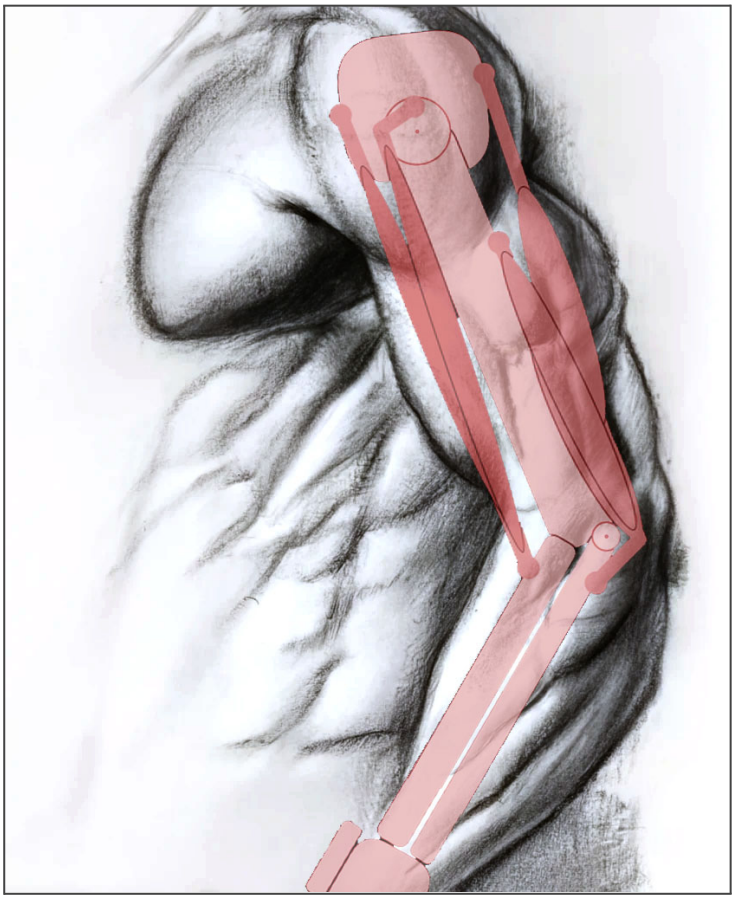
Implementació d'un Model Computacional de Sistemes Musculoesquelètics



Pseudònim: **Castanyer**

Projecte de Recerca - 2023-2024

Agraïments

Aquest treball de recerca ha estat possible gràcies a l'ajuda i assessorament de diverses persones.

En primer lloc, voldria donar les gràcies a la meva tutora pel seu ajut i paciència al llarg d'aquesta recerca, especialment per tots els seus comentaris i valuosos suggeriments.

En segon lloc, al meu pare pel seu assessorament tècnic, tant per ajudar-me a resoldre alguns problemes de programació que han sortit, com per orientar-me a l'hora de plantejar l'arquitectura del programa.

Finalment, a la meva mare per ajudar-me a millorar l'expressió i la llegibilitat d'aquest treball, mitjançant la seva dedicació a múltiples revisions.

Resum

El sistema musculoesquelètic és el conjunt format per músculs, ossos i els elements estructurals que els uneixen, que treballen en conjunt per moure i suportar un cos. Donat que el múscul és l'únic element actiu, el seu funcionament és el més complex de modelar. Per això en l'àmbit de la fisiologia s'han proposat diferents models matemàtics per descriure el seu comportament. El més conegut és el model de Hill, que modela el múscul com una combinació de tres elements: un contràctil, un elàstic en sèrie i un elàstic en paral·lel.

L'objectiu d'aquest treball és implementar un model computacional que permeti simular mecànicament models musculoesquelètics. Per fer-ho s'ha desenvolupat un *framework*, programat amb *Processing*, que a més d'incloure un objecte que modela un múscul, implementant el model de Hill, també inclou la resta d'elements passius necessaris per definir i simular sistemes musculoesquelètics. A més, per poder avaluar millor el *framework*, s'ha programat una interfície gràfica que permet interactuar amb la simulació, controlant el nivell d'activació dels músculs i observant en temps real com varien les seves propietats internes.

Finalment, aquesta eina s'ha utilitzat per dur a terme quatre experiments que comparen els resultats simulats amb els comportaments típics de músculs reals. Els resultats de la comparació han estat molt positius, concloent que la simulació s'aproxima molt a la realitat. Tot el codi desenvolupat està disponible sota una llicència de codi obert.

Abstract

The musculoskeletal system is a group made up of muscles, bones and the structural elements that join them, which work together to move and support the body. Given that the muscle is the only active element, its behaviour is the most complex to model. For this reason, in the field of physiology, different mathematical models have been proposed to describe how it works. The most used one is Hill's model, which models the muscle as an ensemble of three elements: a contractile, a series elastic and a parallel elastic one.

The goal of this work is to implement a computational model that simulates mechanical musculoskeletal models. To do this, a framework, coded in *Processing*, has been developed that includes an object that models a muscle, using Hill's model, and the rest of elements necessary to define and simulate musculoskeletal systems. In addition, in order to better evaluate the framework, a graphical interface has been programmed that allows for interaction with the simulation, controlling the activation level of the muscles and assessing in real time how their internal properties vary.

Finally, this tool has been used to carry out four experiments that compare the simulated results with the ones of real muscles. The results of the comparison have been very positive, meaning that the simulation is very realistic. All the developed code is available under an open source licence.

Keywords: *Musculoskeletal System*, *Processing*, *Hill's Model*, *Computational Physics, Simulation*

Resumen

El sistema músculo-esquelético es el conjunto formado por los músculos, huesos y elementos estructurales que los unen, que trabajan en conjunto para mover y soportar un cuerpo. Dado que el músculo es el único elemento activo, su funcionamiento es el más complejo de modelar. Por eso, en el ámbito de la fisiología se han propuesto diferentes modelos matemáticos para describir su comportamiento. El más conocido es el modelo de Hill, que modela el músculo como una combinación de tres elementos: uno contráctil, uno elástico en serie y uno elástico en paralelo.

El objetivo de este trabajo es implementar un modelo computacional que permita simular mecánicamente modelos músculo-esqueléticos. Para ello se ha desarrollado un framework, programado con Processing, que además de incluir un objeto que modela un músculo, implementando el modelo de Hill, también incluye el resto de elementos pasivos necesarios para definir y simular sistemas músculo-esqueléticos. Además, para poder evaluar mejor el framework, se ha programado una interfaz gráfica que permite interactuar con la simulación, controlando el nivel de activación de los músculos y observando en tiempo real cómo varían sus propiedades internas.

Por último, esta herramienta se ha utilizado para llevar a cabo cuatro experimentos que comparan los resultados simulados con los comportamientos típicos de músculos reales. Los resultados de la comparación han sido muy positivos, concluyendo que la simulación se aproxima mucho a la realidad. Todo el código desarrollado está disponible bajo una licencia de código abierto.

Índex

Introducció

A l'hora de triar un tema pel treball de recerca vaig voler centrar-me en una gran afició que tinc des de fa uns anys: l'exercici físic i el desenvolupament muscular. Malgrat això, no vaig trobar cap tema que em cridés prou l'atenció.

Per això, vaig considerar ampliar-lo cap a la mecànica, una àrea que també m'interessa. Això va donar lloc a què l'àmbit final sigui sobre la biomecànica del sistema musculoesquelètic. Aquesta àrea em semblava interessant però no sabia com concretar-la.

Per aquesta raó vaig voler afegir-hi una tercera àrea: la programació. Una àrea amb la qual estic molt familiaritzat pel meu interès des de primària. Així doncs, finalment vaig poder delimitar el meu treball de recerca al tema dels models computacionals musculoesquelètics.

Trobo aquest tema realment interessant. Tot i que havia llegit sobre el sistema muscular, la immensa majoria d'informació que he rebut és sobre l'aspecte biològic. Això havia creat un buit en el meu coneixement sobre l'aspecte mecànic del cos, que em seria beneficiós omplir. A més, programar un model computacional m'interessava perquè realitzar un projecte d'aquesta mida i complexitat em servirà per aprofundir i millorar els meus coneixements de programació.

La metodologia del treball es va plantejar en dues fases:

- Una primera teòrica on faria recerca bibliogràfica sobre les dues àrees principals: models musculoesquelètics i biomecànica, i motors físics computacionals per simular sistemes mecànics.

- I una segona d'implementació pràctica on crearia un model musculoesquelètic computacional. Com que era un tema desconegut per mi no hi havia certesa de fins a on podria arribar amb el límit de temps existent;

va semblar bona idea plantejar una sèrie d'objectius progressius de complexitat creixent.

Finalment es van definir els següents objectius:

- Aprendre sobre models musculoesquelètics i sistemes biomecànics
- Aprendre sobre motors computacionals físics de simulació mecànica
- Programar un model computacional del múscul
- Programar un *framework* per simular models musculoesquelètics
- Programar una demostració interactiva d'un sistema muscular

I.
Recerca teòrica

1. El sistema musculoesquelètic

1.1. Què és el sistema musculoesquelètic?

El sistema musculoesquelètic és el conjunt de músculs i ossos que treballen junts per moure i suportar el cos. També el componen tendons, lligaments i fàscies. Els músculs generen força, els ossos donen suport i els tendons, lligaments i fàscies connecten ossos i músculs.

1.2. Els músculs

Els músculs són òrgans que poden generar força. Un múscul es contrau amb l'estimulació, i es relaxa quan l'estimulació s'atura. També pot ser estirat sense lesionar-se i s'adapta a l'ambient mitjançant l'atròfia i la hipertròfia.

Els músculs del cos es poden agrupar en 3 tipus: estriat, visceral i cardíac. Els músculs estriats són els únics músculs que es poden controlar voluntàriament. El moviment del múscul visceral és involuntari i es troba dins òrgans com l'estómac i venes. Finalment, el múscul cardíac es troba només al cor i es controla involuntàriament.

1.2.1. Tipus de músculs

1.2.1.1 Músculs viscerals

El múscul visceral **[Figura 1.1]**, també anomenat múscul llis, és un tipus de múscul que es controla involuntàriament. La contracció s'estimula mitjançant el sistema

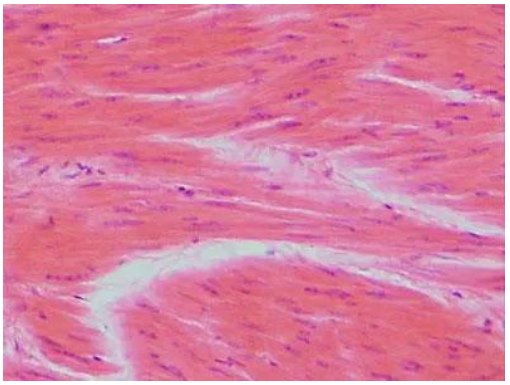


Figura 1.1 - Imatge microscòpica del múscul visceral - *www.britannica.com /science/smooth-muscle*

nerviós. Es troba en òrgans com l'estómac, els intestins i les venes.

Aquests músculs, a diferència dels músculs estriats, estan formats per filaments de només actina. Aquestes fibres s'uneixen en grup anomenats cossos densos. Els cossos densos poden agrupar-se en músculs viscerals multiunitaris, es poden trobar per exemple en l'iris i la tràquea, però també poden trobar-se sols en músculs viscerals unitaris, per exemple en l'úter i en les venes.

