Universitá degli Studi di Trento

Dipartimento di Ingegneria Industriale

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA MECCATRONICA

Tesi di Laurea

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

Laureando:
Davide Stocco

Relatore:
Prof. Enrico Bertolazzi

Anno Accademico 2019 · 2020

Sommario

This dissertation details …

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli acronimi

Introduzione 1

1.1 Obiettivi

La motivazione di questa tesi sta nella trovata collaborazione tra il *Dipartimento di Ingegneria Industriale* dell'Università di Trento e *AnteMotion S.r.l.*, azienda specializzata in realtà virtuale e simulazione di veicoli multibody. In particolare il modello di veicolo e pnumatico precedentemente studiati da Larcher in [4] sarà integrato nel simulatore di guida in tempo reale di AnteMotion. Pertanto, lo sviluppo del modello è stato finalizzata a minimizzare i tempi di compilazione massimizzando l'accuratezza del modello. La necessità di sviluppare un algoritmo che calcoli i parametri dell'interazione tra terreno e pneumatico 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 reale. Ciò comporta diversi vantaggi in termini di tempo, costi e sicurezza. Infatti, il modello digitale può essere facilmente riconfigurato e analizzato, mentre questo è solitamente impossibile o troppo oneroso del punto di vista di tempi e costi da fare con il sistema reale [5]. Al giorno d'oggi esistono numerosi modelli di veicolo

e pneumatico, certamente, più semplice è il modello, più veloce è la risoluzione delle equazioni costituenti e, a seconda delle applicazioni, deve essere scelta la giusta complessità per il modello. Per la maggior parte delle applicazioni di guida autonoma, un modello semplice è sufficiente per caratterizzare con un livello di dettaglio sufficiente il comportamento del veicolo, e poiché queste analisi sono molto spesso fatte con l'ausilio di *Hardware in the Loop* (HIL), il modello dinamico del veicolo deve essere risolto in tempo reale con tipico passo di tempo di 1 millisecondo. Il vincolo in tempo reale implica un modello di veicolo di calcolo veloce, ciò significa che i modelli semplici con pochi parametri, di solito modelli lineari a singola traccia, 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 si sposta nella regione non lineare e i fenomeni transitori non sono più trascurabili. Ciò significa che un modello più dettagliato di quello utilizzato per la guida in condizioni "standard". L'accuratezza dinamica del modello è di grande importanza per ricavare previsioni realistiche delle prestazioni del veicolo e del sistema di controllo. È importante notare che modellare in modo esaustivo tutti i sistemi di un'auto sarebbe un compito estremamente arduo e a volte anche impossibile. Esistono quindi modelli empirici come il modello della *MagicFormula* di Hans Pacejka e il modello *Fiala* che cercano di imitare il reale comportamento del sistema. Il calcolo dei parametri di questo tipo di modelli richiede l'interpolazione di un set di dati di grandi dimensioni, e può quindi essere numericamente inefficiente o comunque troppo oneroso in termini di tempo.

Per studiare il comportamento del sistema in diversi scenari di guida, i moderni strumenti di simulazione spesso richiedono una grande quantità di di dati e l'unico modo per ottenerli è esecuzione stessa di migliaia di simulazioni. A questo proposito è quindi necessario un conducente virtuale o artificiale in grado di controllare il veicolo fino ai limiti di guidabilità. Pertanto, la caratteristica principale di un driver artificiale o virtuale è la capacità di guidare una varietà di veicoli con diverse caratteristiche dinamiche. Indipendentemente dall'architettura e dalla metodologia utilizzata per sviluppare il conducente artificiale, deve utilizzare una sorta di modello dinamico che rappresenta il comportamento del veicolo controllato. Si tratta dunque di una sorta di modello di comportamento dinamico del veicolo.

Lo scopo di questo lavoro si collega a quello già svolto da Larcher in [4], dove grazie

a un modello di veicolo completo con 14 gradi di libertà ha fornito un modello in grado di catturare con un livello di dettaglio appropriato il comportamento del veicolo quando viene spinto ai suoi limiti di maneggevolezza. La necessità di calcolare in tempo reale i parametri di input per il modello di ruota scelto da [4] definisce l'obiettivo di questo lavoro. Ovvero di avere una libreria scritta in `C++`, che con alcuni semplici parametri in input come la denominazione *European Tyre and Rim Technical Organisation* (ETRTO) dello pneumatico e la posizione nello spazio, calcola i dati relativi all'interazione pneumatico strada quali l'intersezione del punto sotto il centro ruota, l'area di contatto, e l'inclinazione locale del piano strada. Il tutto cercando di minimizzare i tempi di compilazione. [6] [2] [10] [9]

Il Pneumatico 2

2.1 Una Breve Introduzione

Gli pneumatici sono probabilmente i componenti più complessi di un'auto in quanto combinano decine di componenti che devono essere formati, assemblati e curati insieme. Il loro successo finale dipende dalla loro capacità di fondere tutti i componenti separati in un prodotto coeso che soddisfa le esigenze del conducente [8]. Gli pneumatici sono caratterizzati da un comportamento altamente non lineare con una dipendenza da diversi fattori costruttivi e ambientali. Tuttavia, le forze di contatto possono essere descritte completamente da un vettore di forza risultante applicato in un punto specifico della patch di contatto e da un vettore di coppia, come illustrato nella Figura??. Componenti cruciali per la movimentazione dei veicoli e il comportamento di guida, le forze degli pneumatici richiedono particolare attenzione e cura soprattutto quando, lungo il comportamento stazionario, il comportamento non stazionario deve essere coperto. Attualmente, è possibile identificare tre gruppi di modelli:

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

La prima tipologia di modello tenta di rappresentare le caratteristiche fisiche del pneumatico attraverso una descrizione puramente matematica. Pertanto questi tipi di modelli partono da un curve caratteristiche ricavate sperimentalmente e cercano di derivare un comportamento approssimativo dall'interpolazione di dati. Un esem-

pio ben noto di questo approccio è il modello di Pacejka o *Magic Formula Tire Model* [7]. Questo tipo di modellazione è adatta per la simulazione di manovre di guida, dove il comportamento di interesse è per lo più la gestione del veicolo e le frequenze di uscita sono ben al di sotto delle frequenze di risonanza della cintura dello pneumatici. I modelli fisici o i modelli ad alta frequenza, come i modelli agli elementi finiti, sono in grado di rilevare fenomeni di frequenza più elevata come le vibrazioni della membrana. 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 computer per essere risolto, nonché molti dati. Dall'altro lato, i modelli matematici sono veloci in termini di calcolo, ma richiedono un'accurata pre-elaborazione dei dati sperimentali. La terza tipologia di modelli consiste in un'estensione dei modelli matematici attraverso le leggi fisiche al fine di coprire una gamma di frequenza più ampia.
Il modello di pneumatico sviluppato nel modello di veicolo presentato da Larcher in [4] si basa sulla Magic Formula 6.2. Una panoramica generale sui modelli della Magic Formula e sul calcolo dello slip degli pneumatici è descritta nelle sezioni seguenti, mentre l'insieme completo delle equazioni del modello implementato è riportato nell'Appendice. Come vedremo nel Capitolo 3 il modello di pneumatico deve essere combinato con un'interfaccia di pneumatico/strada modellata adeguata per ottenere risultati significativi.

2.2 Il modello Pacejka

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

$$y(x) = D\sin\{C\arctan[B(x+S_h) - E(B(x+S_h) - \arctan(B(x+S_h)))]\} + S_v \tag{2.1}$$

Parametri	Significato
B	Fattore di rigidezza
C	Fattore di forma
D	Valore massimo della forza o coppia
E	Fattore di curvatura
S_v	Spostamento in verticale della curva caratteristica
S_h	Spostamento in orizzontale della curva caratteristica

Tabella 2.1: Significato dei valori del modello di Pacejka.

2.3 Standardizzazione ETRTO

2.4 Contatto con la Superficie Stradale

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

$$z = z(x, y) \tag{2.2}$$

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

$$P^\star = M - R_0 \boldsymbol{e}_{zC} \begin{bmatrix} x^\star \\ y^\star \\ z^\star \end{bmatrix} \tag{2.3}$$

dove R_0 è il raggio dello pneumatico indeformato e $\boldsymbol{e}_{zC}$ è il vettore unitario che definisce l'asse z_c del sistema di riferimento del vettore ruota.
La prima stima del sistema di riferimento del punto di contatto $RF_{PC^\star}$ è un frame con origine in $P^\star$ e orientamento dell'asse definito dall'orientamento dell'asse del sistema portante della ruota.

$$RF_{PC^\star} = \left[\begin{array}{ccc|c} & [R_{RF_{wh}}] & & \begin{matrix} x^\star \\ y^\star \\ z^\star \end{matrix} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2.4}$$

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

$$\boldsymbol{e}_x = \frac{\boldsymbol{e}_{yC} \times \boldsymbol{e}_n}{|\boldsymbol{e}_{yC} \times \boldsymbol{e}_n|} \qquad \boldsymbol{e}_y = \boldsymbol{e}_n \times \boldsymbol{e}_x \tag{2.5}$$

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

$$\begin{aligned} {}_{PC^\star}\boldsymbol{r}_{Q^\star_{1,2}} &= \pm\Delta x \\ {}_{PC^\star}\boldsymbol{r}_{Q^\star_{3,4}} &= \pm\Delta y \end{aligned} \tag{2.6}$$

e quindi, i quattro punti di campionamento sono:

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

Al fine di campionare la patch di contatto nel modo più efficiente possibile, le distanze di Δx e Δy, dell'equazione precedente, vengono regolate in base al raggio del pneumatico indeformato R_0 e alla larghezza del pneumatico B. I valori di queste due quantità possono essere trovate in letteratura e sono $\Delta x = 0.1R_0$ e $\Delta x = 0.3B$. Attraverso questa definizione, si può ottenere un comportamento realistico durante la simulazione. Ora il componente traccia z in corrispondenza dei quattro punti campione viene valutato attraverso la funzione $z(x, y)$ (Eq. 3.1), quindi, aggiornando la terza coordinata dei punti di sondaggio $Q_i^\star$, otteniamo i corrispondenti punti campione Q_i sulla superficie della pista locale. La linea fissata dai punti Q_1, Q_2 e rispettivamente Q_3, Q_4, può ora essere utilizzata per definire il vettore normale del piano della pista locale (Figura??). Pertanto, il vettore normale è definito come:

$$\boldsymbol{e}_n = \frac{\boldsymbol{r}_{Q_1Q_2} \times \boldsymbol{r}_{Q_4Q_3}}{|\boldsymbol{r}_{Q_1Q_2} \times \boldsymbol{r}_{Q_4Q_3}|} \tag{2.8}$$

dove sono $\boldsymbol{r}_{Q_2Q_1}$ e $\boldsymbol{r}_{Q_4Q_3}$ sono i vettori che puntano rispettivamente da Q_1 a Q_2 e da Q_3 a Q_4. Applicando Eq 3.4 è ora possibile calcolare i vettori unitari $\boldsymbol{e}_x$ e $\boldsymbol{e}_y$ del piano di locale del punto di contatto. Il punto di contatto P si ottiene aggiornando

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

$$P = \frac{1}{4} \begin{bmatrix} \sum_{i=1}^{4} x_i \\ \sum_{i=1}^{4} y_i \\ \sum_{i=1}^{4} z_i \end{bmatrix} \tag{2.9}$$

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

$$RF_{PC} = \left[\begin{array}{ccc|c} & & & x_P \\ \left[\boldsymbol{e}_x\right] & \left[\boldsymbol{e}_y\right] & \left[\boldsymbol{e}_z\right] & y_P \\ & & & z_P \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2.10}$$

Attraverso questo approccio di modellazione, le informazioni della traccia locale normal vector $\boldsymbol{e}_n$, insieme al punto di contatto locale P sono in grado di rappresentare l'irregolarità locale in modo soddisfacente. Come accade in realtà, bordi taglienti o discontinuità del manto stradale saranno smussate da questo approccio. Alcuni casi dimostrativi sono illustrati nella Figura**??**.

