 2.0);
25
26     TireGround::vec2 SegTangPtA = TireGround::vec2(1.0, 1.0);
27     TireGround::vec2 SegTangPtB = TireGround::vec2(-1.0, 1.0);
28
29     // Initialize intersection points and output types
30     TireGround::vec2 IntSegIn1_1, IntSegIn1_2, IntSegIn2_1, IntSegIn2_2, IntSegOut_1,
31         IntSegOut_2, IntSegTang_1, IntSegTang_2;
32     TireGround::int_type PtIn1, PtIn2, PtOut, PtTang;
33
34     // Calculate intersections
35     PtIn1 = NewDisk.intersectSegment(
36         SegIn1PtA, SegIn1PtB, IntSegIn1_1, IntSegIn1_2
37     );
38     PtIn2 = NewDisk.intersectSegment(
39         SegIn2PtA, SegIn2PtB, IntSegIn2_1, IntSegIn2_2
40     );
41     PtOut = NewDisk.intersectSegment(
42         SegOutPtA, SegOutPtB, IntSegOut_1, IntSegOut_2
43     );
44     PtTang = NewDisk.intersectSegment(
45         SegTangPtA, SegTangPtB, IntSegTang_1, IntSegTang_2
46     );
47
48     // Display results
49     std::cout
50         << "GEOMETRY TEST 2 - SEGMENT DISK INTERSECTION\n\n"
51         << "Radius = " << NewDisk.getRadius() << std::endl
52         << "Origin = [" << NewDisk.getOriginXZ().x() << ", " << NewDisk.getOriginXZ().y() << "]\n"
53         << std::endl
54         << "Segment 1 with two intersections -> " << PtIn1 << " intersections found\n"
55         << "Segment Point A\t= [" << SegIn1PtA.x() << ", " << SegIn1PtA.y() << "]\n"
56         << "Segment Point B\t= [" << SegIn1PtB.x() << ", " << SegIn1PtB.y() << "]\n"
57         << "Intersection Point 1\t= [" << IntSegIn1_1.x() << ", " << IntSegIn1_1.y() << "]\n"
58         << "Intersection Point 2\t= [" << IntSegIn1_2.x() << ", " << IntSegIn1_2.y() << "]\n"
59         << std::endl
60         << "Segment 2 with two intersections -> " << PtIn2 << " intersections found\n"
61         << "Segment Point A\t= [" << SegIn2PtA.x() << ", " << SegIn2PtA.y() << "]\n"
62         << "Segment Point B\t= [" << SegIn2PtB.x() << ", " << SegIn2PtB.y() << "]\n"
63         << "Intersection Point 1\t= [" << IntSegIn2_1.x() << ", " << IntSegIn2_1.y() << "]\n"
64         << "Intersection Point 2\t= [" << IntSegIn2_2.x() << ", " << IntSegIn2_2.y() << "]\n"
65         << std::endl

```

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

C.1.3 Geometry-test3.cc

```
1 // GEOMETRY TEST 3 - POINT INSIDE CIRCLE
2
3 #include <fstream> // fStream - STD File I/O Library
4 #include <iostream> // Iostream - STD I/O Library
5 #include <string> // String - STD String Library
6
7 #include "PatchTire.hh" // Tire Data Processing
8 #include "RoadRDF.hh" // Tire Data Processing
9
10 // Main function
11 int
12 main() {
13 // Initialize disk
14 TireGround::Disk NewDisk(TireGround::vec2(0.0, 0.0), 0.0, 1.0);
15
16 // Query points and intersection bools
17 TireGround::vec2 PointIn = TireGround::vec2(0.0, 0.0);
18 TireGround::vec2 PointOut = TireGround::vec2(2.0, 0.0);
19 TireGround::vec2 PointBorder = TireGround::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     TireGround::vec2 PointA = TireGround::vec2(0.0, 0.0);
15     TireGround::vec2 PointB = TireGround::vec2(1.0, 1.0);
16
17     // Query points and intersection bools
18     TireGround::vec2 PointIn    = TireGround::vec2(0.5, 0.5);
19     TireGround::vec2 PointOut   = TireGround::vec2(-1.0, -1.0);
20     TireGround::vec2 PointBorder = TireGround::vec2(1.0, 1.0);
21
22     // Calculate intersection
23     bool PtInBool  = TireGround::algorithms::intersectPointSegment(PointA, PointB, PointIn);
24     bool PtOutBool = TireGround::algorithms::intersectPointSegment(PointA, PointB, PointOut);
25     bool PtBordBool = TireGround::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 a singolo disco