1.2.1.2 Músculs cardíacs

El múscul cardíac **[Figura 1.2]** és un tipus de múscul que només es troba en el cor, i es diferencia dels altres en què és l'únic que no utilitza el sistema nerviós per controlar la seva contracció. Sinó que és controlat per les cèl·lules marcapassos que dicten el batec del cor. Aquesta és la raó perquè sovint el cor continua batejant després d'una lesió al sistema nerviós.

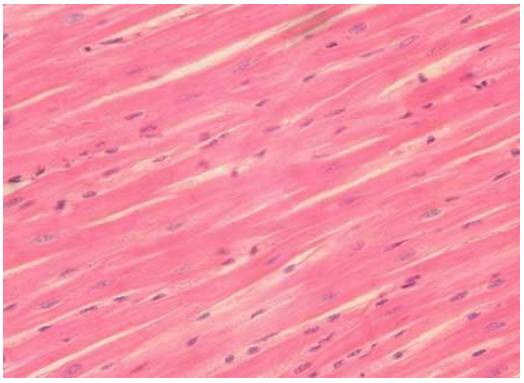


Figura 1.2 - Imatge microscòpica del múscul cardíac - *www.eugraph.com /histology/cardiova/cardiac.html*

1.2.1.3. Músculs estriats

El múscul estriat **[Figura 1.3]**, també anomenat esquelètic, és l'únic múscul que es controla voluntàriament. És el tipus de múscul més comú. Hi ha 430 parelles d'aquest tipus de múscul pel cos i el seu pes contribueix entre el 40 i el 45% de tot el pes del cos.

Aquest múscul pot realitzar contraccions isotòniques, isomètriques i isocinètiques:

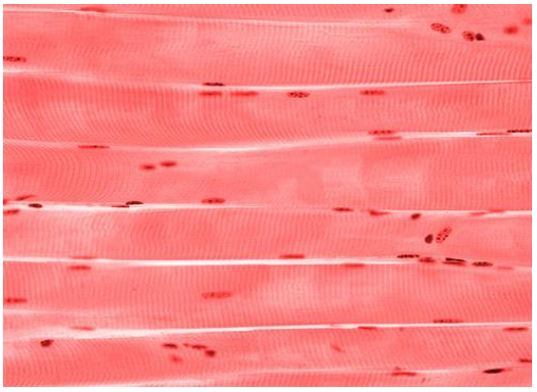


Figura 1.3 - Imatge microscòpica del múscul estriat - *www.teaching.ncl.ac.uk /bms/wiki/index.php/Skeletal_muscle*

- Les contraccions **isotòniques** mantenen la tensió del múscul constant però varia la llargada i la velocitat del moviment; es fan servir en esforços dinàmics.

- En les contraccions **isomètriques** la llargada del múscul és constant però la velocitat i la tensió del múscul varia; aquestes contraccions es fan servir en esforços estàtics quan l'esforç generat és menor que la càrrega.

- Finalment, les contraccions **isocinètiques** mantenen la velocitat de contracció del múscul constant però varia la tensió i la llargada; aquestes contraccions són les més eficients en la hipertròfia dels músculs.

Aquest tipus de múscul, rodejat per una fàscia, és l'únic que està format per fibres musculars d'actina i miosina, amb cèl·lules posicionades paral·lelament.

1.2.2. Fibres musculars estriades

Les fibres musculars són el tipus de cèl·lula que formen els músculs. Així doncs, les fibres musculars estriades són les cèl·lules que formen els músculs estriats i, per tant, que es poden contraure voluntàriament.
Les fibres musculars contenen les miofibril·les (gruix 10-100 µm, llargada 1-30 cm). Estan alineades paral·lelament dins del sarcoplasma de la fibra muscular, que és líquid que omple les cèl·lules musculars. Cada miofibril·la (diàmetre 1 µm) està formada per una cadena de feixos de sarcòmers, els quals són les unitats bàsiques que generen la força muscular [Figura 1.4].

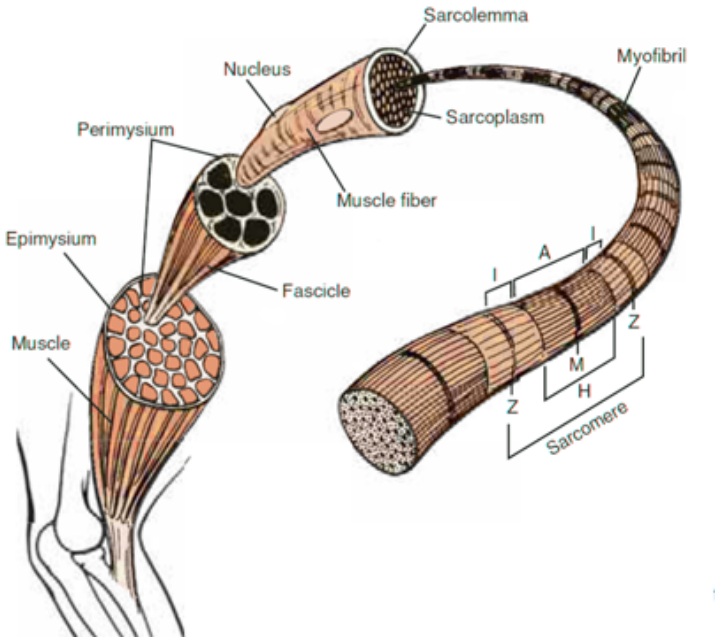


Figura 1.4 - Estructura d'un múscul. Múscul → Fascicle → Fibra Muscular → Miofibril·les → Sarcòmers. - *[Lorenz, T., & Campello, M. 2012]*

1.2.2.1. Estructura dels sarcòmers

El sarcòmer (llargada 2 µm) és una estructura composta per les proteïnes: actina, miosina i titina **[Figura 1.5]**. Les dues primeres són la part contràctil del sarcòmer, i l'altra és part del citoesquelet que les uneix **[Stromer. 1998]**.

L'**actina** està en forma d'una doble hèlix molt prima (diàmetre 5 nm). Un cantó de l'hèlix està connectat a una línia lateral del sarcòmer anomenada línia Z.

Les **miosines** (diàmetre 15 nm) estan intercalades entre les actines, i estan connectades a una línia fosca (línia M) al centre del sarcòmer. Tenen una estructura amb forma de tronc amb protuberàncies.

La molècula de **titina** (llargada ≈1 µm) té una part elàstica i una part inelàstica. La inelàstica està dins la miosina i la part elàstica es connecta des del final de la miosina fins a la línia Z.

La zona on l'actina i la miosina se solapen es diu zona A, la zona on només hi ha miosina es diu zona H i la zona on només hi ha actina es diu zona I.

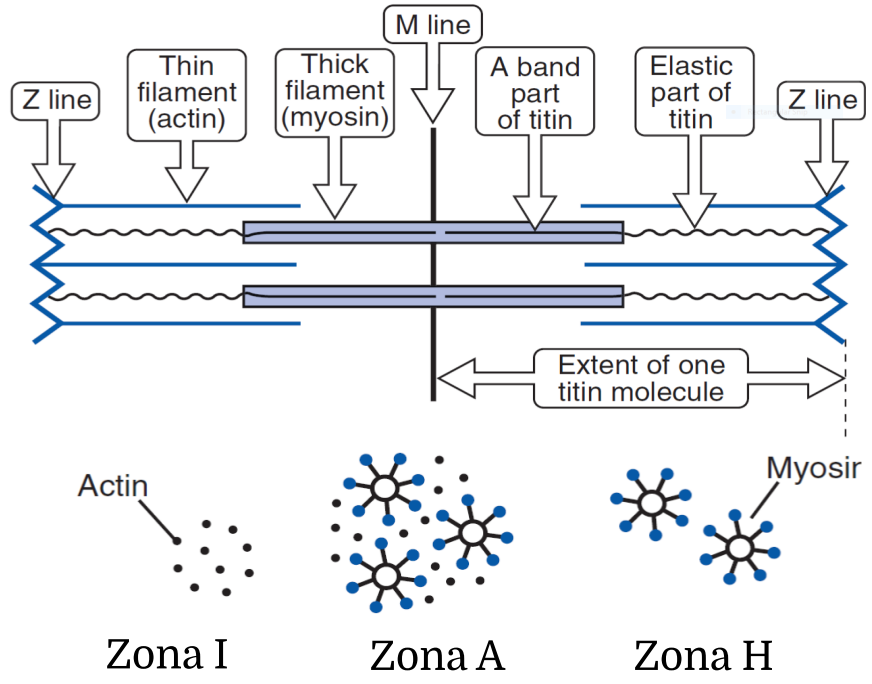


Figura 1.5 - Estructura d'un sarcòmer. Nom de la zona d'un sarcòmer dependent de quines proteïnes hi ha. - *[Lorenz, T., & Campello, M. 2012]*

1.2.2.2. Generació de força

No se sap amb 100% de certesa com un múscul genera força, però la teoria més acceptada és la teoria del *filament lliscant*. Proposada per Andrew Huxley el 1964; aquesta teoria explica que l'energia de contracció d'un múscul ve del contacte entre les proteïnes actina i miosina (Zona A) **[Figura 1.6]**. Com més solapament hi ha, més força podrà generar **[Huxley. 1974]**.

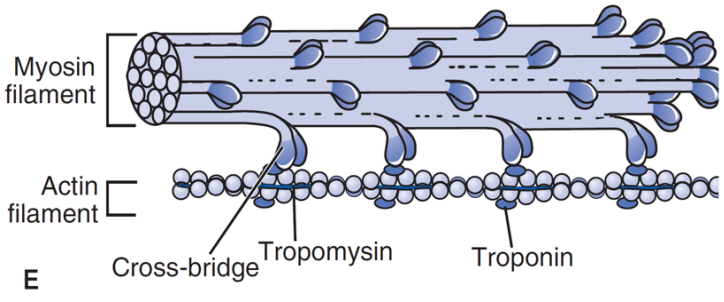


Figura 1.6 - Zona A del sarcòmer. - *[Lorenz, T., & Campello, M. 2012]*

Dins de la doble hèlix de l'actina es troben dues proteïnes addicionals, la troponina i la tropomiosina, que es creu que controlen el contacte entre l'actina i la miosina. La tropomiosina té forma de cadena i se situa entre ranures creades per les proteïnes d'actina, i la troponina s'enganxa aleatòriament a la tropomiosina.

Un element important a la teoria de Huxley és l'ió calci (Ca^{2+}). Quan la part contràctil del múscul fa contacte amb l'ió calci el múscul es contrau, i quan l'ió és retirat el múscul es relaxa. Les fibres s'agrupen en el que es coneix com a parts motores. Aquestes es contrauen totalment si tenen calci i no es contrauen si no en tenen, és a dir, no tenen un nivell intermedi de contracció. **[Lorenz & Campello. 2012]**

Des del punt de vista energètic, hi ha 2 tipus de fibra muscular; de contracció lenta (tipus I) i de contracció ràpida (tipus II). Els dos tipus de fibres es troben en cada múscul, però diferents músculs tindran diferents quantitats de cada una:

- La fibra de tipus I consumeix energia més lentament i pot treballar durant períodes de temps molt llargs. Per exemple, el múscul *gastrocnemi*[1] té una concentració molt alta de fibres de contracció lenta perquè s'utilitza sempre per mantenir l'equilibri.
- La fibra de tipus II reacciona quasi instantàniament, però es cansa molt de pressa i només serveix per esforços puntuals. Un exemple són tots els músculs de la parpella que tenen exclusivament fibres de tipus II, ja que necessiten una contracció molt ràpida i si intentes parpellejar consecutivament moltes vegades els músculs es cansaran molt ràpidament **[Rüegg. 1987]**.