Computation Geometry Algorithms 3

3.1 A Brief Introduction

Computational geometry is a branch of *computer science* devoted to the study of algorithms which can be expressed in other forms of geometry. Historically, it is considered one of the oldest fields in computing, although modern computational geometry is a recent development. The main reason for the development of computational geometry has been due to progress made in computer graphics, *Computer-Aided Design* (CAD), *Computer-Aided Manufacturing* (CAM) and mathematical visualization. Applications of computational geometry can be found in robotics, integrated circuit design, computer vision, *Computer-Aided Engineering* (CAE) and *Geographic Information Systems* (GIS).
The main branches of computational geometry are:

- *Combinatorial computational* (or *algorithmic geometry*), which deals with geometric objects as discrete entities. For example, it can be used to determine the smallest polyhedron or polygon that contains all points that are given (convex hull problem) or the closest point to a query point from a set of points (nearest neighbor problem).
- *Numerical computational geometry* (or *machine geometry*, or *Computer-Aided Geometric Design* (CAGD)), which deals mainly with representing real world objects in forms suitable for computer computations in CAD and CAM systems. This branch may be seen as a development of descriptive geometry and is often considered a branch of computer graphics or CAD. For example, im-

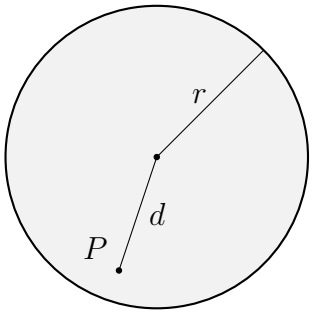


Figura 3.1: Point-circle intersection problem scheme.

portant portions here are parametric surfaces and curves, such as *spline curves* and *Bezier curves*.

In this chapter all algorithms which will be used later during the geometrical analysis of the mesh-tire intersection are explained in depth. These algorithms are the solution of some simple but very important problems which must be solved efficiently. In particular the intersections between:

- point and segment;
- point and circle;
- ray and circle;
- ray and triangle;

will be explored in order to find the best way in terms of *computational efficiency* to solve the specific geometrical problem.

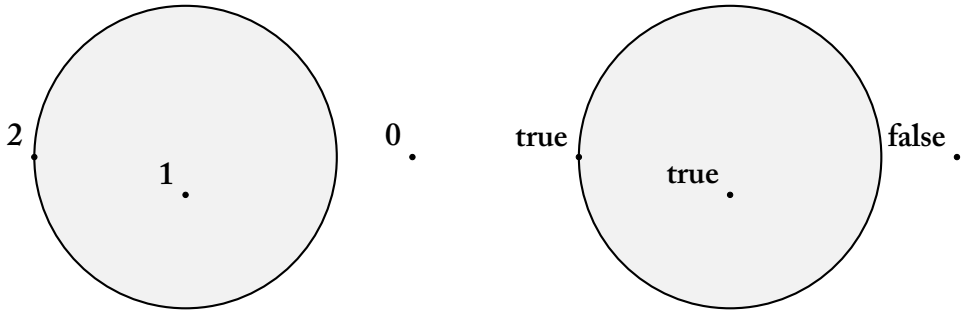
3.2 Point-Segment Intersection

3.3 Point-Circle Intersection

Having a circle with center $C = (x_c, y_c)$ and radius r, the problem consists in finding out whether a query point $P = (x_p, y_p)$ is inside, outside or on the circle. The solution to the problem is simple: the distance between the circle center C and the query point P is given by the *Pythagorean theorem* as

$$d = \sqrt{(x_p - x_c)^2 + (y_p - y_c)^2} \tag{3.1}$$

The query point P is *inside* the circle if $d < r$, on the circle if $d = r$, and *outside* the circle if $d > r$. Little work can be saved by comparing d^2 with r^2 instead: the point



(a) Integer output (b) Boolean output

FIGURA 3.2: Output schemes of the point-circle intersection problem.

P is *inside* the circle if $d^2 < r^2$, on the circle if $d^2 = r^2$, and *outside* the circle if $d^2 > r^2$. Thus, the final comparison will be between the number $(x_p - x_c)^2 + (y_p - y_c)^2$ and r^2.
The *inputs* of the point-circle intersection algorithm are:

- the circle center $C = (x_c, y_c)$;
- the circle radius r;
- a query point $P = (x_p, y_p)$.

The *output* could be an integer which value is:

- 0 if the point is outside;
- 1 if the point is inside;
- 2 if the point is on the circle.

Another option could be a boolean which value is:

- false if the point is outside;
- true if the point is inside or on the circle.

On Figura 3.3 the schemes for the point-circle intersection algorithm with integer and boolean outputs are reported.

3.4 Ray-Circle Intersection

3.5 Ray-Triangle Intersection

Having a triangle with vertices (V_1, V_2, V_3) and a ray R with origin R_O and direction R_D, the problem consists in finding out whether the ray hits or not the triangle

With integer output

$d = (x_p - x_c)^2 + (y_p - y_c)^2$
if $(d > r^2)\{$
  return 0
$\}$ **else if** $(d < r^2)\{$
  return 1
$\}$ **else** $\{$
  $// d = r^2$
  return 2
$\}$

With boolean output

$d = (x_p - x_c)^2 + (y_p - y_c)^2$
if $(d > r^2)\{$
  return false
$\}$ **else** $\{$
  $// d <= r^2$
  return true
$\}$

Figura 3.3: Point-circle intersection algorithm schemes.

and if so, where is the intersection point P. Over the last decades, plenty of algo-

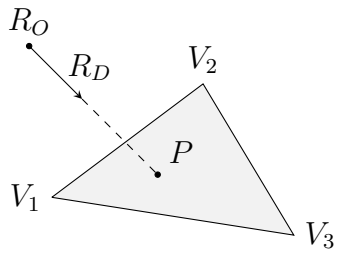


Figura 3.4: Ray-triangle intersection problem scheme.

rithms for solving this problem had been purposed, so there are several solutions to the ray/triangle or ray-triangle intersection problem. Three of the most relevant algorithms are:

- *Badouel* algorithm;
- *Segura* algorithm;
- *Möller-Trumbore* algorithm.

As Jiménez, Segura e Feito states in [2], the Möller-Trumbore's is the faster algorithm when the normal and/or the projection plane have not been previously stored, as in this thesis.

3.5.1 Möller-Trumbore Algorithm

The inputs of the Möller-Trumbore algorithm are:

- Triangle vertices (V_1, V_2, V_3);

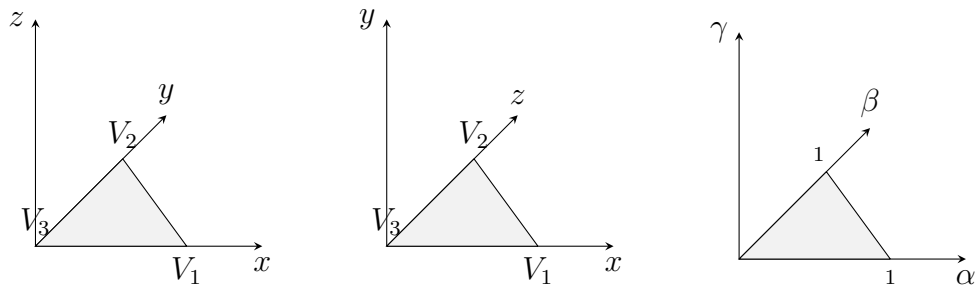


FIGURA 3.5: Transformation and base change of ray in Möller-Trumbore algorithm.

- Segment points (Q_1, Q_2).

With back-face culling

$Q = Q_2 - Q_1$
$E_1 = V_2 - V_1$
$E_2 = V_3 - V_1$
$A = Q \times E_2$
$D = A \cdot E_1$
if $(D > \varepsilon)\{$
  $T = Q_1 - V_1$
  $u = A \cdot T$
  if $(u < 0.0 \,||\, u > D)\{$
    return false
  $\}$
  $B = T \times E_1$
  $v = B \cdot Q$
  if $(v < 0.0 \,||\, u + v > D)\{$
    return false
  $\}$
$\}$ **else if** $(D < -\varepsilon)\{$
  $T = Q_1 - V_1$
  $u = A \cdot T$
  if $(u > 0.0 \,||\, u < D)\{$
    return false
  $\}$
  $B = T \times E_1$
  $v = B \cdot Q$
  if $(v > 0.0 \,||\, u + v < D)\{$
    return false
  $\}$
$\}$ **else** $\{$
  return false
$\}$
$D_{inv} = 1.0/D$
$t = (B \cdot E_2) * D_{inv}$
if $(t > 0.0)\{$
  $P = Q + D * t$
  return true
$\}$ **else** $\{$
  return false
$\}$

Without back-face culling

$Q = Q_2 - Q_1$
$E_1 = V_2 - V_1$
$E_2 = V_3 - V_1$
$A = Q \times E_2$
$D = A \cdot E_1$
if $(D < \varepsilon)\{$
  return false
$\}$
$T = Q_1 - V_1$
$u = A \cdot T$
if $(u < 0.0 \,||\, u > D)\{$
  return false
$\}$
$B = T \times E_1$
$v = B \cdot Q$
if $(v < 0.0 \,||\, u + v > D)\{$
    return false
$\}$
$D_{inv} = 1.0/D$
$t = (B \cdot E_2) * D_{inv}$
if $(t > 0.0)\{$
  $P = Q + D * t$
  return true
$\}$ **else** $\{$
  return false
$\}$

Figura 3.6: Point-circle intersection algorithm schemes.

A Chapter of Examples 4

4.1 A Table

Feature	Misuse-based	Anomaly-based
Modeled activity:	Malicious	Normal
Detection method:	Matching	Deviation
Threats detected:	Known	Any
False negatives:	High	Low
False positives:	Low	High
Maintenance cost:	High	Low
Attack desc.:	Accurate	Absent
System design:	Easy	Difficult

Tabella 4.1: Duality between misuse- and anomaly-based intrusion detection techniques. Note that, an anomaly-based CAMs can detect "Any" threat, under the assumption that an attack always generates a deviation in the modeled activity.