C.2.1 MagicFormula-test1.cc

```
1 // PATCH EVALUATION TEST 1 - LOAD THE DATA FROM THE RDF FILE THEN PRINT IT INTO
2 // A FILE Out.txt. THEN CHARGE THE TIRE DATA AND ASSOCIATE THE CURRENT MESH TO
3 // IT.
4
5 #include <chrono>      // chrono - STD Time Measurement Library
6 #include <fstream>     // fStream - STD File I/O Library
7 #include <iostream>    // Iostream - STD I/O Library
8
9 #include "PatchTire.hh" // Tire Data Processing
10 #include "RoadRDF.hh"  // Tire Data Processing
11 #include "TicToc.hh"   // Processing Time Library
12
13 // Main function
14 int
15 main() {
16
17     try {
18
19         // Instantiate a TicToc object
20         TicToc tictoc;
21
22         std::cout
23             << "MAGIC FORMULA TIRE TEST 1 - CHECK INTERSECTION ON UNKNOWN MESH.\n\n";
24
25         // Load .rdf File
26         TireGround::RDF::MeshSurface Road("./RDF_files/Town04.rdf");
27
28         // Print OutMesh.txt file
29         // Road.printData("OutMesh.txt");
30
31         // Initialize the Magic Formula Tire
32         TireGround::Tire* TireSD = new TireGround::MagicFormula(0.250, 55, 11, 10);
33
34         // Orient the tire in the space
35         TireGround::real_type Yaw = 0*G2lib::m_pi;
36         TireGround::real_type Camber = 0*G2lib::m_pi;
37
38         // Transformation matrix for X and Z-axis rotation
39         TireGround::mat3 Rot_Z;
40         Rot_Z << cos(Yaw), -sin(Yaw), 0,
41                 sin(Yaw),  cos(Yaw), 0,
42                 0,         0, 1;
43         TireGround::mat3 Rot_X;
44         Rot_X << 1, 0, 0,
45                 0, cos(Camber), -sin(Camber),
46                 0, sin(Camber),  cos(Camber);
47         // Update Rotation Matrix
48         TireGround::mat3 RotMat = Rot_Z * Rot_X;
49
50         TireGround::vec3 Origin(-400.0, -20.0, 0.237); //0.8, 19.0, 0.26
51         TireGround::ReferenceFrame Pose(Origin, RotMat);
52
53         TireGround::mat4 RotM;
54
55         RotM << 0.999982, -0.126237, -0.00255133, -398.800,
56                 0.126237, 0.999982, -3.50917e-05, -19.1698,
57                 0.00255571, -0.000286982, 1.00000, 0.273023,
58                 0.00000, 0.00000, 0.00000, 1.00000;
```

```

60 // Start chronometer
61 tictoc.tic();
62
63 // Set an orientation and calculate parameters (true = print results)
64 bool Out = TireSD->setup( Road, RotM); //Pose.getTotalTransformationMatrix() );
65
66 // Stop chronometer
67 tictoc.toc();
68
69 // Display current tire data on command line
70 if (Out) TireSD->print(std::cout);
71
72 // This constructs a duration object using milliseconds
73 std::cout
74     << "Execution time = " << tictoc.elapsed_ms() << " ms\n"
75     << "\nCheck the results...\n"
76     << "\nMAGIC FORMULA TIRE TEST 1: Completed\n\n";
77
78 } catch ( std::exception const & exc ) {
79     std::cerr << exc.what() << '\n';
80 }
81 catch (...) {
82     std::cerr << "Unknown error\n";
83 }
84 }

```

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         TireGround::vec3 Vertices[3];
25         Vertices[0] = TireGround::vec3(100.0, 0.0, 1.0);
26         Vertices[1] = TireGround::vec3(0.0, 100.0, 0.0);
27         Vertices[2] = TireGround::vec3(0.0, -100.0, 0.0);
28         TireGround::RDF::TriangleRoad_list PtrTriangleVec;
29         PtrTriangleVec.push_back(
30             TireGround::RDF::TriangleRoad_ptr( new TireGround::RDF::TriangleRoad(Vertices, 1.0) ) );
31
32         // Build the mesh
33         TireGround::RDF::MeshSurface Road(PtrTriangleVec);
34
35         // Initialize the Magic Formula Tire
36         TireGround::Tire* TireSD = new TireGround::MagicFormula(0.205, 60, 15, 0);

```