1.3. Els ossos

Els ossos són els òrgans més durs del cos. Formen una estructura rígida però dinàmica en la qual els ossos es connecten entre si mitjançant lligaments. No només serveixen per crear un sistema esquelètic, sinó que també serveixen com a punts

[1] El gastrocnemi és un múscul de dos caps posicionat al darrere de la cama inferior i serveix per aixecar el taló. Col·loquialment, és referit com als 'bessons' fent referència als seus dos caps.

d'ancoratge per músculs que a través de tendons podran transmetre la seva força al múscul per crear moviments al cos.

Són fets d'un dels teixits més actius del cos, ja que té una capacitat elevada d'autoreparació i pot modificar la seva estructura per adaptar-se dinàmicament a les necessitats del cos. Això optimitza el consum de recursos, ja que no gasta matèria enfortint ossos que ja estan sobre desenvolupats.

1.3.1. Composició dels ossos

Els ossos són fets majoritàriament d'una part inorgànica i la resta orgànica. La part inorgànica està formada de cristalls petits de fosfat de calci ($Ca_{10}(PO_4)_6(OH)_2$), que constitueix un 60% del pes de l'os o un 40% del seu volum.

El material orgànic és una matriu orgànica de fibres i substàncies, està format majoritàriament per col·lagen tipus I. Aquestes fibres són dures i tenen un límit elàstic molt baix. Constitueixen un 30% del pes o un 35% del volum de l'os, la resta és aigua que constitueix un 10% del pes o un 25% del volum **[Lorenz & Campello. 2012]**.

Com ha estat esmentat prèviament, els ossos tenen una capacitat d'automodificació segons les seves condicions. Per aquesta raó la seva composició pot variar segons l'edat, dieta i si l'organisme dels ossos està afectat amb una malaltia **[Henkel. 2013]**.

1.3.2. Estructura dels ossos

Els ossos estan formats per una estructura complexa anomenada osteona, també coneguda com a *sistema d'Havers* **[Figura 1.7]**. És una estructura cilíndrica que es troba dins els ossos, i que en el centre conté venes i nervis. Envoltant l'osteona hi ha dos tipus de cèl·lules: osteoblasts i osteòcits. Dins l'osteona hi ha unions de fibres de col·lagen que determinarà com n'és de fort, l'os. Unions fortes augmentaran la resistència mecànica de l'os **[Lorenz & Campello. 2012]**.

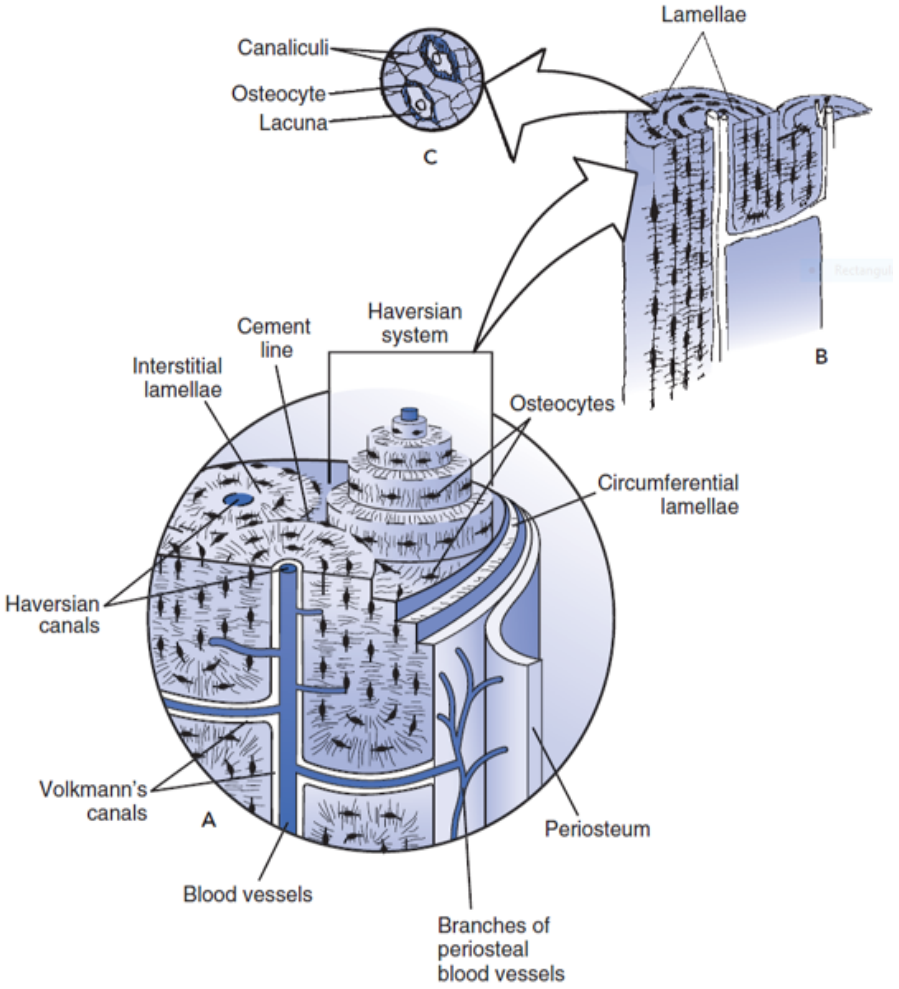


Figura 1.7 - Estructura del os (A), dins hi ha osteones (B) també conegut com sistema d'Havers. - *[Lorenz, T., & Campello, M. 2012]*

En el cos els ossos es presenten en dues formes, la primària i la secundària. La primària és l'os immadur que es forma en l'embrió. És més fràgil, ja que les osteones no estan alineades. Després d'un mes de vida el nadó començarà la remodelació òssia que transformarà els ossos de forma primària en forma secundària. Aquest segon tipus d'os és més dur pel fet que les osteones estan alineades.

Mecànicament, l'os és bifàsic, ja que està fet de dos materials. Un material bifàsic és un material que conté dos elements diferents que junts són més durs que cada un individualment; un exemple és el formigó que utilitza ciment i sorra per crear un material molt més dur. L'os utilitza minerals com a primer material i col·lagen com a segon material, una combinació que és més dura que els minerals i el col·lagen

individualment. El col·lagen és l'element més important per la duresa de l'os **[Burr. 2002]**, ja que el col·lagen gastat i falta de col·lagen poden reduir la duresa de l'os fins a un 60%, indicant que el col·lagen és el qui prevé esquerdes a l'os.

1.3.3. Articulacions

Les articulacions dels ossos són les unions en què s'ajunten dos o més ossos. Perquè el nostre cos sigui dinàmic, els ossos s'han de poder moure uns respecte als altres, sinó el cos seria rígid i els músculs no podrien fer gens de feina. Per tant, quan dos ossos es connecten s'han d'unir amb un element que els permeti fer el moviment que necessiten, això és el que fan les articulacions.

Segons el grau de moviment que requereixi la unió hi ha 3 tipus d'articulacions: les fibroses, les cartilaginoses i les sinovials.

1.3.3.1. Articulacions fibroses

Les articulacions fibroses són unes unions fixes que impedeixen el moviment entre els ossos. Hi ha 3 tipus d'articulacions fibroses: les sutures, les gomfosis i les sindesmosis.

- Les articulacions de **sutura** eliminen tot el moviment entre els dos ossos. Es troba en quasi tot el crani, ja que el crani ha de funcionar com un os unitari.

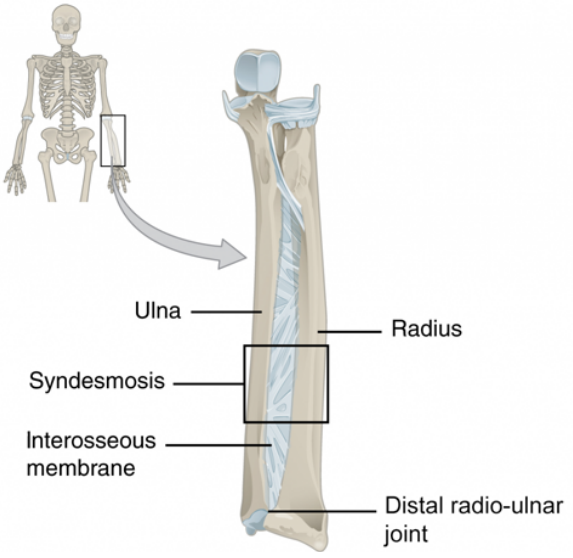


Figura 1.8 - Articulació sindesmosi entre els ossos de l'avantbraç: radi i cúbit. - *en.wikipedia.org/wiki/Syndesmosis*

- Les **gomfosis** també són immòbils però en comptes d'enganxar els ossos junts, un os entra dins de l'altre i s'uneixen. Per exemple les dents s'uneixen a la mandíbula amb una unió gomfosi.

- Finalment, hi ha les articulacions **sindesmosi**, que uneixen dos ossos paral·lels però els hi deixa un marge mínim de moviment. Un exemple és el radi i cúbit de l'avantbraç **[Figura 1.8]** que estan units però tenen una mica de moviment per poder rotar l'avantbraç **[UH. 2018 (1)]**.

1.3.3.2. Articulacions cartilaginoses

Com es veu en el nom aquestes articulacions uneixen els ossos amb cartílag i això fa que tinguin un moviment bastant reduït. Hi ha dos tipus d'aquestes articulacions: de sincondrosi i de símfisi.

- Les articulacions de **sincondrosi** s'uneixen amb cartílag hialí, es troben en les unions entre les costelles i l'estèrnum.

- L'altre tipus d'unió és la de **símfisi** que està feta de fibrocartílag, un cartílag que conté fibres de col·lagen tipus I. Aquestes unions es troben en la columna vertebral, entre les vèrtebres. **[UH. 2018 (2)]**.

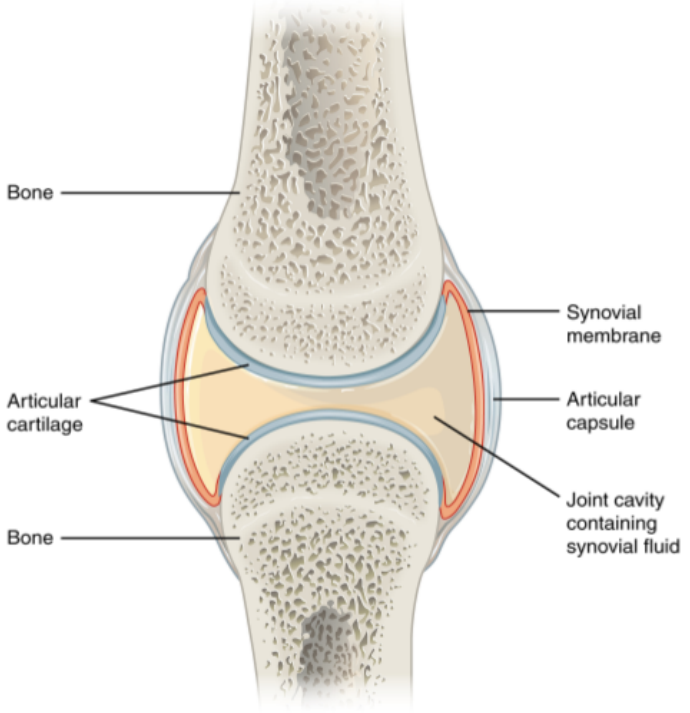


Figura 1.9 - Articulació sinovial gínglim - *en.wikipedia.org/wiki/Synovial_joint*

1.3.3.3. Articulacions sinovials

Les articulacions sinovials són les articulacions més comunes en el cos, i són les que permeten la major llibertat de moviment. Els ossos no es connecten directament entre ells, de manera que quan es mouen hi ha menys fregament i la rotació és més suau.