4.2 Code

```
1  /* ... */ cd['<'] = {0.1, 0.11} cd['a'] = {0.01, 0.2} cd['b'] =
2  {0.13, 0.23} /* ... */
3
4  b = decode(arg3_value);
5
```

```
6  if ( !(cd['c'][0] < count('c', b) < cd['c'][1]) ||\
7       !(cd['<'][0] < count('<', b) < cd['<'][1]) ||\
8       ... || ...)  fire_alert("Anomalous content detected!");
9  /* ... */
```

4.3 A Sideways Table

Approach	Time	Header	Payload	Stochastic	Determ.	Clustering
[phad]		•				•
[kruegel:sac2002:anomaly]		•	•	•		
[protocolanom]		•		•	•	
[ramadas]			•			•
[rules-payl]	•		•		•	
[zanero-savaresi]		•	•			•
[wang:raid2004:payl]			•	•		
[zanero-pattern]		•	•			•
[DBLP:conf/iwia/BolzoniEHZ06]		•	•			•
[wang:raid2006:anagram]			•	•		

Tabella 4.2: Taxonomy of the selected state of the art approaches for network-based anomaly detection.

4.4 A Figure

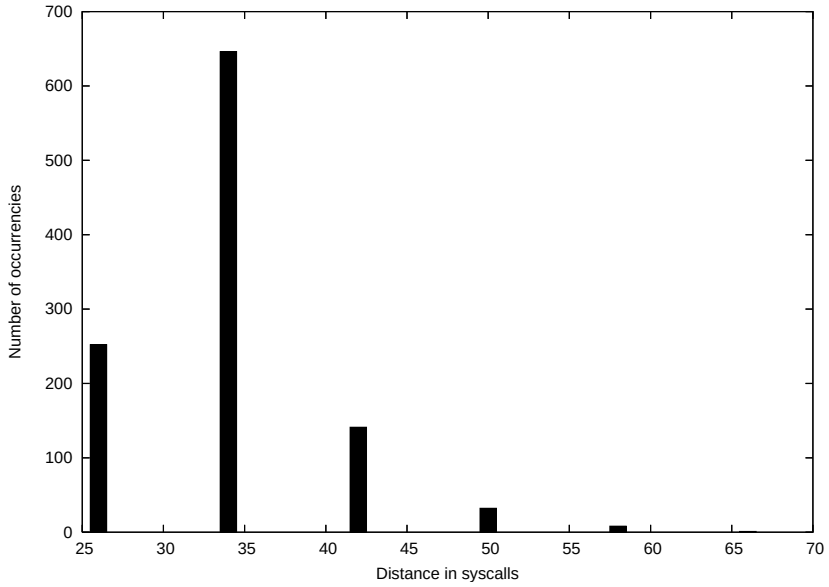


Figura 4.1: `telnetd`: distribution of the number of other system calls among two `execve` system calls (i.e., distance between two consecutive `execve`).

4.5 Bulleted List

- O =“Intrusion”, $\neg O$ =“Non-intrusion”;
- A =“Alert reported”, $\neg A$ =“No alert reported”.

4.6 Numbered List

1. O =“Intrusion”, $\neg O$ =“Non-intrusion”;
2. A =“Alert reported”, $\neg A$ =“No alert reported”.

4.7 A Description

Time refers to the use of *timestamp* information, extracted from network packets, to model normal packets. For example, normal packets may be modeled by their minimum and maximum inter-arrival time.

4.8 An Equation

$$d_a(i,j) := \begin{cases} K_a + \alpha_a \delta_a(i,j) & \text{if the elements are different} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

4.9 A Theorem, Proposition & Proof

Theorem 4.9.1 $a^2 + b^2 = c^2$

Proposition 4.9.2 $3 + 3 = 6$

Proof 4.9.1 *For any finite set* $\{p_1, p_2, ..., p_n\}$ *of primes, consider* $m = p_1 p_2 ... p_n + 1$. *If* m *is prime it is not in the set since* $m > p_i$ *for all* i. *If* m *is not prime it has a prime divisor* p. *If* p *is one of the* p_i *then* p *is a divisor of* $p_1 p_2 ... p_n$ *and hence is a divisor of* $(m - p_1 p_2 ... p_n) = 1$, *which is impossible; so* p *is not in the set. Hence a finite set* $\{p_1, p_2, ..., p_n\}$ *cannot be the collection of all primes.*

4.10 Definition

Definition 4.10.1 (Anomaly-based CAM) *An* anomaly-based CAM *is a type of CAM that generate alerts* $\mathbb{A}$ *by relying on normal activity profiles.*

4.11 A Remark

Remark 1 *Although the network stack implementation may vary from system to system (e.g.,* Windows *and* Cisco *platforms have different implementation of CAM).*

4.12 An Example

Example 4.12.1 (Misuse *vs.* Anomaly) *A misuse-based system* M *and an anomaly-based system* A *process the same log containing a full dump of the system calls invoked by the kernel of an audited machine. Log entries are in the form:*

```
<function_name>(<arg1_value>, <arg2_value>, ...)
```

4.13 Note

Note 4.13.1 (Inspection layer) *Although the network stack implementation may vary from system to system (e.g.,* Windows *and* Cisco *platforms have different implementation of CAM), it is important to underline that the notion of IP, TCP, HTTP* packet *is well defined in a system-agnostic way, while the notion of* operating system activity *is rather vague and by no means standardized.*

Convenzioni e Notazioni A

A.1 Convenzioni

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

A.2 Matrice di Trasformazione

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

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

dove R_m è la matrice di rotazione 3×3 del sistema di riferimento in movimento e O_{mx}, O_{my} e O_{mz} sono le coordinate della sua origine nel sistema di riferimento assoluto o nativo. L'introduzione dell'elemento fittizio 1 nel vettore della posizione di origine e la successiva spaziatura interna zero della matrice rende possibili le moltiplicazioni matrice-vettore, rendendo la matrice di trasformazione un modo

compatto 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.

Library Code B

B.1 RoadRDF.hh

```
1 ///
2 /// file: MeshRDF.hh
3 ///
4
5 #pragma once
6 #include <AABBtree.hh>
7 #include <Eigen/Dense>  // Eigen linear algebra Library
8 #include <cmath>        // Math.h - STD math Library
9 #include <fstream>      // fStream - STD File I/O Library
10 #include <iostream>     // Iostream - STD I/O Library
11 #include <string>       // String - STD String Library
12 #include <vector>       // Vector - STD Vector/Array Library
13 #include <memory>
14
15 // Print progress to console while loading (large models)
16 #define RDF_CONSOLE_OUTPUT
17
18 #ifndef RDF_ERROR
19   #define RDF_ERROR(MSG) {                 \
20     std::ostringstream ost; ost << MSG;    \
21     throw std::runtime_error( ost.str() ); \
22   }
```

```
23 #endif
24
25 #ifndef RDF_ASSERT
26   #define RDF_ASSERT(COND,MSG) \
27     if ( !(COND) ) RDF_ERROR( MSG )
28 #endif
29
30 //! RDF mesh computations routine
31 namespace RDF {
32
33   typedef double real_type;  //!< Real number type
34   typedef int    int_type;   //!< Integer number type
35
36   typedef Eigen::Vector2d vec2;  //!< 2D vector type
37   typedef Eigen::Vector3d vec3;  //!< 3D vector type
38   typedef Eigen::Matrix3d mat3;  //!< 3x3 matrix type
39
40   typedef std::basic_ostream<char> ostream_type;  //!< Output
       stream type
41
42   //! Class that handle triangle bounding box
43   class BBox3D {
44   private:
45     real_type Xmin;  //!< Xmin shadow domain point
46     real_type Ymin;  //!< Ymin shadow domain point
47     real_type Xmax;  //!< Xmax shadow domain point
48     real_type Ymax;  //!< Ymax shadow domain point
49
50   public:
51
52     //! Default constructor for orientation object.
53     BBox3D() {}
54
55     //! Variable set constructor for Bounding box object.
56     BBox3D( vec3 const Vertices[3] ) {
57       updateBBox3D( Vertices );
58     }
59
60     //! Set Xmin shadow domain point.
```

```
61    void setXmin(real_type _Xmin) { Xmin = _Xmin; }
62
63    //! Set Ymin shadow domain point.
64    void setYmin(real_type _Ymin) { Ymin = _Ymin; }
65
66    //! Set Xmax shadow domain point.
67    void setXmax(real_type _Xmax) { Xmax = _Xmax; }
68
69    //! Set Ymax shadow domain point.
70    void setYmax(real_type _Ymax) { Ymax = _Ymax; }
71
72    //! Get Xmin shadow domain point.
73    real_type getXmin() const { return Xmin; }
74
75    //! Get Ymin shadow domain point.
76    real_type getYmin() const { return Ymin; }
77
78    //! Get Xmax shadow domain point.
79    real_type getXmax() const { return Xmax; }
80
81    //! Get Ymax shadow domain point.
82    real_type getYmax() const { return Ymax; }
83
84    //! Clear the bounding box domain.
85    void clear(void);
86
87    //! Print bounding box vertices.
88    void
89    print(ostream_type & stream) const {
90      stream
91        << "BBOX (xmin,ymin,xmax,ymax) = ( " << Xmin << ", " <<
             Ymin
92        << ", " << Xmax << ", " << Ymax << " )\n";
93    }
94
95    //! Update the bounding box domain with multiple input
        triangles object.
96    void
97    updateBBox3D( vec3 const Vertices[3] );
```

```
98  };
99
100 //! Class for handling triangles
101 class Triangle3D {
102 private:
103   vec3      Vertices[3];  //!< Vertices vector
104   real_type Friction;     //!< Face friction coefficient
105   BBox3D    TriangleBBox; //!< Triangle bounding box
106
107 public:
108
109   //! Default constructor for orientation object.
110   Triangle3D() {}
111
112   //! Variable set constructor for orientation object.
113   Triangle3D( vec3 const _Vertices[3], real_type _Friction ) {
114     Vertices[0] = _Vertices[0];
115     Vertices[1] = _Vertices[1];
116     Vertices[2] = _Vertices[2];
117     Friction    = _Friction;
118     TriangleBBox.updateBBox3D(Vertices);
119   }
120
121   //! Get triangle face normal versor.
122   vec3
123   Normal(void) const {
124     vec3 d1 = Vertices[1] - Vertices[0];
125     vec3 d2 = Vertices[2] - Vertices[0];
126     return d1.cross(d2).normalized();
127   }
128
129   //! Set vertices vector and update bounding box domain.
130   void
131   setVertices( vec3 const _Vertices[3] ) {
132     Vertices[0] = _Vertices[0];
133     Vertices[1] = _Vertices[1];
134     Vertices[2] = _Vertices[2];
135     TriangleBBox.updateBBox3D(Vertices);
136   }
```

```
137
138   //! Set friction.
139   void
140   setFriction( real_type _Friction ) { Friction = _Friction; }
141
142   //! Get i-th vertex.
143   vec3 const &
144   getithVertex( unsigned i ) const
145   { return Vertices[i]; }
146
147   //! Get friction coefficent on the face.
148   real_type
149   getFriction(void) const
150   { return Friction; }
151
152   //! Get triangle bonding box.
153   BBox3D const &
154   getBBox(void) const
155   { return TriangleBBox; }
156
157   //! Print vertices information.
158   void
159   print( ostream_type & stream ) const {
160     stream
161       << "V1:\t" << Vertices[0] << '\n'
162       << "V2:\t" << Vertices[1] << '\n'
163       << "V3:\t" << Vertices[2] << std::endl;
164   }
165 };
166
167 typedef std::vector<Triangle3D> Triangle3D_list;
168
169 //! Algorithms for RDF mesh computations routine
170 namespace algorithms {
171
172   //! Split a string into a string array at a given token.
173   void
174   split(
175     std::string const        & in,    //!< Input string
```

```
176       std::vector<std::string> & out,   //!< Output string vector
177       std::string const        & token  //!< Token
178     );
179
180     //! Get tail of string after first token and possibly
            following spaces.
181     std::string
182     tail( std::string const & in );
183
184     //! Get first token of string.
185     std::string
186     firstToken( std::string const & in );
187
188     //! Get element at given index position.
189     template<typename T>
190     T const &
191     getElement(
192       std::vector<T> const & elements,  //!< Elements vector
193       std::string    const & index      //!< Index position
194     );
195   }  // namespace algorithms
196
197   //! Class for handlinf mesh surface object
198   class MeshSurface {
199   private:
200     std::vector<std::shared_ptr<Triangle3D> > PtrTriangleVec;
            //!< Triangles vector list
201     std::vector<G2lib::BBox::PtrBBox>         PtrBBoxVec;
                        //!< Bounding boxes pointers
202     G2lib::AABBtree::PtrAABB PtrTree = std::make_shared<G2lib::
            AABBtree>(); //!< Mesh tree pointer
203
204     MeshSurface( MeshSurface const & ) = delete; // costruttore
            di copia
205     MeshSurface & operator = ( MeshSurface const & ) = delete; //
             operatore di copia
206
207   public:
208     // Default set constructor for mesh object.
```

```
209   MeshSurface() {};
210
211   // Variable set constructor for mesh object.
212   MeshSurface( std::string const & Path ){
213     bool load = LoadFile(Path);
214     RDF_ASSERT( load, "Error while reading file" );
215     updatePtrBBox();
216     PtrTree->build(PtrBBoxVec);
217   }
218
219   //! Get all triangles inside the mesh as a vector.
220   std::vector<std::shared_ptr<Triangle3D> >
221   getTrianglesPtr(void)
222   { return PtrTriangleVec; }
223
224       //! Get i-th triangle.
225   std::shared_ptr<Triangle3D>
226   ithTrianglePtr( unsigned i )
227   { return PtrTriangleVec[i]; }
228
229   //! Get AABB tree.
230   G2lib::AABBtree::PtrAABB
231   getAABBPtr(void)
232   { return PtrTree; }
233
234   //! Print data in file.
235   void
236   printData( std::string const & FileName );
237
238   //! Get the mesh G2lib bounding boxes pointers vector.
239   std::vector<G2lib::BBox::PtrBBox> const &
240   getPtrBBox() const
241   { return PtrBBoxVec; }
242
243   void
244   set( MeshSurface const & in ) {
245     this->PtrTriangleVec = in.PtrTriangleVec;
246     this->PtrBBoxVec  = in.PtrBBoxVec;
247     this->PtrBBoxVec  = in.PtrBBoxVec;
```

```
248     }
249
250     //! Load the RDF model and print information on a file.
251     //! If RDF model is properly loaded true value is returned
252     bool
253     LoadFile( std::string const & Path );
254
255   private:
256     //! Update the mesh G2lib bounding boxes pointers vector.
257     void updatePtrBBox(void);
258
259     // Generate vertices from a list of positions face line.
260     void
261     GenVerticesFromRawRDF(
262       std::vector<vec3> const & iNodes,
263       std::string       const & icurline,
264       vec3                      oVerts[3]
265     );
266   };
267
268 }  // namespace RDF
269
270 ///
271 /// eof: MeshRDF.hh
272 ///
```