```
37
38 // Orient the tire in the space
39 TireGround::real_type Yaw = 0*G2lib::m_pi;
40 TireGround::real_type Camber = 0*G2lib::m_pi;
41
42 // Transformation matrix for X and Z-axis rotation
43 TireGround::mat3 Rot_Z;
44 Rot_Z << cos(Yaw), -sin(Yaw), 0,
45           sin(Yaw), cos(Yaw), 0,
46           0, 0, 1;
47 TireGround::mat3 Rot_X;
48 Rot_X << 1, 0, 0,
49           0, cos(Camber), -sin(Camber),
50           0, sin(Camber), cos(Camber);
51 // Update Rotation Matrix
52 TireGround::mat3 RotMat = Rot_Z * Rot_X;
53
54 TireGround::vec3 Origin( 50.0, 10.0, 0.26+0.5 );
55 TireGround::ReferenceFrame Pose(Origin, RotMat);
56
57 // Start chronometer
58 tictoc.tic();
59
60 // Set an orientation and calculate parameters (true = print results)
61 bool Out = TireSD->setup( Road, Pose.getTotalTransformationMatrix() );
62
63 // Stop chronometer
64 tictoc.toc();
65
66 // Display current tire data on command line
67 if (Out) TireSD->print(std::cout);
68
69 // This constructs a duration object using milliseconds
70 std::cout
71 << "Execution time = " << tictoc.elapsed_ms() << " ms\n"
72 << "\nCheck the results...\n"
73 << "\nMAGIC FORMULA TIRE TEST 2: Completed\n";
74
75 } catch ( std::exception const & exc ) {
76     std::cerr << exc.what() << '\n';
77 }
78 catch (...) {
79     std::cerr << "Unknown error\n";
80 }
81 }
```

C.3 Tests per il modello a più dischi

C.3.1 MultiDisk-test1.cc

```
1 // PATCH EVALUATION TEST 1 - LOAD THE DATA FROM THE RDF FILE THEN PRINT IT INTO
2 // A FILE Out.txt. THEN CHARGE THE TIRE DATA AND ASSOCIATE THE CURRENT MESH TO
3 // IT.
4
5 #include <chrono> // chrono - STD Time Measurement Library
6 #include <fstream> // fStream - STD File I/O Library
7 #include <iostream> // Iostream - STD I/O Library
8
9 #include "PatchTire.hh" // Tire Data Processing
10 #include "RoadRDF.hh" // Tire Data Processing
11 #include "TicToc.hh" // Processing Time Library
12
```



```

13 // Main function
14 int
15 main() {
16
17     try {
18
19         // Instantiate a TicToc object
20         TicToc tictoc;
21
22         std::cout
23             << "MULTIDISK TIRE TEST 1 - CHECK INTERSECTION ON UNKNOWN MESH.\n\n";
24
25         // Load .rdf File
26         TireGround::RDF::MeshSurface Road("./RDF_files/Town04.rdf");
27
28         // Print OutMesh.txt file
29         // Road.printData("OutMesh.txt");
30
31         // Initialize the MultiDisk Tire
32         TireGround::Tire* TireMD = new TireGround::MultiDisk(0.250, 55, 11, 10, 20 , 10);
33
34         // Orient the tire in the space
35         TireGround::real_type Yaw    = 0*M_PI;
36         TireGround::real_type Camber = 0*M_PI;
37
38         // Transformation matrix for X and Z-axis rotation
39         TireGround::mat3 Rot_Z;
40         Rot_Z << cos(Yaw), -sin(Yaw), 0,
41                 sin(Yaw),  cos(Yaw), 0,
42                 0,         0, 1;
43         TireGround::mat3 Rot_X;
44         Rot_X << 1, 0, 0,
45                 0, cos(Camber), -sin(Camber),
46                 0, sin(Camber),  cos(Camber);
47         // Update Rotation Matrix
48         TireGround::mat3 RotMat = Rot_Z * Rot_X;
49
50         TireGround::vec3 Origin(-400, -20.0, 0.237); //0.8, 19.0, 0.26
51         TireGround::ReferenceFrame Pose(Origin, RotMat);
52
53         TireGround::mat4 RotM;
54
55         RotM << 0.999982, -0.126237, -0.00255133, -398.800,
56                 0.126237,  0.999982, -3.50917e-05, -19.1698,
57                 0.00255571, -0.000286982, 1.00000, 0.273023,
58                 0.00000, 0.00000, 0.00000, 1.00000;
59
60         // Start chronometer
61         tictoc.tic();
62
63         // Set an orientation and calculate parameters (true = print results)
64         bool Out = TireMD->setup( Road, RotM); //Pose.getTotalTransformationMatrix() );
65
66         // Stop chronometer
67         tictoc.toc();
68
69         // Display current tire data on command line
70         if (Out) TireMD->print(std::cout);
71
72         // This constructs a duration object using milliseconds
73         std::cout
74             << "Execution time = " << tictoc.elapsed_ms() << " ms\n"
75             << "\nCheck the results...\n"
76             << "\nMULTIDISK TIRE TEST 1: Completed\n\n";
77

```