Aquesta articulació es diferencia de les altres perquè la unió està rodejada amb una càpsula articular que està plena de líquid sinovial que serveix de lubricant. A més, les puntes dels ossos estan cobertes d'una capa fina de cartílag hialí que allisa les puntes per reduir el fregament.

Hi ha 6 tipus diferents d'articulacions sinovials però les més típiques que es repeteixen sovint en el cos només són 3: els gínglims, trocoïdal i esfèriques **[Figura 1.10]**.

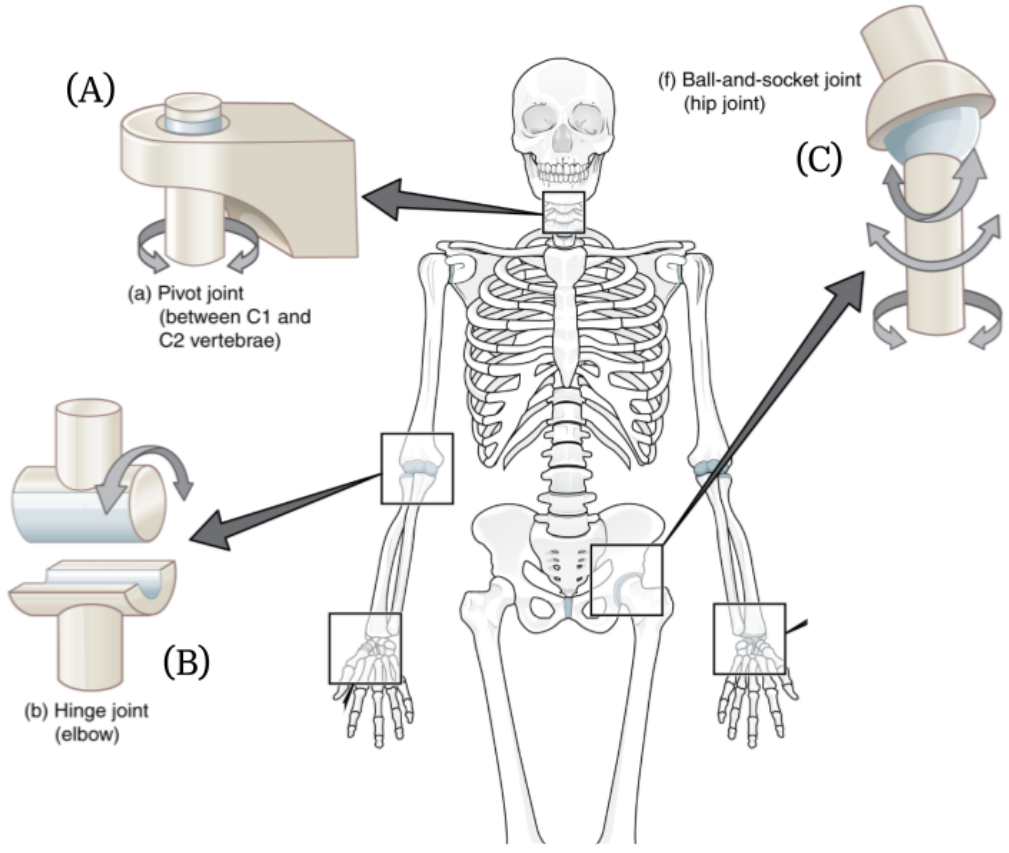


Figura 1.10 - (A) Articulació trocoïdal, (B) Articulació gínglim, (C) Articulació esfèrica - *https://en.wikipedia.org/wiki/Synovial_joint*

- L'articulació **gínglim [Figura 1.9]** és la més freqüent. Permet doblegar i estirar la unió en una direcció, de manera que es classifica com una articulació uniaxial. Un exemple d'aquesta unió és el colze que rota el radi i el cúbit de l'húmer.

- Un altre tipus és la unió **trocoïdal**. En aquesta unió, una part arrodonida de l'os és inserida dins l'altre os, que té forma d'anell. Deixa rotar l'os arrodonit dins l'anell de l'altre, per exemple el coll té una articulació així, ja que permet rotar-lo.

- Finalment, l'últim tipus d'unió és l'**esfèrica**. Aquesta articulació és la que permet el nivell màxim de moviment en 3 eixos de rotació. Una punta d'un os té forma esfèrica i està rodejat per l'altre os que té forma de bol. Articulacions com aquesta es poden trobar entre el maluc i el fèmur, i en l'espatlla entre l'húmer i l'escàpula. **[UH. 2018 (3)]**

1.4. Interacció entre els músculs i els ossos

1.4.1 Unions entre músculs i ossos

En la majoria dels músculs, la unió entre el múscul i l'os es fa mitjançant tendons que sorgeixen dels extrems dels músculs i s'insereixen en els ossos **[Figura 1.11]**. Això significa que la força màxima dels músculs també està limitada per la força que pot aguantar el tendó.

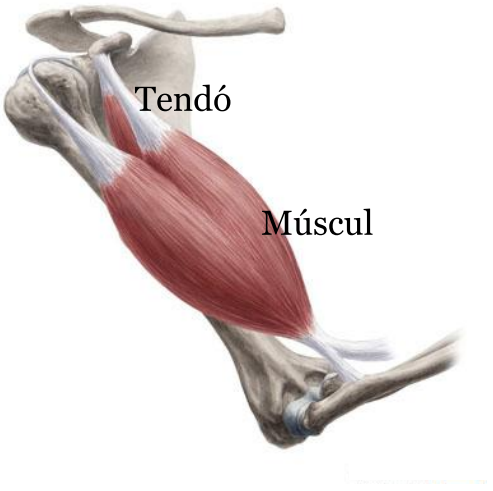


Figura 1.11 - El Biceps Brachii amb el tendó i el múscul. - www.kenhub.com /en/library/anatomy/biceps-brachii-muscle

Els tendons són un conjunt de fibres que estan fetes predominantment de col·lagen tipus I. Aquestes fibres són parcialment elàstiques. Quan un múscul es contrau, el tendó s'allarga una mica i emmagatzema

energia elàstica potencial i això crea un retrocés automàtic quan el múscul es relaxa **[Lorenz & Campello. 2012]**.

Els tendons s'integren amb els músculs mitjançant una estructura anomenada la unió miotendinosa **[Schweitzer. 2010]**. De forma semblant, els tendons i els ossos s'uneixen mitjançant les unions osteotendinoses **[Benjamin. 2006]**.

1.4.2 Tipus de moviments

Hi ha 5 moviments principals que poden generar els músculs i els ossos, 4 dels quals estan emparellats amb el moviment oposat.

- Adducció **[Figura 1.12]**: Moviment en el sentit cap el centre vertical del cos. Per exemple, quan prems les cames juntes quan estàs sobre un cavall per no caure.

- Abducció **[Figura 1.12]**: Moviment d'una part del cos amb el sentit oposat al centre vertical del cos. Per exemple, quan carregues bosses de compra però les apartes de tu perquè les bosses no xoquin amb tu.

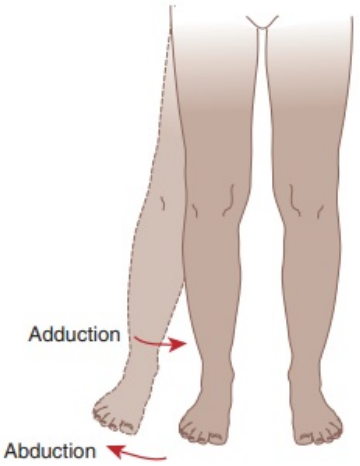


Figura 1.12 - Adducció i abducció de la cama - *www.pharmacy180.com/media/imgph04/xIHbiZj.jpg*

- Flexió **[Figura 1.13]**: Doblegar una unió per reduir l'angle entre les parts de la unió. Per exemple, quan doblegues la cama per xutar una pilota de futbol.

- Extensió **[Figura 1.13]**: Estirar una unió per augmentar l'angle entre les parts de la unió. Per exemple, quan un boxejador estira el braç per pegar l'oponent.

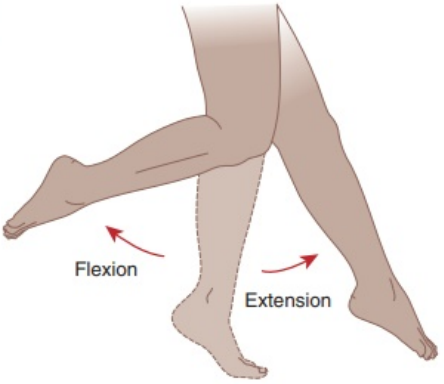


Figura 1.13 - Flexió i extensió de la cama - *www.pharmacy180.com/media/imgph04/xIHbiZj.jpg*

- Rotació **[Figura 1.14]**: Girar una part del cos en un eix. Per exemple, quan gires el cap per veure què hi ha al teu costat.

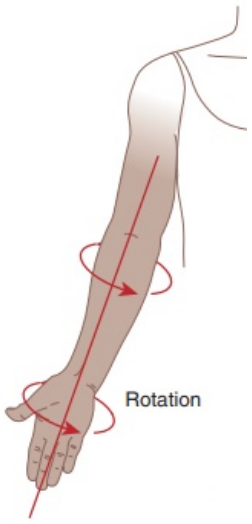


Figura 1.14 - Rotació del braç - *www.pharmacy180.com /media/imgph04/t8VpzGR.jpg*

2. Models musculoesquelètics

2.1. Utilitat de models musculoesquelètics

La física clàssica és útil per mesurar i calcular forces externes i energies d'un cos, com: la inèrcia angular, el moment, la temperatura, etc. Però en la biomecànica també cal calcular les forces generades pel múscul mateix. Per això s'utilitzen els models musculars, que a partir de la llargada del múscul, la velocitat de contracció, l'activació i l'angle de les fibres permet obtenir la força generada per qualsevol múscul.

Com que els músculs no treballen sols també s'utilitza models ossis per calcular les tensions dels ossos. Però com que els ossos no generen tensió el seu comportament es pot calcular mitjançant fórmules de la mecànica de parells i deformacions. Els dos models s'uneixen per simular el comportament del sistema musculoesquelètic.

Aquests models musculoesquelètics són molt útils en experiments sobre el cos humà que són massa perillosos per fer servir subjectes de prova humans. Per exemple, un estudi **[Mortensen & Merryweather. 2018]** va determinar que fer servir humans per examinar l'afecte d'una caiguda lliure al crani humà no era factible, i doncs va decidir fer servir l'*OpenSim*[2], un simulador musculoesquelètic.

2.2. Model muscular estàtic

Un múscul es pot modelar com un objecte estàtic, on a partir de les seves dimensions es calcula la seva força màxima que pot generar. Però, per fer això, s'ha de decidir en quines dimensions es calcularà el múscul **[Figura 2.1]**.