B.2 RoadRDF.cc

```
1 #include "RoadRDF.hh"  // RDF file extention Loader
2
3
4 //! RDF mesh computations routine
5 namespace RDF {
6
7   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - - - - - -
8   // class BBox3D
9   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - - - - - -
```

```
10
11 void
12 BBox3D::clear(void) {
13   Xmin = std::numeric_limits<real_type>::quiet_NaN();
14   Ymin = std::numeric_limits<real_type>::quiet_NaN();
15   Xmax = std::numeric_limits<real_type>::quiet_NaN();
16   Ymax = std::numeric_limits<real_type>::quiet_NaN();
17 }
18
19 //! Update the bounding box domain with multiple input
      triangles object
20 void
21 BBox3D::updateBBox3D( vec3 const Vertices[3] ) {
22   G2lib::minmax3( Vertices[0][0], Vertices[1][0], Vertices
        [2][0], Xmin, Xmax );
23   G2lib::minmax3( Vertices[0][1], Vertices[1][1], Vertices
        [2][1], Ymin, Ymax );
24 }
25
26 // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      - - - - - - - -
27 // class MeshSurface
28 // - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      - - - - - - - -
29
30 //! Print data in file
31 void
32 MeshSurface::printData( std::string const & FileName ) {
33   // Create/Open Out.txt
34   std::ofstream file(FileName);
35
36   file
37     // Print Vertices
38     << "LOADED RDF MESH DATA\n\n"
39     // Print Legend
40     << "Legend:\n"
41     << "\tVi: i-th vertex\n"
42     << "\t N: normal to the face\n"
43     << "\t F: friction coefficient\n\n";
```

```
44
45    for ( unsigned i = 0; i < PtrTriangleVec.size(); ++i ) {
46      Triangle3D const & Ti = *PtrTriangleVec[i];
47      vec3 const & V0 = Ti.getithVertex(0);
48      vec3 const & V1 = Ti.getithVertex(1);
49      vec3 const & V2 = Ti.getithVertex(2);
50      vec3          N = Ti.Normal();
51      // Print Vertices and Friction
52      file
53        << "TRIANGLE " << i
54        << "\n\tV0:\t" << V0[0] << ", " << V0[1] << ", " << V0[2]
55        << "\n\tV1:\t" << V1[0] << ", " << V1[1] << ", " << V1[2]
56        << "\n\tV2:\t" << V2[0] << ", " << V2[1] << ", " << V2[2]
57        << "\n\t N:\t" << N[0]  << ", " << N[1]  << ", " << N[2]
58        << "\n\t F:\t" << Ti.getFriction()
59        << "\n\n";
60    }
61    // Close File
62    file.close();
63  }
64
65  //! Update the mesh G2lib bounding boxes pointers vector
66  void MeshSurface::updatePtrBBox(void) {
67    PtrBBoxVec.clear();
68    RDF::BBox3D iBBox;
69    for (unsigned id = 0; id < PtrTriangleVec.size(); ++id) {
70      iBBox = (*PtrTriangleVec[id]).getBBox();
71      PtrBBoxVec.push_back(G2lib::BBox::PtrBBox(
72          new G2lib::BBox(iBBox.getXmin(), iBBox.getYmin(), iBBox
               .getXmax(),
73                          iBBox.getYmax(), id, 0)));
74      iBBox.clear();
75    }
76  }
77
78  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
       - - - - - - - -
79  // namespace algorithms
80  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
    - - - - - - - -
81
82  //! Holds all of the algorithms needed for the mesh processing
83  namespace algorithms {
84
85    //! Split a string into a string array at a given token
86    void
87    split(
88      std::string const        & in,    //!< Input string
89      std::vector<std::string> & out,   //!< Output string vector
90      std::string const        & token  //!< Token
91    ) {
92      out.clear();
93
94      std::string temp;
95
96      for ( int i = 0; i < int(in.size()); ++i ) {
97        std::string test = in.substr(i, token.size());
98        if (test == token) {
99          if (!temp.empty()) {
100           out.push_back(temp);
101           temp.clear();
102           i += (int)token.size() - 1;
103         } else {
104           out.push_back("");
105         }
106       } else if (i + token.size() >= in.size()) {
107         temp += in.substr(i, token.size());
108         out.push_back(temp);
109         break;
110       } else {
111         temp += in[i];
112       }
113     }
114   }
115
116   //! Get tail of string after first token and possibly
        following spaces
117   std::string
```

```
118   tail(std::string const & in ) {
119     size_t token_start = in.find_first_not_of(" \t");
120     size_t space_start = in.find_first_of(" \t", token_start);
121     size_t tail_start  = in.find_first_not_of(" \t",
          space_start);
122     size_t tail_end    = in.find_last_not_of(" \t");
123     if (tail_start != std::string::npos && tail_end != std::
          string::npos) {
124       return in.substr(tail_start, tail_end - tail_start + 1);
125     } else if (tail_start != std::string::npos) {
126       return in.substr(tail_start);
127     }
128     return "";
129   }
130
131   //! Get first token of string
132   std::string
133   firstToken( std::string const & in ) {
134     if (!in.empty()) {
135       size_t token_start = in.find_first_not_of(" \t\r\n");
136       if (token_start != std::string::npos) {
137         size_t token_end = in.find_first_of(" \t\r\n",
              token_start);
138         if (token_end != std::string::npos) {
139           return in.substr(token_start, token_end - token_start
                );
140         } else {
141           return in.substr(token_start);
142         }
143       }
144     }
145     return "";
146   }
147
148   //! Get element at given index position
149   template<typename T>
150   T const &
151   getElement(
152     std::vector<T> const & elements,  //!< Elements vector
```

```
153       std::string    const & index         //!< Index position
154     ) {
155       // std::cout << "Index: " << index << std::endl;
156       int_type id = std::stoi(index);
157
158       if ( id < 0 ) perror("ELEMENTS indexes cannot be negative")
              ;
159
160       return elements[id - 1];
161     }
162
163   }  // namespace algorithms
164
165   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - - - - - -
166   // class Loader
167   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - - - - - -
168
169   //! Load the RDF model and print information on a file
170   bool
171   MeshSurface::LoadFile( std::string const & Path ) {
172     // Check if the file is an ".rdf" file, if not return false
173     if (Path.substr(Path.size() - 4, 4) != ".rdf") {
174       perror("Not a .rdf file");
175       return false;
176     }
177
178      // Check if the file had been correctly open, if not return
           false
179     std::ifstream file(Path);
180     if (!file.is_open()) {
181       perror("File not opened");
182       return false;
183     }
184
185     // Vector for nodes coordinates
186     std::vector<vec3> Nodes;
187
```

```
188     bool nodes_parse    = false;
189     bool elements_parse = false;
190
191 #ifdef RDF_CONSOLE_OUTPUT
192     int_type const outputEveryNth = 5000;
193     int_type outputIndicator      = outputEveryNth;
194 #endif
195
196     std::string curline;
197     while (std::getline(file, curline)) {
198 #ifdef RDF_CONSOLE_OUTPUT
199       if ((outputIndicator = ((outputIndicator + 1) %
              outputEveryNth)) == 1) {
200         std::cout
201           << "\r- "
202           << "Loading mesh..."
203           << "\t triangles > "
204           << PtrTriangleVec.size() << std::endl;
205       }
206 #endif
207
208       std::string token = algorithms::firstToken(curline);
209       if ( token == "[NODES]" || token == "NODES" ) {
210         nodes_parse    = true;
211         elements_parse = false;
212
213         continue;
214       } else if (token == "[ELEMENTS]" || token == "ELEMENTS") {
215         nodes_parse    = false;
216         elements_parse = true;
217         continue;
218       } else if (token[0] == '{') {
219         // commento multiriga, continua a leggere fino a che
              trovo '}'
220         continue;
221       } else if (token[0] == '%' || token[0] == '#' || token[0]
             == '\r') {
222         // Check comments or empty lines
223         continue;
```

```
224       }
225
226       // Generate a Vertex Position
227       if (nodes_parse) {
228         std::vector<std::string> spos;
229         vec3 vpos;
230
231         algorithms::split(algorithms::tail(curline), spos, " ");
232
233         vpos[0] = std::stod(spos[0]);
234         vpos[1] = std::stod(spos[1]);
235         vpos[2] = std::stod(spos[2]);
236
237         Nodes.push_back(vpos);
238       }
239
240       // Generate a Face (vertices & indices)
241       if (elements_parse) {
242         // Generate the triangle vertices from the elements
243         vec3 iVerts[3];
244         GenVerticesFromRawRDF( Nodes, curline, iVerts );
245
246         // Get the triangle friction from current line
247         std::vector<std::string> curlinevec;
248         algorithms::split(curline, curlinevec, " ");
249         real_type iFriction = std::stod(curlinevec[4]);
250
251         // Create a shared pointer for the last triangle and push
                it in the pointer vector
252         PtrTriangleVec.push_back(std::shared_ptr<Triangle3D>(new
               Triangle3D(iVerts,iFriction)));
253       }
254     }
255
256 #ifdef RDF_CONSOLE_OUTPUT
257     std::cout << std::endl;
258 #endif
259
260     file.close();
```

```
261
262     if (PtrTriangleVec.empty()) {
263       perror("Loaded mesh is empty");
264       return false;
265     } else {
266       return true;
267     }
268   }
269
270   // Generate vertices from a list of positions face line
271   void
272   MeshSurface::GenVerticesFromRawRDF(
273     std::vector<vec3> const & iNodes,
274     std::string       const & icurline,
275     vec3                      oVerts[3]
276   ) {
277     std::vector<std::string> svert;
278     vec3                     vVert;
279     algorithms::split( icurline, svert, " " );
280
281     int_type control_size = int(svert.size() - 4);
282     for ( int i = 1; i < int(svert.size() - control_size); ++i )
          {
283       // Calculate and store the vertex
284       vVert = algorithms::getElement(iNodes, svert[i]);
285       oVerts[i-1] = vVert; // CONTROLLARE i <=
             3!!!!!!!!!!!!!!!!!!!!!!
286     }
287   }
288 }  // namespace RDF
289
290 ///
291 /// eof: MeshRDF.hh
292 ///
```