```
78 } catch ( std::exception const & exc ) {
79     std::cerr << exc.what() << '\n';
80 }
81 catch (...) {
82     std::cerr << "Unknown error\n";
83 }
84 }
```

C.3.2 MultiDisk-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             << "MULTIDISK TIRE TEST 2 - CHECK INTERSECTION ON KNOWN MESH.\n\n";
22
23         // Initialize a quite big triangle
24         TireGround::vec3 Vertices[3];
25         Vertices[0] = TireGround::vec3(100.0, 0.0, 1.0);
26         Vertices[1] = TireGround::vec3(0.0, 100.0, 0.0);
27         Vertices[2] = TireGround::vec3(0.0, -100.0, 0.0);
28         TireGround::RDF::TriangleRoad_list PtrTriangleVec;
29         PtrTriangleVec.push_back(
30             TireGround::RDF::TriangleRoad_ptr( new TireGround::RDF::TriangleRoad(Vertices, 1.0) ) );
31
32         // Build the mesh
33         TireGround::RDF::MeshSurface Road(PtrTriangleVec);
34
35         // Initialize the Magic Formula Tire
36         TireGround::Tire* TireMD = new TireGround::MultiDisk(0.205, 60, 15, 5, 5, 10);
37
38         // Orient the tire in the space
39         TireGround::real_type Yaw = 0*G2lib::m_pi;
40         TireGround::real_type Camber = 0*G2lib::m_pi;
41
42         // Transformation matrix for X and Z-axis rotation
43         TireGround::mat3 Rot_Z;
44         Rot_Z << cos(Yaw), -sin(Yaw), 0,
45                 sin(Yaw),  cos(Yaw), 0,
46                 0,         0, 1;
47         TireGround::mat3 Rot_X;
48         Rot_X << 1, 0, 0,
49                 0, cos(Camber), -sin(Camber),
50                 0, sin(Camber),  cos(Camber);
51         // Update Rotation Matrix
52         TireGround::mat3 RotMat = Rot_Z * Rot_X;
53
54         TireGround::vec3 Origin( 50.0, 10.0, 0.26+0.5 );
```

```
55     TireGround::ReferenceFrame Pose(Origin, RotMat);
56
57     // Start chronometer
58     tictoc.tic();
59
60     // Set an orientation and calculate parameters (true = print results)
61     bool Out = TireMD->setup( Road, Pose.getTotalTransformationMatrix() );
62
63     // Stop chronometer
64     tictoc.toc();
65
66     // Display current tire data on command line
67     if (Out) TireMD->print(std::cout);
68
69     // This constructs a duration object using milliseconds
70     std::cout
71         << "Execution time = " << tictoc.elapsed_ms() << " ms\n"
72         << "\nCheck the results...\n"
73         << "\nMULTIDISK TIRE TEST 2: Completed\n";
74
75 } catch ( std::exception const & exc ) {
76     std::cerr << exc.what() << '\n';
77 }
78 catch (...) {
79     std::cerr << "Unknown error\n";
80 }
81 }
```


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.