[2] L'*OpenSim* és un programa de codi obert per la modelització de sistemes biomecànics en 3D. Originàriament creat el 2007, és àmpliament utilitzat en el món acadèmic.

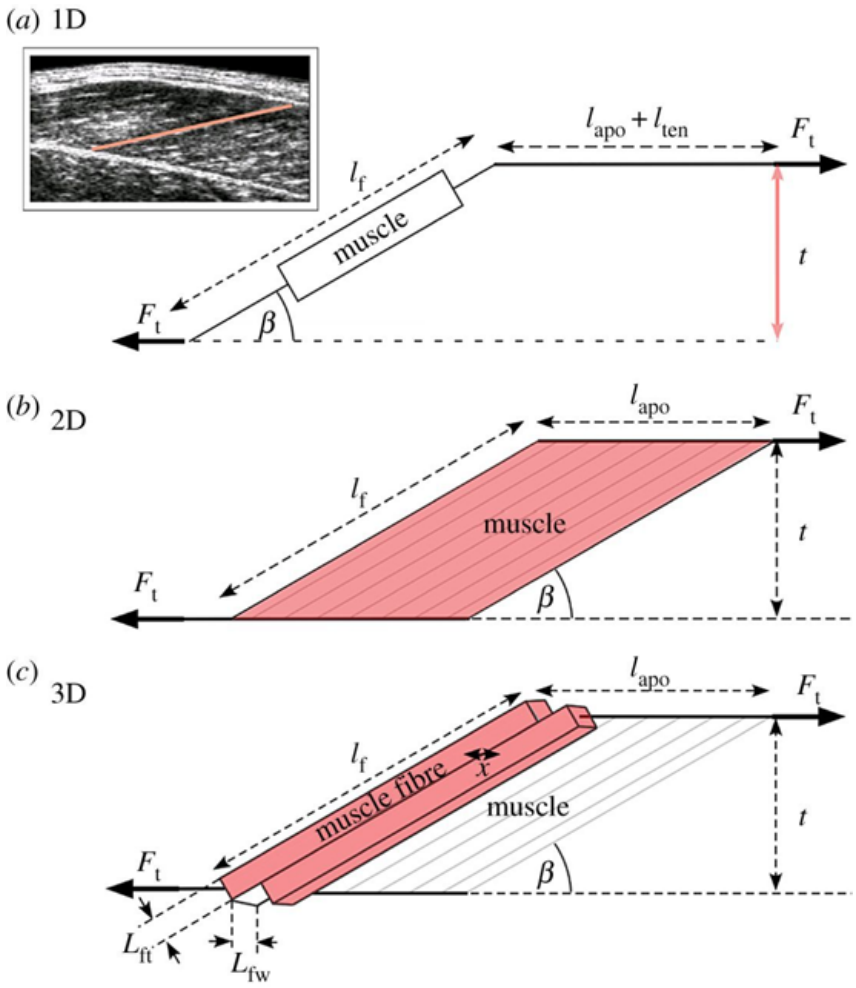


Figura 2.1 - Representació esquemàtica de models musculars en 1D (A), 2D (B) i 3D (C) - *[Dick, TJM., & Wakeling, JM. 2018]*

Com qualsevol objecte real, un múscul està en 3D. Per aquesta raó sembla que s'hauria de modelar un múscul en 3D, però no sempre és el cas, ja que models amb menys dimensions poden aconseguir una precisió similar que els models amb més dimensions.

En decidir quantes dimensions tindrà un model s'ha de tenir en compte que com més dimensions tingui un model més complicat serà. Aquesta complicació del model surt car, ja que la precisió extra que s'obté és mínima comparat amb models més simples. Models en 1D prediuen una llargada i angle dels fascicles similars que models en 2D i en 3D. Però, els models en 1D estan limitats a l'hora de calcular la mecànica del múscul quan la forma del múscul canvia durant la contracció. **[Dick & Wakeling. 2018]**

2.2.1. Fórmules

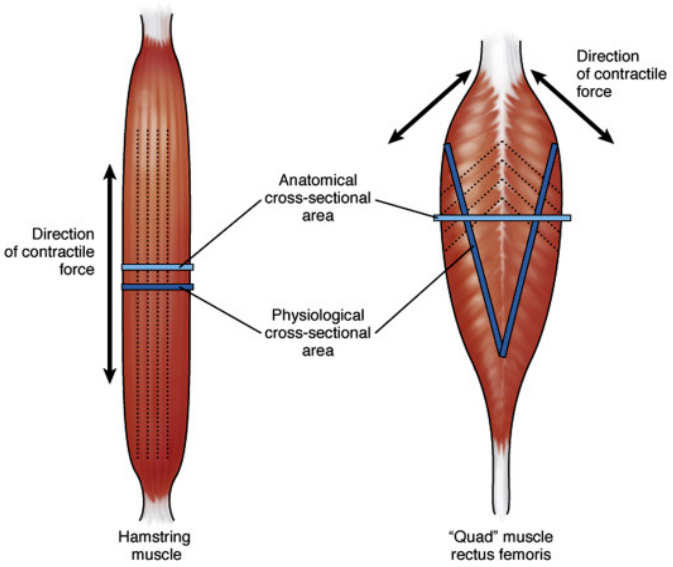


Figura 2.2 - Diferència entre la secció anatòmica (blau clar) i la secció fisiològica (blau fosc) - *www.westsubpainrelief.com/deltoid-muscles*

La força màxima d'un múscul depèn del seu gruix o secció, i es calcula com el seu esforç unitari màxim (σ_{max}), amb valors de 30 a 40 N/cm^2 **[SLH. 2007]**, multiplicat per la seva secció fisiològica (s_f) **[Figura 2.2]**. Això és perquè, per mesurar la força que genera una un múscul, s'ha de calcular la força en el mateix angle de les fibres musculars, no del cos del múscul. Per tant, per calcular la força, s'agafa la secció fisiològica i no la secció anatòmica. **[Delp. 2022 (2)]**

$$F_{max} = s_f \cdot \sigma_{max}$$

La secció fisiològica és molt simple de calcular. S'agafa el volum del múscul (V^m) i es divideix per la llarga de les fibres del múscul en repòs ($l^f_{repòs}$).

$$s_f = V^m \cdot l^f_{repòs}$$

I la llargada de les fibres es pot aconseguir de dues maneres. Multiplicant el nombre de sarcòmers (n) en la fibra per la llargada d'un sarcòmer en repòs ($l^s_{repòs}$), uns 2 µm per la majoria de músculs **[Lorenz & Campello. 2012]**.

$$l^f_{repòs} = n \cdot l^s_{repòs}$$

O, per no haver de comptar la quantitat de sarcòmers, es pot mesurar empíricament la llargada d'una fibra (l^f) i la llargada dels seus sarcòmers (l^s) i aplicar-los aquesta fórmula per obtenir la llargada d'una fibra muscular en repòs. **[Delp. 2022 (2)]**

$$l^f_{repòs} = l^f \cdot (l^s_{repós} \div l^s)$$

2.2.2. Angle de pennació

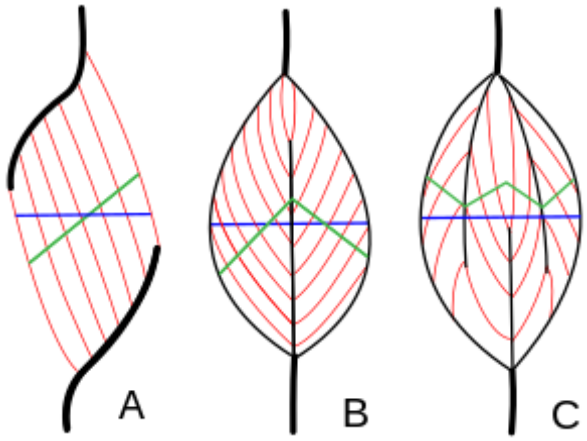


Figura 2.3 - Representació de les seccions en músculs amb diferents pennacions - *www.wikipedia.org/wiki/Pennate_muscle*

En alguns músculs, les fibres no estan posicionades paral·lelament a la direcció de la força, sinó que estan en un angle. Aquests músculs es diuen músculs pennats, i l'angle en què estan es diu l'*angle de pennació*.

Els músculs pennats es classifiquen en 3 grups **[Figura 2.3]**:

- **Unipennats (A)**; totes les fibres estan al mateix angle de pennació. Per exemple, el *Vastus Lateralis* dels quàdriceps.

- **Bipennats (B)**; el múscul es divideix entre dos i cada grup té un angle de pennació simètric de l'altre. Per exemple, el *Rectus Femoris* dels quàdriceps.

- **Multipennats (C)**; les fibres estan agrupades en molts grups amb cada grup amb el seu angle de pennació. Per exemple, el *Deltoideus*.

El benefici de què les fibres tinguin un angle de pennació és que així poden generar més força. Això és així perquè la força màxima que pot generar un múscul depèn de la tensió i la secció fisiològica, i com més gran sigui l'angle de pennació més gran serà la secció [Delp. 2022 (2)].

Però aquesta força addicional ve a un cost, que les fibres musculars han de ser més curtes. Això és

Figura 2.4 - Els músculs Rectus Femuris i Vastus Midialis en la cama d'en Jay Cutler - *www.the barbell.com/jay-cutler-quad-stomp*

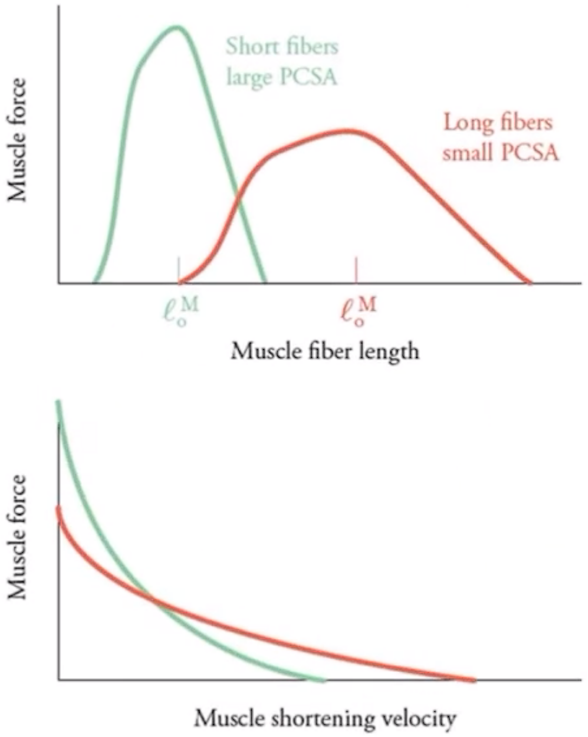


Figura 2.5 - Diferència en força, velocitat i contracció en fibres musculars llargues i curtes - *[Delp, S. 2022 (2)]*

així perquè com més rotades estan, menys distància hi ha fins a la perifèria del múscul. Que les fibres siguin més curtes causa que la diferència de la llargada del múscul entre la contracció i la relaxació disminueixi, i per tant també disminueixi el moviment màxim de l'articulació. A més, també causa que la velocitat de contracció baixi perquè hi ha menys sarcòmers per cada fibra. **[Figura 2.5] [Delp. 2022 (2)]**

Com s'ha establert en l'apartat 2.2.1, quan un múscul es contrau també es fa més ample per mantenir el seu volum constant **[Figura 2.6]**, això també s'aplica als músculs pennats. Però quan s'amplien, l'angle de pennació s'incrementa, el que significa que les propietats mecàniques del múscul varia durant la contracció **[Dick & Wakeling. 2018]**.