B.3 PatchTire.hh

```
1 ///
2 /// file: PatchTire.hh
```

```
3 ///
4
5 /*!
6
7 \mainpage
8
9 Contact Patch Evaluation
10 ========================
11
12 Master's Thesis
13
14 Algorithm for Tire Contact Patch Evaluation in Soft Real-Time
15
16 Academic Year 2019 · 2020
17
18 Graduant:
19 --------
20
21 Davide Stocco\n
22 Department of Industrial Engineering\n
23 University of Trento\n
24 davide.stocco@studenti.unitn.it
25
26 Supervisor:
27 ----------
28
29 Prof. Enrico Bertolazzi\n
30 Department of Industrial Engineering\n
31 University of Trento\n
32 enrico.bertolazzi@unitn.it
33
34 */
35
36 #pragma once
37
38 #include <Eigen/Dense>  // Eigen linear algebra Library
39 #include <chrono>       // chrono - STD Time Measurement Library
40 #include <cmath>        // Math.h - STD math Library
41 #include <fstream>      // fStream - STD File I/O Library
```

```
42 #include <iostream>     // Iostream - STD I/O Library
43 #include <string>       // String - STD String Library
44 #include <vector>       // Vector - STD Vector/Array Library
45
46 #include "RoadRDF.hh"  // RDF file extention Loader
47
48 //! Tire computations routine
49 namespace PatchTire {
50
51   typedef double real_type;  //!< Real number type
52   typedef int    int_type;   //!< Integer number type
53
54   typedef Eigen::Vector2d vec2;  //!< 2D vector type
55   typedef Eigen::Vector3d vec3;  //!< 3D vector type
56   typedef Eigen::Matrix3d mat3;  //!< 3x3 matrix type
57
58   static real_type quiteNaN =
59       std::numeric_limits<real_type>::quiet_NaN();  //!< Not-a-
          Number type
60
61   typedef std::basic_ostream<char> ostream_type;  //!< Output
       stream type
62
63   real_type const epsilon = std::numeric_limits<real_type>::
       epsilon();
64
65   //! Algorithms for tire computations routine.
66   namespace algorithms {
67
68     //! Check if a ray hits a triangle object through Möller-
         Trumbore
69     //! intersection algorithm.
70     bool
71     rayIntersectsTriangle(
72       vec3 const      & RayOrigin,        //!< Ray origin
          position
73       vec3 const      & RayVector,        //!< Ray direction
          vector
74       RDF::Triangle3D & Triangle,         //!< Triangle object
```

```
75      vec3            & IntersectionPoint //!< Intersection point
76    );
77
78    //! Find the intersection points between a circle and a line
          segment.
79    //! Output integer gives the number of intersection points.
80    int_type
81    segmentIntersectsCircle(
82      vec2 const & Origin,         //!< Circle origin position
83      real_type    R,              //!< Circle radius
84      vec2 const & Point_1,        //!< Line segment point 1
85      vec2 const & Point_2,        //!< Line segment point 2
86      vec2       & Intersection_1, //!< Intersection point 2
87      vec2       & Intersection_2  //!< Intersection point 2
88    );
89
90    //! Check if a point lays inside or outside a circumfererence
          .
91    //! If output bool is true the point is inside the
          circumfererence,
92    //! otherwise it is outside.
93    bool
94    pointInsideCircle(
95      vec2 const & Origin,  //!< Circle origin position
96      real_type    R,    //!< Circle radius
97      vec2 const & Point    //!< Query point
98    );
99
100   //! Check if a point lays inside or outside a line segment.
101   //! Warning: The point query point must be on the same rect
          of the line
102   //! segment.
103   bool
104   pointOnSegment(
105     vec2 const & Point_1,  //!< Line segment point 1
106     vec2 const & Point_2,  //!< Line segment point 2
107     vec2 const & Point     //!< Query point
108   );
109
```

```
110 }  // namespace algorithms
111
112 //! Class for handling tire disks
113 class Disk {
114 private:
115   vec2              OriginXZ;       //!< X0,Z0 origin vector
116   real_type         Y;              //!< Y0 (= D) origin
        coordinate or offset from center
117   real_type         R;              //!< Disk radius
118   std::vector<vec2> PointsSequence; //!< Sampled terrain points
        vector
119   real_type         PatchLength;    //!< Local patch length
120
121   Disk const & operator = ( Disk const & ); // operatore di
        copia
122
123 public:
124   //! Variable set constructor for orientation object.
125   Disk(
126     vec2 const & _OriginXZ,  //!< X0,Z0 origin coordinate
127     real_type    _Y,    //!< Y0 (= D) origin coordinate
128     real_type    _R     //!< Disk radius
129   ) {
130     OriginXZ = _OriginXZ;
131     Y        = _Y;
132     R        = _R;
133   }
134
135   Disk( Disk const & obj ) {
136     this->OriginXZ       = obj.OriginXZ;
137     this->Y              = obj.Y;
138     this->R              = obj.R;
139     this->PatchLength    = obj.PatchLength;
140     this->PointsSequence = obj.PointsSequence;
141   }
142
143   //! Set origin vector.
144   void
145   setOriginXZ( vec2 const & _OriginXZ )
```

```
146     { OriginXZ = _OriginXZ; }
147
148     //! Get origin vector XZ-axes coordinates.
149     vec2 const & getOriginXZ(void) { return OriginXZ; }
150
151     //! Get origin Y-axis coordinate.
152     real_type getOriginY(void) { return Y; }
153
154     //! Get the overall point sequance length inside the disk.
155     real_type getPatchLength(void) { return PatchLength; }
156
157     //! Set sampled terrain points vector and calculate area
            between disk and
158     //! segment.
159     //! Polygonal chain
160     void
161     setPointsSequence( std::vector<vec2> const & _PointsSequence
            ) {
162       PointsSequence.clear();
163       PointsSequence = _PointsSequence;
164       updatePatchLength();
165     }
166
167   private:
168     //! Calculate area between disk and segment.
169     void updatePatchLength(void);
170   };
171
172   //! Class for tire shadow bounding box
173   class Shadow {
174   private:
175     real_type Xmin;                                  //!< Xmin
            shadow domain point
176     real_type Ymin;                                  //!< Ymin
            shadow domain point
177     real_type Xmax;                                  //!< Xmax
            shadow domain point
178     real_type Ymax;                                  //!< Ymax
            shadow domain point
```

```
179   std::vector<G2lib::BBox::PtrBBox> PtrBBoxVec;  //!< Bounding
          boxes pointers
180   G2lib::AABBtree::PtrAABB PtrTree =
181       std::make_shared<G2lib::AABBtree>();  //!< Mesh tree
              pointer
182
183   Shadow( Shadow const & ); // costruttore di copia
184   Shadow const & operator = ( Shadow const & ); // operatore di
          copia
185
186 public:
187   //! Default constructor for orientation object.
188   Shadow() {}
189   //! Variable set constructor for orientation object.
190   Shadow(
191     real_type _Xmin,  //!< Xmin shadow domain point
192     real_type _Ymin,  //!< Ymin shadow domain point
193     real_type _Xmax,  //!< Xmax shadow domain point
194     real_type _Ymax   //!< Ymax shadow domain point
195   ) {
196     Xmin = _Xmin;
197     Ymin = _Ymin;
198     Xmax = _Xmax;
199     Ymax = _Ymax;
200     updatePtrBBox();
201     PtrTree->build(PtrBBoxVec);
202   }
203
204   //! Set class member Xmin, second argument boolean allows to
          update the
205   //! AABB tree of the tire shadow after the change of domain.
206   void
207   setXmin(
208     real_type _Xmin,  //!< Xmin shadow domain point
209     bool      updateTree = false  //!< Optional boolean for
              updating the tire shadow AABB tree
210   ) {
211     Xmin = _Xmin;
212     if ( updateTree == true ) {
```

```
213         updatePtrBBox();
214         PtrTree->build(PtrBBoxVec);
215       }
216     }
217
218     //! Set class member Ymin, second argument boolean allows to
          update the
219     //! AABB tree of the tire shadow.
220     void
221     setYmin(
222       real_type _Ymin,  //!< Ymin shadow domain point
223       bool      updateTree = false  //!< Optional boolean for
            updating the tire shadow AABB tree
224     ) {
225       Ymin = _Ymin;
226       if (updateTree == true) {
227         updatePtrBBox();
228         PtrTree->build(PtrBBoxVec);
229       }
230     }
231
232     //! Set class member Xmax, second argument boolean allows to
          update the
233     //! AABB tree of the tire shadow.
234     void
235     setXmax(
236       real_type _Xmax,  //!< Xmax shadow domain point
237       bool      updateTree = false  //!< Optional boolean for
            updating the tire shadow AABB tree
238     ) {
239       Xmax = _Xmax;
240       if (updateTree == true) {
241         updatePtrBBox();
242         PtrTree->build(PtrBBoxVec);
243       }
244     }
245
246     //! Set class member Ymax, second argument boolean allows to
          update the
```

```
247   //! AABB tree of the tire shadow.
248   void
249   setYmax(
250     real_type _Ymax,  //!< Ymax shadow domain point
251     bool      updateTree = false  //!< Optional boolean for
          updating the tire shadow AABB tree
252   ) {
253     Ymax = _Ymax;
254     if (updateTree == true) {
255       updatePtrBBox();
256       PtrTree->build(PtrBBoxVec);
257     }
258   }
259
260   //! Get tire shadow AABB tree.
261   G2lib::AABBtree::PtrAABB const &
262   getAABBPtr(void) const
263   { return PtrTree; }
264
265   //! Get Xmin shadow domain point.
266   real_type getXmin() const { return Xmin; }
267
268   //! Get Ymin shadow domain point.
269   real_type getYmin() const { return Ymin; }
270
271   //! Get Xmax shadow domain point.
272   real_type getXmax() const { return Xmax; }
273
274   //! Get Ymax shadow domain point.
275   real_type getYmax() const { return Ymax; }
276
277   //! Clear the tire shadow domain.
278   void
279   clear(void) {
280     Xmin = quiteNaN;
281     Ymin = quiteNaN;
282     Xmax = quiteNaN;
283     Ymax = quiteNaN;
284   }
```

```
285
286   //! Print tire shadow bounding box vertices.
287   void
288   print( ostream_type & stream ) const {
289     stream
290       << "BBOX (xmin,ymin,xmax,ymax) = ( " << Xmin << ", " <<
              Ymin
291       << ", " << Xmax << ", " << Ymax << " )\n";
292   }
293
294   //! Get the mesh G2lib bounding boxes pointers vector.
295   std::vector<G2lib::BBox::PtrBBox> const &