A primera vista aquest canvi de mecànica no sembla massa important, però el fet que estigui relacionat amb la contracció del múscul fa que el múscul augmenti la força per càrregues grans i la velocitat per càrregues petites. Bàsicament, el múscul té un sistema 'de marxes automàtic' que depèn de la càrrega que ha de moure. **[Azizi. 2007]**

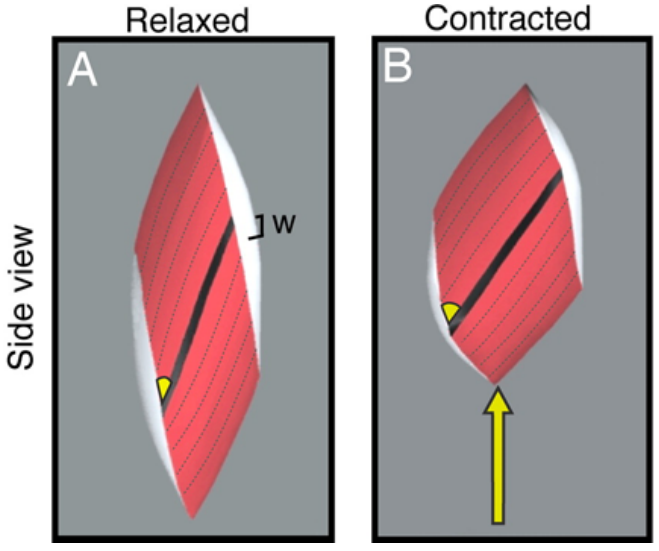


Figura 2.6 - Dimensions d'un múscul quan està estirat i contret. El volum és constant - *[Azizi, E., et al. 2007]*

Aquest comportament del múscul és un factor que complica els models i és el desavantatge principal que té els models en 1D comparats amb els models en 2D i 3D. En contraccions on l'activació del múscul és màxima i no hi ha aquest sistema de marxes automàtic, els models en 1D calculen l'angle de pennació amb una correlació amb les dades reals de r^2 = 0,99. Però si l'activació no és màxima el model en 1D no podrà calcular aquests canvis de la mecànica del múscul. **[Dick & Wakeling. 2018]**

2.2.3. Implementació de l'angle de pennació

L'angle de pennació **[Figura 2.7]** és un factor important en calcular la força màxima d'un múscul. Aquest angle afecta, indirectament, la llargada de les fibres del múscul. Però si no es pot obtenir la llargada de les fibres musculars empíricament, es pot ignorar la llargada de les fibres i obtenir la secció fisiològica a partir de la secció anatòmica mitjançant trigonometria.

$$s_f = s_a \div \cos(\beta)$$

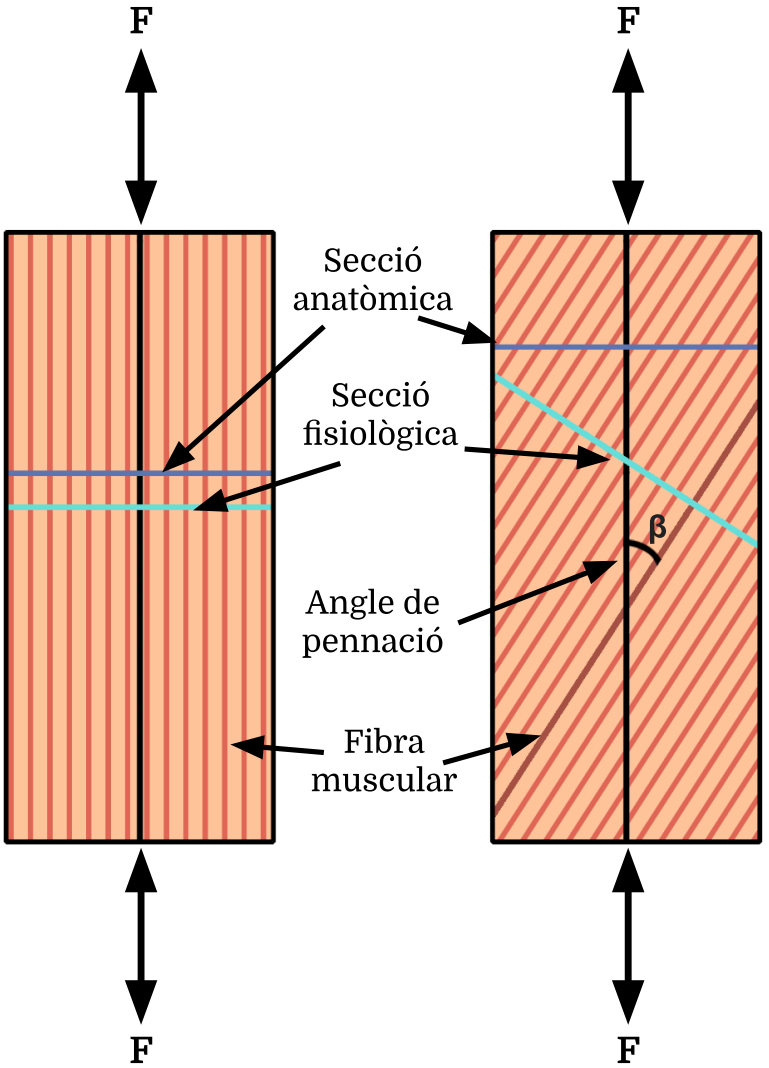


Figura 2.7 - Representació d'un múscul sense angle de pennació (esquerra), i un múscul amb angle de pennació (dreta)

2.3. Model muscular dinàmic

2.3.1. Propietats dinàmiques dels músculs

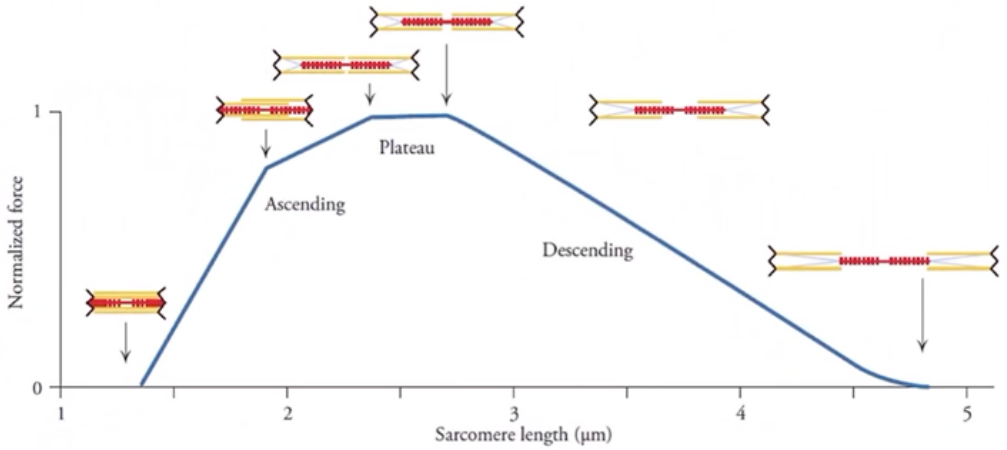


Figura 2.8 - Força màxima un sarcòmer dependentment de la seva llargada - [Delp, S. 2022 (1)]

Una de les propietats dinàmiques que tenen els músculs és que són quasi isovolumètrics **[Dick & Wakeling. 2018]**. Això vol dir que quan el múscul fa força i es contrau, també fa una força perpendicular que augmenta el gruix del múscul. Aquesta segona força fa que el volum del múscul es mantingui quasi constant durant la contracció.

També els músculs tenen una relació entre la seva llargada i la força que poden generar **[Figura 2.8]**. Seguint la teoria de Huxley, com més solapament hi ha entre la miosina i l'actina en el sarcòmer, més força es podrà generar. D'això podem extreure que hi ha un punt en la contracció on hi ha el màxim solapament i, per tant, on es genera la màxima força. Però si el sarcòmer es contrau massa l'actina se superposarà amb ella mateixa, el

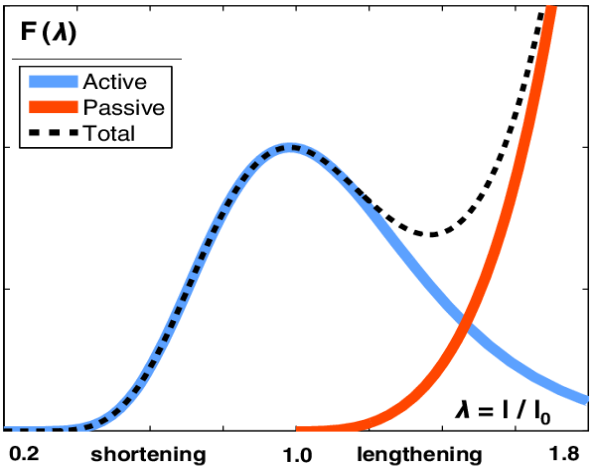


Figura 2.9 - Força activa i força passiva màxima dependentment de la llargada del múscul - *[Wisdom, K., et al. 2014]*

solapament amb la miosina es reduirà, i la força generada pel múscul serà menor. **[Delp. 2022 (1)]**

Finalment, un múscul genera dos tipus de força: activa i passiva. La força activa (línia blava a la figura 2.9) és generada com diu la teoria de Huxley per l'actina i la miosina. Però la força passiva (línia vermella a la figura 2.9) és una tensió generada per la proteïna titina. La titina funciona com una molla no lineal que com més allargada està més tensió generarà **[Kellermayer. 1997]**.

2.3.2 Model Hill

N'Archibald Hill va ser un pioner en l'àmbit de la fisiologia i es va centrar en la producció de calor pels músculs. L'any 1938, en l'article anomenat "*The heat of shortening and the dynamic constants of muscle*"[3] es va proposar el model de Hill. Aquest article mostra les equacions fonamentals per la modelació dels músculs i crea un disseny de com es pot modelar un múscul. Segons el model Hill, un múscul es modela en 3 elements **[Figura 2.10] [Lemaire. 2016]**.

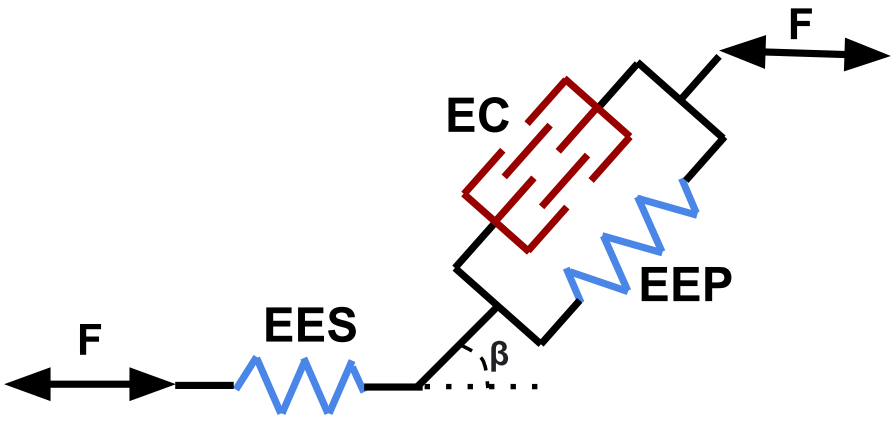


Figura 2.10 - Diagrama mecànic d'un múscul segons el model de Hill

- Té un **element contràctil (EC)** que és qui genera la força activa. Aquest element, com es veu en el diagrama, es representa amb forma de sarcòmer, ja que és el sarcòmer qui genera la força.