296   getPtrBBox() const
297   { return PtrBBoxVec; }
298
299 private:
300   //! Update the mesh G2lib bounding boxes pointers vector.
301   void
302   updatePtrBBox(void) {
303     PtrBBoxVec.clear();
304     PtrBBoxVec.push_back(
305       G2lib::BBox::PtrBBox(
306         new G2lib::BBox(Xmin, Ymin, Xmax, Ymax, 0, 0)
307       )
308     );
309   }
310
311   friend class TireDisks;
312   friend class TireData;
313 };
314
315 //! Class for handling tire origin, yaw angle and camber angle
316 class Orientation {
317 private:
318   vec3      Origin;  //!< Origin position
319   real_type Yaw;     //!< Yaw angle [rad]
320   real_type Camber;  //!< Camber angle [rad]
321
322   Orientation( Orientation const & ) = delete; // costruttore
```

```
        di copia
323   Orientation & operator = ( Orientation const & ) = delete; //
          operatore di copia
324
325 public:
326
327   //! Default constructor for orientation object.
328   Orientation() {}
329
330   //! Variable set constructor for orientation object.
331   Orientation(
332     vec3    & _Origin,      //!< Origin position
333     real_type _Yaw,    //!< Yaw angle [rad]
334     real_type _Camber  //!< Camber angle [rad]
335   ) {
336     Origin = _Origin;
337     Yaw    = _Yaw;
338     Camber = _Camber;
339   }
340
341   //! Get axes versors components.
342   mat3 getXYZ(void) const;
343
344   //! Get current X-axis versor components.
345   vec3
346   getX(void)
347   { return (getXYZ() * vec3(1.0, 0, 0)).normalized(); }
348
349   //! Get current Y-axis versor components.
350   vec3
351   getY(void)
352   { return (getXYZ() * vec3(0, 1.0, 0)).normalized(); }
353
354   //! Get current Z-axis versor components.
355   vec3
356   getZ(void)
357   { return (getXYZ() * vec3(0, 0, 1.0)).normalized(); }
358
359   //! Get camber angle.
```

```
360     real_type getCamber(void) const { return Camber; }
361
362     //! Get yaw angle.
363     real_type getYaw(void) const { return Yaw; }
364
365     //! Get origin vector.
366     vec3 const & getOrigin(void) const { return Origin; }
367
368     //! Set camber angle.
369     void setCamber( real_type _Camber ) { Camber = _Camber; }
370
371     //! Set yaw angle.
372     void setYaw( real_type _Yaw ) { Yaw = _Yaw; }
373
374     //! Set camber angle.
375     void setOrigin( vec3 const & _Origin ) { Origin = _Origin; }
376
377     //! Set camber angle.
378     void set( Orientation const & in ) {
379       this->Origin = in.Origin;
380       this->Yaw    = in.Yaw;
381       this->Camber = in.Camber;
382     }
383
384     friend class TireDisks;
385   };
386
387   //! Class for handling the patch evaluation precision
388   class SolverPrecision {
389   private:
390     int_type Xdiv = 20;  //!< Number of divisions in X-axis
391     int_type Ydiv = 20;  //!< Number of divisions in Y-axis
392
393     SolverPrecision( SolverPrecision const & ) = delete; //
          costruttore di copia
394     SolverPrecision & operator = ( SolverPrecision const & ) =
          delete; // operatore di copia
395
396   public:
```

```
397     //! Default constructor for SolverPrecision object.
398     SolverPrecision() {}
399     //! Variable set constructor for SolverPrecision object.
400     SolverPrecision(
401       real_type _Xdiv, //!< Number of divisions in X-axis
402       real_type _Ydiv  //!< Number of divisions in Y-axis
403     ) {
404       Xdiv = _Xdiv;
405       Ydiv = _Ydiv;
406     }
407
408     //! Get number of divisions in X-axis.
409     real_type getXdiv() const { return Xdiv; }
410
411     //! Get number of divisions in Y-axis.
412     real_type getYdiv() const { return Ydiv; }
413
414     //! Set number of divisions in X-axis.
415     void setXdiv( real_type _Xdiv ) { Xdiv = _Xdiv; }
416
417     //! Set number of divisions in Y-axis.
418     void setYdiv( real_type _Ydiv ) { Ydiv = _Ydiv; }
419
420     void
421     set( SolverPrecision const & in ) {
422       this->Xdiv = in.Xdiv;
423       this->Ydiv = in.Ydiv;
424     }
425
426     friend class TireDisks;
427   };
428
429   //! Class for handling ETRTO tire data
430   class ETRTO {
431   private:
432     real_type SectionWidth;  //!< Tire section width[mm]
433     real_type AspectRatio;   //!< Tire aspect ratio [%]
434     real_type RimDiameter;   //!< Rim diameter [in]
435
```

```
436   ETRTO( ETRTO const & ); // costruttore di copia
437
438 public:
439   //! Default constructor for orientation object.
440   ETRTO() {}
441
442   ETRTO const &
443   operator = ( ETRTO const & rhs ) {
444     this->SectionWidth = rhs.SectionWidth;
445     this->AspectRatio  = rhs.AspectRatio;
446     this->RimDiameter  = rhs.RimDiameter;
447     return *this;
448   }
449
450   //! Variable set constructor for tire object.
451   ETRTO(
452     real_type _SectionWidth,  //!< Tire section width[mm]
453     real_type _AspectRatio,   //!< Tire aspect ratio [%]
454     real_type _RimDiameter    //!< Rim diameter [in]
455   ) {
456     SectionWidth = _SectionWidth;
457     AspectRatio  = _AspectRatio;
458     RimDiameter  = _RimDiameter;
459   }
460
461   //! Get sidewall height [m].
462   real_type
463   getSidewallHeight(void) const
464   { return SectionWidth / 1000.0 * AspectRatio / 100; }
465
466   //! Get external tire diameter [m].
467   real_type
468   getTireDiameter(void) const
469   { return RimDiameter * 0.0254 + getSidewallHeight() * 2.0; }
470
471   //! Get section width [m].
472   real_type
473   getSectionWidth(void) const
474   { return SectionWidth / 1000.0; }
```

```
475
476   //! Display tire data.
477   void
478   print( ostream_type & stream ) const {
479     stream
480       << "Current Tire Data:\n"
481       << "\tSection width = " << SectionWidth << " mm\n"
482       << "\tAspect ratio  = " << AspectRatio << " %\n"
483       << "\tRim diameter  = " << RimDiameter << " in\n"
484       << "\tS.wall Height = " << getSidewallHeight() * 1000 <<
          " mm\n"
485       << "\tTire diameter = " << getTireDiameter() * 1000 << "
          mm\n\n";
486   }
487 };
488
489 //! Class for evaluating the contact patch
490 class TireDisks {
491 private:
492   std::vector<Disk> DiskVector;  //! Disks instance vector
493   ETRTO             TireDenom;   //!< ETRTO tire denomination
       object
494   SolverPrecision   Precision;   //!< Solver precision object
495   Orientation       Orient;      //!< Orientation object
496   Shadow            iShadow;     //!< Shadow bounding box
       object
497   RDF::MeshSurface  iMesh;       //! RDF mesh object pointer
498   std::vector<std::shared_ptr<RDF::Triangle3D> >
       intersectionTriPtr; //!< Local intersected triangles
       vector
499   Eigen::MatrixXd      intersectionGrid;  //! Local sampling
       grid
500
501   TireDisks( TireDisks const & ); // costruttore di copia
502   TireDisks const & operator = ( TireDisks const & ); //
       operatore di copia
503
504 public:
505   TireDisks(
```

```
506       ETRTO           const & _TireDenom,
507       SolverPrecision const & _Precision
508     ) {
509       TireDenom        = _TireDenom;
510       Precision.set(_Precision);
511       intersectionGrid = Eigen::MatrixXd( Precision.Ydiv + 1,
              Precision.Xdiv + 1 );  // Y sampling (rows), X sampling
              (columns)
512       std::vector<real_type> Dvec = offsetDisk(Precision.Ydiv);
513       real_type R                 = TireDenom.getTireDiameter() /
              2.0;
514       for ( int_type i = 0; i <= Precision.Ydiv; ++i )
515         DiskVector.push_back( Disk( vec2(0, 0), Dvec[i], R) );
516     }
517
518     //! Set the tire orientation
519     void
520     setOrientation( Orientation const & _Orient ) {
521       Orient.set(_Orient);
522       updateShadow();
523       //updateIntersectionList();
524     }
525
526     //! Set the terrain mesh
527     void
528     setMesh( RDF::MeshSurface const & _iMesh ) {
529       iMesh.set(_iMesh);
530     }
531
532     //! Get orientation information
533     Orientation const &
534     getOrientation(void) const
535     { return Orient; }
536
537     //! Get orientation information
538     SolverPrecision const &
539     getPrecision(void) const
540     { return Precision; }
541
```

```
542     //! Get grid step on X-axis
543     real_type
544     getXstep(void) const
545     { return TireDenom.getTireDiameter() / Precision.Xdiv; }
546
547     //! Get grid step on Y-axis
548     real_type
549     getYstep(void) const
550     { return TireDenom.getSectionWidth() / Precision.Ydiv; }
551
552     //! Get i-th intantiated disk
553     Disk const &
554     getithDisk( int_type i ) const
555     { return DiskVector[i]; }
556
557     //! Perform triangles sampling
558     void
559     gridSampling( bool print = false );
560
561     //! Find single point intersection between tire and mesh (
            Pacejka Single
562     //! Contact Point) with AABB tree
563     real_type
564     MF_Pacejka_SCP( bool print = false );
565
566     //! Evaluate the contact patch between tire and mesh (Magic
            Formula
567     //! Swift Contact Patch Evaluation)
568     real_type
569     MF_Swift_PE( bool print = false );
570
571     //! Evaluate the local effective road plane (Magic Formula
572     //!//! Swift Effective road plane)
573     std::vector<real_type>
574     MF_Swift_ERP( bool print = false );
575
576     void
577     Move(
578       vec3      const & start,   //! Starting position
```

```
579       vec3      const & arrival, //! Arrival position
580       real_type const & freq,    //!< Sampling frequency [Hz]
581       real_type const & speed,   //!< Tire speed [m/s]
582       bool              print
583     );
584
585     //! Update the local intersected triangles list
586     void
587     updateIntersectionList(void);
588
589   private:
590     std::vector<real_type>
591     offsetDisk( int_type n );
592
593     //! Find the rectangular shadow domain of the tire in X and Y
          -axis
594     // if true print the new shadow domain
595     void
596     updateShadow( bool print = false );
597
598
599
600     //! Find nearest intersection to origin
601     real_type
602     calculateMagnitude( std::vector<vec3> const &
          IntersectionPointVec );
603   };
604
605 }  // namespace PatchTire
606
607 ///
608 /// eof: PatchTire.hh
609 ///
```