[3] "La calor de contraccions i les constants dinàmiques del múscul"

- Hi ha un **element elàstic en sèrie (EES)** que funciona com una molla no-lineal que està posicionada en seria a l'EC. Aquest element representa el tendó.

- Finalment, hi ha un **element elàstic en paral·lel (EEP)** que també funciona com una molla no-lineal però que està posicionada paral·lelament a l'EC. Representa la fàscia que rodeja el múscul estriat.

2.3.2.1. Element contràctil: força activa

La força activa és creada per l'element contràctil i hi ha 3 factors que afecten la força generada: l'activació, la llargada i la velocitat de contracció **[Dick & Wakeling. 2018]**.

Així doncs, per obtenir la força activa, s'aconsegueix el coeficient de l'efecte que té cada factor a la força màxima, s'obté el coeficient total i es multiplica per la força màxima per tenir la força activa generada.

$$F^a = F_{max}\,(a \cdot f^l{}_a(\bar{l}) \cdot f^v(\bar{v}))$$

On:

- $\mathbf{F^a}$ = força activa
- $\mathbf{F_{max}}$ = força total màxima
- $\mathbf{a}$ = coeficient d'activació del múscul
- $\bar{l}$ = $l \div l_{\text{repòs}}$
- $\bar{v}$ = $v \div v_{\text{max}}$
- $f^v(\bar{v})$ = coeficient de força activa a partir de la velocitat normalitzada
- $f^l{}_a(\bar{l})$ = coeficient de força activa a partir de la llargada normalitzada

L'activació és controlada pel cervell, segons del pes que hagi d'aixecar activarà el múscul més o menys. Així doncs, l'activació és un valor que s'ha d'estimar o obtenir a partir de mesures experimentals.

La relació entre la llargada i la força és específica per cada múscul. Per calcular el coeficient de l'efecte que la llargada té a la força ($\bar{\mathbf{F}}_A$), la llargada normalitzada ($\bar{l}_m$) s'ha d'aplicar a una funció **[Delp. 2022 (3)]**. Aquesta funció s'obté d'interpolar-la de valors empírics **[Figura 2.11]**. La variable independent d'aquesta funció és la llargada del

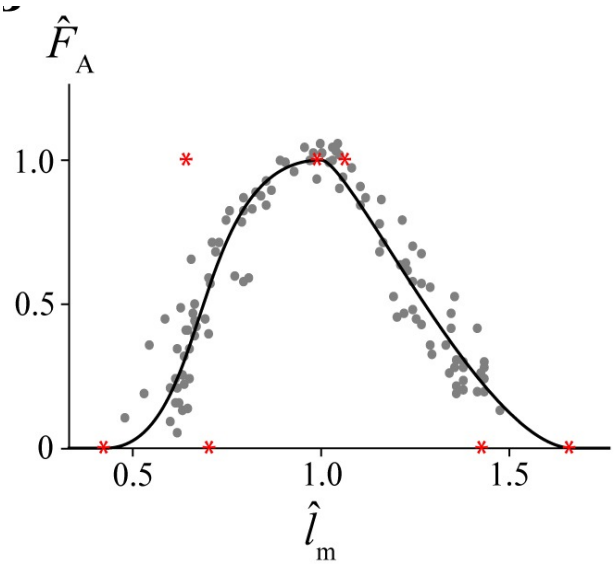


Figura 2.11 - Funció llargada - força activa del model de Hill a partir de valors empírics - *[Ross, et al. 2018]*

s'utilitza una funció obtinguda a partir de valors empírics **[Figura 2.12]**. La variable independent d'aquesta funció és la velocitat del múscul normalitzada. En que és un valor negatiu quan el múscul es contrau (ja que s'està fent més curt, doncs té velocitat negativa) i un nombre positiu quan el múscul s'està relaxant (perquè s'està fent més llarg, i doncs la velocitat és positiva). La funció retornarà un valor entre 0 i 1,5 que és el coeficient de la força màxima que pot generar aquella velocitat. **[Delp. 2022 (3)]**

múscul normalitzada del múscul. O sigui, la llargada del múscul dividida entre la llargada del múscul en repòs. La funció retornarà un valor entre 0 i 1 que és el coeficient de la força màxima que el múscul pot fer a aquella llargada.

L'altre factor que afecta la força activa ($\bar{\mathbf{F}}_V$) del múscul és la velocitat contracció i relaxació ($\bar{v}_m$). Una contracció muscular generarà menys força que una relaxació. I com més ràpida sigui, més diferència hi haurà. Per calcular aquest coeficient també

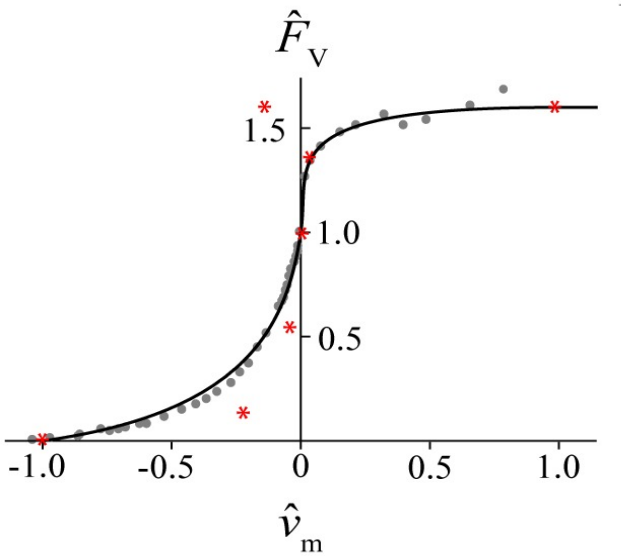


Figura 2.12 - Funció velocitat - força activa del model de Hill a partir de valors empírics - *[Ross, et al. 2018]*

2.3.2.2. Elements elàstics: forces passives

La força passiva és creada pels dos elements elàstics: el paral·lel i l'en sèrie. Aquesta força només depèn de la llargada del múscul. Per tant, per calcular la força passiva s'ha d'obtenir el coeficient de l'efecte que té en la força màxima **[Dick & Wakeling. 2018]**.

$$F^p = F_{max} \cdot f^l_p(\bar{l})$$

On:

- $\mathbf{F^p}$ = força passiva
- $f^l_p(\bar{l})$ = coeficient de força passiva a partir de la llargada normalitzada

La relació entre la llargada del múscul i la força passiva és semblant a una molla. La força passiva només comença a aparèixer quan la llargada del múscul és superior que quan està en repòs (llargada normalitzada ($\bar{l}_m$) > 1). Per tant, com més s'allargui el múscul més força passiva serà generada. El coeficient de l'efecte que la llargada té a la força passiva ($\bar{F}_P$) es calcula a partir de la funció obtinguda a partir de valors empírics **[Figura 2.13]**. La variable independent és la llargada normalitzada del múscul. Aquesta funció no té en compte les forces màximes que pot aguantar el tendó, teòricament, pot retornar un valor del 0 a l'infinit. Però, com que un tendó no aguanta més de 60 MPa d'esforç unitari i un múscul no l'aplica més de 30 MPa, la funció té un límit en l'eix Y dependentment de quin múscul s'està calculant **[Delp. 2022 (3)]**.

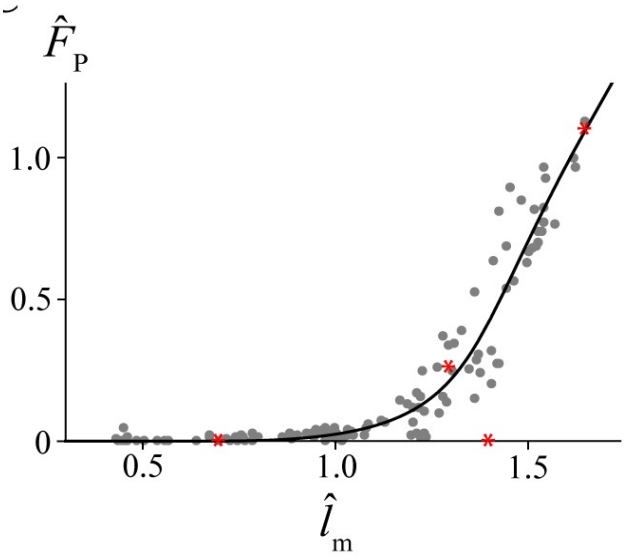


Figura 2.13 - Funció llargada - força passiva del model de Hill a partir de valors empírics - *[Ross, et al. 2018]*

2.3.2.3. Model conjunt

Ara que ja se sap com obtenir la força activa (F^a) i passiva (F^p), es pot obtenir la força total (F) que genera un múscul. Això és fàcil:

$$F = F^a + F^p$$

La força del múscul és la suma de la força activa i la força passiva. Així doncs, si substituïm F^a i F^p per la seva fórmula obtingudes a 2.3.2.1 i 2.3.2.2. tenim:

$$F = F_{max}\,(a \cdot f^l{}_a(\bar{l}) \cdot f^v(\bar{v})) + F_{max} \cdot f^l{}_p(\bar{l})$$

I, finalment, si extraiem el factor comú F_{max}, tenim la fórmula general de la força generada per un múscul **[Delp. 2022 (3)]**:

$$F = F_{max}\,(a \cdot f^l{}_a(\bar{l}) \cdot f^v(\bar{v}) + f^l{}_p(\bar{l}))$$

2.3.2.4. Equació d'estat

En Hill, en la seva recerca, també va enllaçar les contraccions i propietats musculars amb l'energia tèrmica, i va derivar aquesta equació d'estat que s'aplica a tots els músculs **[Hill. 1938]**:

$$(F + a)\,(v + b) = b\,(F_{max} + a)$$

En que:

- **F** = tensió o càrrega en el múscul
- **a** = coeficient de calor de contracció
- **v** = velocitat de contracció
- **b** = $\mathbf{a \cdot v_{max} \div F_{max}}$
- $\mathbf{v_{max}}$ = velocitat de contracció màxima quan **F = 0**

Aquesta equació és molt important en l'àmbit de la fisiologia, però té el problema de no tenir en compte el nivell d'activació muscular. Això vol dir que aquesta equació només es pot fer servir en casos ideals en que l'activació és màxima. I, per tant, té poca utilitat en els models biomecànics.

2.4. Models ossis

Mecànicament, un os no és més que un cos rígid; no genera forces i és vulnerable a forces externes. La seva feina és aguantar el pes del cos i ser un punt d'unió dels músculs perquè puguin transmetre la seva força a través del cos. Això significa que han de ser un cos dur, però com tots els cossos que reben forces; també tindran un cert punt de deformacions.