B.4 PatchTire.cc

```
1 #include "PatchTire.hh"  // RDF file extention Loader
2
3 //! Tire computations routine
```

```
4 namespace PatchTire {
5
6  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
7  // namespace algorithms
8  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
9
10 //! Holds all of the algorithms needed for the contact patch
       evaluation
11 namespace algorithms {
12
13   //! Check if a ray hits a triangle object through Möller-
         Trumbore
14   //! intersection algorithm
15   bool
16   rayIntersectsTriangle(
17     vec3 const      & RayOrigin,        //!< Ray origin
           position
18     vec3 const      & RayDirection,     //!< Ray direction
           vector
19     RDF::Triangle3D & Triangle,         //! Triangle object
20     vec3            & IntersectionPoint //! Intersection point
21   ) {
22     vec3      E1  = Triangle.getithVertex(1) - Triangle.
           getithVertex(0);
23     vec3      E2  = Triangle.getithVertex(2) - Triangle.
           getithVertex(0);
24     vec3      A   = RayDirection.cross(E2);
25     real_type det = A.dot(E1);
26     real_type t_param;
27
28     if ( det > epsilon ) {
29       vec3      T = RayOrigin - Triangle.getithVertex(0);
30       real_type u = A.dot(T);
31       if ( u < 0.0 || u > det ) return false;
32       vec3      B = T.cross(E1);
33       real_type v = B.dot(RayDirection);
34       if ( v < 0.0 || u + v > det ) return false;
```

```
35        t_param = (B.dot(E2))/det;
36      } else if ( det < -epsilon ) {
37        vec3      T = RayOrigin - Triangle.getithVertex(0);
38        real_type u = A.dot(T);
39        if ( u > 0.0 || u < det ) return false;
40        vec3      B = T.cross(E1);
41        real_type v = B.dot(RayDirection);
42        if ( v > 0.0 || u + v < det ) return false;
43        t_param = (B.dot(E2))/det;
44      } else {
45        return false;
46      }
47      // At this stage we can compute t to find out where the
           intersection
48      // point is on the line
49      if ( t_param >= 0 ) { // ray intersection
50        IntersectionPoint = RayOrigin + RayDirection * t_param;
51        return true;
52      } else {
53        // This means that there is a line intersection on
             negative side
54        return false;
55      }
56    }
57
58    // Find the points of intersection.
59    int_type
60    segmentIntersectsCircle(
61      vec2 const & Origin,
62      real_type    R,
63      vec2 const & Point_1,
64      vec2 const & Point_2,
65      vec2       & Intersect_1,
66      vec2       & Intersect_2
67    ) {
68      real_type t_param;
69
70      vec2      d   = Point_2 - Point_1;
71      vec2      P1O = Point_1 - Origin;
```

```
72      real_type A   = d.dot(d);
73      real_type B   = 2 * d.dot(P1O);
74      real_type C   = P1O.dot(P1O) - R*R;
75      real_type det = B*B - 4 * A * C;
76      if ( A <= epsilon || det < 0 ) {
77        // No real solutions
78        Intersect_1 = vec2(quiteNaN, quiteNaN);
79        Intersect_2 = vec2(quiteNaN, quiteNaN);
80        return 0;
81      } else if ( det == 0.0 ) {
82        // One solution
83        t_param     = -B / (2*A);
84        Intersect_1 = Point_1 + t_param * d;
85        Intersect_2 = vec2(quiteNaN, quiteNaN);
86        return 1;
87      } else {
88        // Two solutions
89        t_param = (-B + std::sqrt(det)) / (2 * A);
90        Intersect_1 = Point_1 + t_param * d;
91        t_param = (-B - std::sqrt(det)) / (2 * A);
92        Intersect_2 = Point_1 + t_param * d;
93        return 2;
94      }
95    }
96
97    bool
98    pointInsideCircle(
99      vec2 const & Origin,
100     real_type    R,
101     vec2 const & Point
102   ) {
103     // Compare radius of circle with distance
104     // of its center from given point
105     vec2 PO = Point - Origin;
106     return PO.dot(PO) <= R*R;
107   }
108
109   bool
110   pointOnSegment(
```

```
111       vec2 const & Point_1,
112       vec2 const & Point_2,
113       vec2 const & Point
114     ) {
115       // A and B are the extremities of the current segment C is
              the point to
116       // check
117
118       // Create the vector AB
119       vec2 AB = Point_2 - Point_1;
120       // Create the vector AC
121       vec2 AC = Point - Point_1;
122
123       // Compute the cross product of AB and AC
124       // Check if the three points are aligned (cross product is
              null)
125       // if ((AB.cross3(AC)).squaredNorm() > epsilon)
126       // return false;
127
128       // Compute the dot product of vectors
129       real_type KAC = AB.dot(AC);
130       if ( KAC      < -epsilon ) return false;
131       if ( abs(KAC) < epsilon  ) return true;
132
133       // Compute the square of the segment lenght
134       real_type KAB = AB.dot(AB);
135       if ( KAC > KAB ) return false;
136       if ( abs(KAC - KAB) < epsilon ) return true;
137
138       // The point is on the segment
139       return true;
140     }
141   }  // namespace algorithms
142
143   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          - - - - - - -
144   // class Disk
145   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          - - - - - - -
```

```
146
147 //! Calculate area between disk and segment
148 void
149 Disk::updatePatchLength(void) {
150   // Reset class variable
151   PatchLength = 0.0;
152
153   for ( unsigned i = 0; i < PointsSequence.size() - 1; ++i ) {
154     vec2     Intersection_1, Intersection_2;
155     vec2     Point_1 = PointsSequence[i];
156     vec2     Point_2 = PointsSequence[i + 1];
157     int_type Type    = algorithms::segmentIntersectsCircle(
158       OriginXZ, R, Point_1, Point_2, Intersection_1,
          Intersection_2
159     );
160     // std::cout << " " << Type;
161     if ( Type == 0 ) {
162       // No contact points, the line segment is not into the
          Disk
163       continue;
164     } else if (Type == 1) {
165       // Tangent, no length added
166       continue;
167     } else if (Type == 2) {
168       // Check whether the two segment points are into the
          circle
169       bool Pose_pt1 = algorithms::pointInsideCircle(OriginXZ, R
          , Point_1);
170       bool Pose_pt2 = algorithms::pointInsideCircle(OriginXZ, R
          , Point_2);
171
172         // Check whether the two intersection points are onto
            the line segment
173       bool Pose_int1 = algorithms::pointOnSegment(Point_1,
          Point_2, Intersection_1);
174       bool Pose_int2 = algorithms::pointOnSegment(Point_1,
          Point_2, Intersection_2);
175
176       // Cases
```

```
177         // Line segment Point_1 and line segment Point_2 into the
                circle,
178         // intersection points oustide line segment
179         if ( Pose_pt1 && Pose_pt2 && !Pose_int1 && !Pose_int2 ) {
180           PatchLength += (Point_2 - Point_1).norm();
181           continue;
182         }
183         // Intersection points into the line segment and line
                segment points
184         // outside the circle
185         else if ( !Pose_pt1 && !Pose_pt2 && Pose_int1 &&
                Pose_int2 ) {
186           PatchLength += (Intersection_2 - Intersection_1).norm()
                  ;
187           continue;
188         }
189         // Line segment Point_1 outside the circle, line segment
                Point_2
190         // inside the circle
191         else if ( !Pose_pt1 && Pose_pt2 ) {
192           if ( Pose_int1 && !Pose_int2 ) {
193             // Add length from Intersection_1 to Point_2
194             PatchLength += (Intersection_1 - Point_2).norm();
195             continue;
196           } else if ( Pose_int2 && !Pose_int1 ) {
197             // Add length from Intersection_2 to Point_2
198             PatchLength += (Intersection_2 - Point_2).norm();
199             continue;
200           }
201         }
202         // Line segment Point_1 inside the circle, line segment
                Point_2
203         // outside the circle
204         else if ( Pose_pt1 && !Pose_pt2 ) {
205           if ( Pose_int1 && !Pose_int2 ) {
206             // Add length from Intersection_1 to Point_1
207             PatchLength += (Intersection_1 - Point_1).norm();
208             continue;
209           } else if ( !Pose_int1 && Pose_int2 ) {
```

```
210             // Add length from Intersection_2 to Point_1
211             PatchLength += (Intersection_2 - Point_1).norm();
212             continue;
213           }
214         }
215       }
216     }
217     // std::cout << "Length: " << PatchLength << std::endl;
218   }
219
220   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - - - - -
221   // class Orientation
222   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - - - - -
223
224   //! Get axes versors components
225   mat3
226   Orientation::getXYZ(void) const {
227     // Transformation matrix for Z-axis rotation
228     mat3 Rot_Z;
229     Rot_Z << cos(Yaw), -sin(Yaw), 0,
230              sin(Yaw),  cos(Yaw), 0,
231                     0,        0, 1;
232     // Transformation matrix for X-axis rotation
233     mat3 Rot_X;
234     Rot_X << 1,           0,           0,
235              0, cos(Camber), -sin(Camber),
236              0, sin(Camber),  cos(Camber);
237
238     return Rot_Z * Rot_X;
239   }
240
241   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - - - - -
242   // class TireDisks
243   // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        - - - - - - -
244
```

```
245 void
246 TireDisks::gridSampling(bool print) {
247
248   // Orient the sampling
249   mat3 MatrixInv = Orient.getXYZ().inverse();
250
251   // Storing indexers
252   unsigned i = 0;
253   unsigned j = 0;
254   std::vector<vec2> PointsSequence;
255
256   for ( real_type y = -TireDenom.getSectionWidth() / 2.0;
257         y <= TireDenom.getSectionWidth() / 2.0;
258         y += getYstep(), ++j ) {
259     for ( real_type x = -TireDenom.getTireDiameter() / 2.0;
260           x <= TireDenom.getTireDiameter() / 2.0;
261           x += getXstep(), ++i ) {
262       // Update ray center in tire coordinates
263       vec3 curCenter = Orient.getOrigin() + MatrixInv * vec3(x,
              y, 0);
264
265       for ( unsigned t = 0; t < intersectionTriPtr.size(); ++t
             ) {
266         vec3 IntersectionPoint;
267         bool intersection = algorithms::rayIntersectsTriangle(
268           Orient.getOrigin(),
269           -Orient.getZ(),  // careful to the minus sign !!!
270           *intersectionTriPtr[t],
271           IntersectionPoint
272         );
273
274         if ( intersection ) {
275           // Store results
276           real_type z = (IntersectionPoint - curCenter).norm();
277           intersectionGrid(j, i) = z;  // Y sampling (rows), X
                 sampling (columns)
278           PointsSequence.push_back(vec2(x, z));
279           break;
280         } else {
```

```
281             intersectionGrid(j, i) = -1.0;  // Y sampling (rows),
                    X sampling (columns)
282           }
283         }
284       }
285       // Calculate chain length inside the circle
286       DiskVector[j].setPointsSequence(PointsSequence);
287
288       if ( print ) {
289         std::cout
290           << "Disk " << j << " of " << DiskVector.size() - 1
291           << " -> Chain: ";
292         for ( unsigned k = 0; k < PointsSequence.size(); ++k )
293           std::cout << PointsSequence[k][1] << " ";
294         std::cout << std::endl;
295       }
296       PointsSequence.clear();
297
298       // Update indexer
299       i = 0;
300     }
301     if ( print ) std::cout << std::endl << intersectionGrid <<
          std::endl;
302   }
303
304   //! Find single point intersection between tire and mesh (
       Pacejka Single
305   //! Contact Point) with AABB tree
306   real_type
307   TireDisks::MF_Pacejka_SCP(bool print) {
308     // Ray-Triangle intesection Point
309     std::vector<vec3> IntersectionPointVec;
310     real_type Magnitude = quiteNaN;
311
312     vec3 IntersectionPoint;
313     for ( unsigned i = 0; i < intersectionTriPtr.size(); ++i ) {
314       if ( algorithms::rayIntersectsTriangle(
315              Orient.getOrigin(), -Orient.getZ(),  // careful to
                    the minus sign !!!
```

```
316             *intersectionTriPtr[i], IntersectionPoint
317           ) ) {
318         // Store intersection point in proper vector
319         IntersectionPointVec.push_back(IntersectionPoint);
320       }
321     }
322     // Find nearest intersection to origin
323     Magnitude = calculateMagnitude(IntersectionPointVec);
324
325     // Display information
326     if ( print )
327       std::cout
328         << "Single contact point for Pacejka MF -> "
329         << IntersectionPointVec.size() << " intersections found"
330         << "\n\tInt. Point  = X " << IntersectionPoint[0]
331         << ", Y " << IntersectionPoint[1]
332         << ", Z " << IntersectionPoint[2]
333         << "\n\tMagnitude   = " << Magnitude
334         << "\n\n";
335     return Magnitude;
336   }
337
338   //! Evaluate the contact patch between tire and mesh (Magic
       Formula
339   //! Swift Contact Patch Evaluation)
340   real_type
341   TireDisks::MF_Swift_PE(bool print) {
342     real_type PatchArea = 0.0;
343     real_type Ystep     = getYstep();
344     for ( unsigned i = 0; i < DiskVector.size(); ++i )
345       PatchArea += DiskVector[i].getPatchLength() * Ystep;
346
347     if ( print )
348       std::cout
349         << "Contact Patch Area for Swift MF -> "
350         << PatchArea * pow(10, 6) << " mm^2\n\n";
351     return PatchArea;
352   }
353
```

```
354 //! Evaluate the local effective road plane (Magic Formula
355 //! Swift Effective road plane)
356 std::vector<real_type>
357 TireDisks::MF_Swift_ERP( bool print ) {
358   int_type nrows = int_type( intersectionGrid.rows() );
359   int_type ncols = int_type( intersectionGrid.cols() );
360
361   real_type angle_RotY  = 0.0;
362   real_type deltaY_RotY = 0.0;
363
364   real_type angle_RotX  = 0.0;
365   real_type deltaY_RotX = 0.0;
366
367   real_type deltaX_RotY = TireDenom.getTireDiameter();
368   real_type deltaX_RotX = TireDenom.getSectionWidth();
369
370   for ( int_type i = 0; i < nrows; ++i ) {
371     deltaY_RotY = intersectionGrid(i, 0) - intersectionGrid(i,
          ncols - 1);
372     angle_RotY += atan2(deltaY_RotY, deltaX_RotY);
373   }
374   angle_RotY /= nrows;
375
376   for (int_type i = 0; i < ncols; i++) {
377     deltaY_RotX = intersectionGrid(0, i) - intersectionGrid(
          nrows - 1, i);
378     angle_RotX += atan2(deltaY_RotX, deltaX_RotX);
379   }
380   angle_RotX /= ncols;
381
382   if ( print )
383     std::cout
384       << "Effective Road Plane for Swift MF -> "
385       << "X: " << angle_RotX * 180 / G2lib::m_pi << "°, "
386       << "Y: " << angle_RotY * 180 / G2lib::m_pi << "°\n\n";
387
388   return std::vector<real_type>(angle_RotX, angle_RotY);
389 }
390
```

```
391 void
392 TireDisks::Move(
393   vec3      const & start,    //! Starting position
394   vec3      const & arrival,  //! Arrival position
395   real_type const & freq,     //!< Sampling frequency [Hz]
396   real_type const & speed,    //!< Tire speed [m/s]
397   bool              print
398 ) {
399   // Set current position
400   vec3 curpos = start;
401
402   // Set and initialize orientation
403   real_type Yaw    = 0;
404   real_type Camber = 0;
405
406   real_type nstep = (arrival - start).norm() / speed * freq;
407   vec3 step       = (arrival - start) / nstep;
408
409   // Start chronometer
410   auto start_move = std::chrono::system_clock::now();
411
412   for ( unsigned i = 0; i < nstep; ++i ) {
413     // Set current orientation
414     Orientation Pose(curpos, Yaw, Camber);
415     setOrientation(Pose);
416
417     // Evaluate Single Contact Point
418     MF_Pacejka_SCP(print);
419
420     // Perform grid sampling
421     gridSampling(false);
422
423     // Evaluate Patch
424     MF_Swift_PE(print);
425
426     // Evaluate Effective Road Plane
427     MF_Swift_ERP(print);
428
429     // Update current position
```

```
430       curpos += step;
431     }
432     // Stop chronometer
433     auto end_move = std::chrono::system_clock::now();
434
435     // This constructs a duration object using milliseconds
436     auto elapsed_move = std::chrono::duration_cast<std::chrono::
           microseconds>(
437           end_move - start_move);
438     std::cout
439       << "Execution time = " << elapsed_move.count() / 1000.0 <<
           " ms\n"
440       << "Step execution time = "
441       << (elapsed_move.count() / 1000.0) / nstep << " ms\n";
442     }
443
444     std::vector<real_type>
445     TireDisks::offsetDisk( int_type n ) {
446       std::vector<real_type> offsetVec;
447       for ( int_type i = 0; i < n; ++i )
448         // Index from Y positive to Y negative
449         offsetVec.push_back( TireDenom.getSectionWidth() / 2.0 -
450                              TireDenom.getSectionWidth() * i / n
                                    );
451       return offsetVec;
452     }
453
454     //! Find the rectangular shadow domain of the tire in X and Y
           -axis
455     void
456     TireDisks::updateShadow( bool print ) {
457       // Calculate maximum covered space
458       real_type diagonal = hypot( TireDenom.getSectionWidth(),
459                                   TireDenom.getTireDiameter() ) /
                                        2;
460
461       // Increment shadow to take in account camber angle
462       real_type inc = 1.1;
463
```

```
464       // Set new tire shadow domain
465       iShadow.setXmax(Orient.Origin[0] + inc * diagonal);
466       iShadow.setXmin(Orient.Origin[0] - inc * diagonal);
467       iShadow.setYmax(Orient.Origin[1] + inc * diagonal);
468       iShadow.setYmin(Orient.Origin[1] - inc * diagonal, true);
469
470       // Print the new shadow domain
471       if ( print ) iShadow.print(std::cout);
472     }
473
474     //! Update the local intersected triangles list
475     void
476     TireDisks::updateIntersectionList(void) {
477       G2lib::AABBtree::VecPairPtrBBox intersectionList;
478       intersectionTriPtr.clear();
479       (*iMesh.getAABBPtr()).intersect(*iShadow.getAABBPtr(),
             intersectionList);
480       std::cout << intersectionList.size() << ", ";
481       for ( unsigned i = 0; i < intersectionList.size(); ++i ) {
482         intersectionTriPtr.push_back(
483           iMesh.ithTrianglePtr((*intersectionList[i].first).Id())
484         );
485         // std::cout << (*intersectionList[i].first).Id() << ",
               ";
486       }
487     }
488
489     //! Find nearest intersection to origin
490     real_type
491     TireDisks::calculateMagnitude(
492       std::vector<vec3> const & IntersectionPointVec
493     ) {
494       real_type iMagnitude = quiteNaN;
495       real_type Magnitude  = quiteNaN;
496       for ( unsigned i = 0; i < IntersectionPointVec.size(); ++i
             ) {
497         iMagnitude = TireDenom.getTireDiameter() / 2.0 -
498                      (IntersectionPointVec[i] - Orient.getOrigin
                        ()).norm();
```

```
499         // std::cout << "iMagnitude " << i << ": " << iMagnitude
                << std::endl;
500         if ( i == 0 ) {
501           Magnitude = iMagnitude;
502           continue;
503         } else {
504           if (iMagnitude > Magnitude)
505             Magnitude = iMagnitude;
506         }
507       }
508       return Magnitude;
509     }
510
511   }  // namespace algorithms
```

Tests Code C

C.1 Computational Geometry Tests

C.2 Contact Patch Evaluation Tests

Bibliografia

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

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

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

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

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

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

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

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

[9] A. J. C. Schmeitz. "A semi-empirical, three-dimensional, tyre model for rolling over arbitrary road unevennesses". Tesi di dott. Technische Universiteit Delft, 2004.

[10] A. J. C. Schmeitz, I. J. M. Besselink e S. T. H. Jansen. “TNO MF-SWIFT”. In: *Vehicle System Dynamics* 45.sup1 (2007), pp. 121–137. DOI: 10.1080/00423110701725208. eprint: https://doi.org/10.1080/00423110701725208. URL: https://doi.org/10.1080/00423110701725208.