2.4.1. Esforços

Les forces externes són les que s'apliquen a un cos. Però, els efectes d'una força depenen de la superfície on s'aplica. És a dir, una força aplicada a una superfície petita tindrà un impacte més gran que la mateixa força aplicada a una superfície més gran. Per saber l'impacte d'aquestes forces, s'utilitzen els esforços unitaris (σ). Assumint que la força es distribueix uniformement, l'esforç unitari es pot calcular com la força (***F***) dividida per a la superfície (***A***). **[Figura 2.14] [Joseph. 2017. {p.168}]**

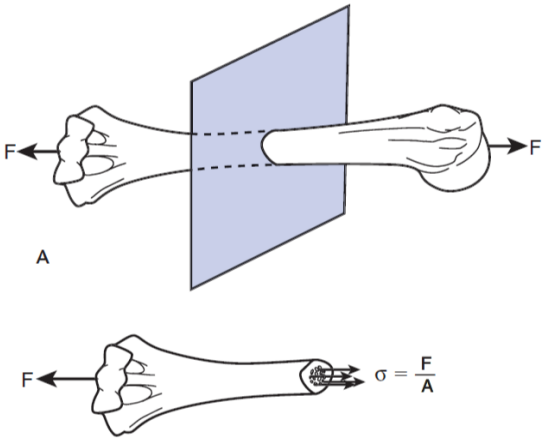


Figura 2.14 - Representació de l'esforç unitari en un os - *[Özkaya, N., & Leger, D. 2012]*

$$\sigma = F \div A$$

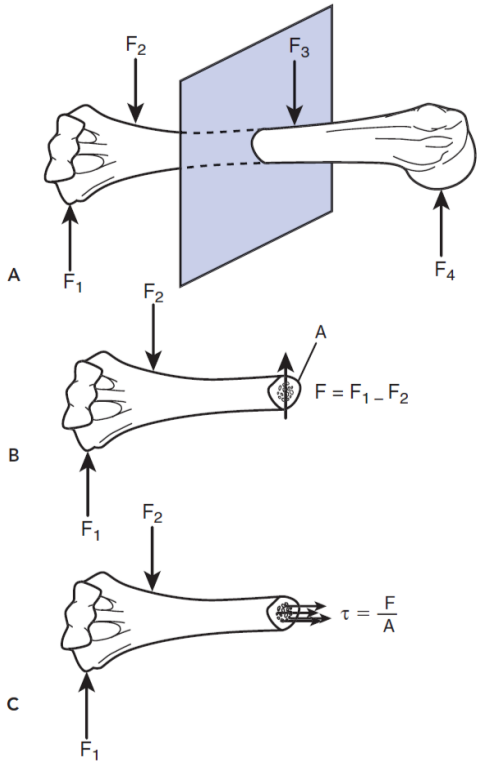


Figura 2.15 - Representació de l'esforç unitari tangencial en un os - *[Özkaya, N., & Leger, D. 2012]*

Aquesta equació és igual que l'equació de pressió, així doncs, la pressió i l'esforç unitari tenen les mateixes unitats N/m^2 o pascals. Aquest tipus d'esforç unitari (que actua perpendicular a la superfície) també és anomenat esforç unitari normal. Això és perquè hi ha un segon tipus d'esforç que s'anomena esforç unitari tangencial (τ).

L'esforç unitari tangencial és un esforç intern del cos que actua tangencialment (paral·lelament) a la superfície. Per calcular això es divideix la força externa per la superfície. **[Figura 2.15] [Özkaya. 2012]**

$$\tau = F \div A$$

2.4.2. Deformacions

Quan una força és aplicada a un cos aquest serà deformat, perquè cap cos és infinitament dur. Això també s'aplica als ossos, les seves propietats com l'estructura i la densitat poden afectar a com són de vulnerables a les forces. I factors externs com la humitat, la temperatura i la magnitud i direcció de la força. **[Özkaya. 2012]**

Les deformacions, com els esforços, també són mesurades unitàriament, ja que una deformació a un cos petit té més impacte que la mateixa deformació a un cos gran. Les deformacions unitàries són la proporció de deformació que rep un cos comparat amb les seves dimensions inicials.

Si un cos és sotmès a tensió, s'allargarà. Si és sotmès a compressió, s'encongirà. Aquestes dues deformacions són producte d'esforços normals, i són deformacions

perpendiculars a la superfície. Però, si l'esforç és tangencial, la deformació serà paral·lela a la superfície.

Les deformacions unitàries normals (ε) representen el coeficient d'allargament (positiu o negatiu) d'un cos. Es calcula a partir de la llargada inicial i l'increment de llargada. **[Figura 2.16] [Joseph. 2017. {p.169}]**

$$\varepsilon = \Delta l \div l_i$$

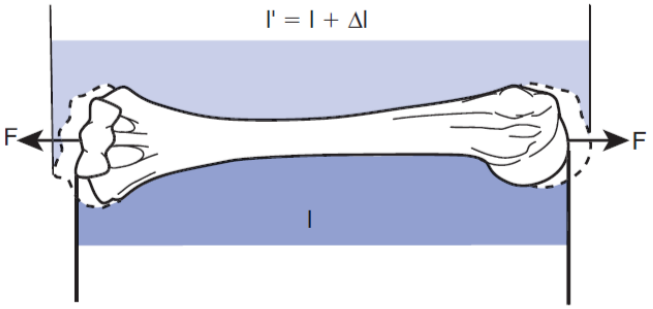


Figura 2.16 - Representació d'una deformació normal en un os - *[Özkaya, N., & Leger, D. 2012]*

Les deformacions unitàries tangencials (γ) són les deformacions causades per les forces tangencials. En comptes d'allargar el cos, 'desplaça' un cantó respecte a l'altre. La deformació és la distància que un cantó és desplaçat dividit entre l'altura del cos o la tangent de l'angle que forma. Fent servir la figura 2.17 com a exemple, la deformació unitària tangencial és **[Figura 2.17] [Özkaya. 2012]**:

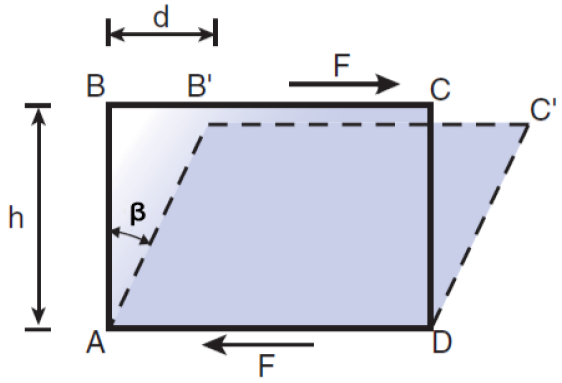


Figura 2.17 - Representació d'una deformació tangencial - *[Özkaya, N., & Leger, D. 2012]*

$$\gamma = d \div h$$

$$\gamma = tg(\beta)$$

Com que l'angle β sovint és molt petit, la tangent del qual és quasi idèntic a l'angle mateix. Així doncs, es pot dir que **[Özkaya. 2012]**:

$$\beta \approx tg(\beta)$$

Per tant:

$$\gamma = d \div h \approx \beta$$

2.4.3. Diagrama de tracció

Un diagrama de tracció **[Figura 2.18]** mostra la relació entre els esforços i les deformacions. És un diagrama on l'abscissa és la deformació unitària del cos i l'ordenada és l'esforç unitari del cos. La majoria dels cossos tenen una línia semblant en aquest diagrama.

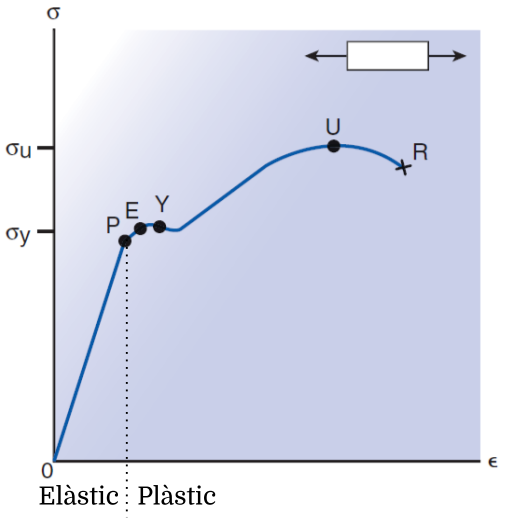


Figura 2.18 - Diagrama de tracció - *[Özkaya, N., & Leger, D. 2012]*

2.4.3.1. Deformació elàstica

En el tram del punt 0 al P, la línia és una recta. Aquest tipus de deformació es diu deformació elàstica. El gradient de la deformació elàstica és constant i és igual a la rigidesa del cos. El gradient es diu mòdul de Young (***E***) i es calcula dividint l'esforç unitari entre la deformació unitària.

$$E = \sigma \div \varepsilon$$

En alguns casos l'esforç unitari tangencial (τ) mostra una relació lineal amb la deformació unitària tangencial (γ). La constant elàstica d'aquests cossos s'anomena el mòdul tangencial (G).

$$G = \tau \div \gamma$$

Les deformacions elàstiques no són permanents, i quan la força finalitza, el cos tornarà a tenir les mateixes dimensions.

2.4.3.2. Deformació plàstica

Quan la deformació es torna permanent la deformació ja no és elàstica, sinó plàstica. Això passa quan l'esforç unitari és superior al límit elàstic (σ_y), a partir del punt E.

La deformació plàstica es pot dividir en tres fases:

- **Fluència** (E - Y). La deformació plàstica comença amb un tram on la deformació del cos incrementa molt sense haver d'incrementar l'esforç.

- **Enduriment** (Y - U). Com diu el nom, en aquest tram el cos s'endureix i s'ha d'aplicar més esforç per augmentar la deformació. Al final d'aquest tram s'arriba a l'esforç màxim que aguanta el cos (σ_u).

- **Estricció** (U - R). En aquest punt l'esforç s'ha de reduir, ja que si no, el cos es trencarà. Cada vegada s'ha d'aplicar menys esforç si no es vol que el cos es trenqui. En cas contrari, s'arriba al punt de trencament (R).

En qualsevol d'aquests trams, si l'esforç finalitza abans d'arribar al punt de trencament, el cos recuperarà la deformació elàstica però no la plàstica. **[Figura 2.19]**

Si s'analitza el diagrama de tracció dels ossos s'observa que el seu tram de deformació plàstica és molt curt, és a dir que són fràgils i es trenquen amb molta poca deformació.

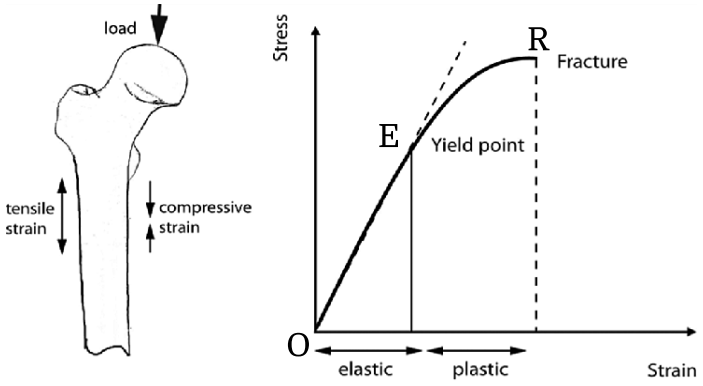


Figura 2.19 - Diagrama de tracció d'un os - *[Wik, TS. 2012]*

3. La física computacional

3.1 Què és la física computacional?

La física computacional és la implementació d'ordinadors per solucionar problemes físics. Fer servir ordinadors en comptes d'una persona és beneficiós perquè un ordinador no comet errors i és aproximadament deu milions de vegades més ràpid que un humà en càlculs aritmètics **[Liqun. 2018]**. Han sigut usats durant dècades per calcular trajectòries de satèl·lits, simulacions de l'univers, paràboles de projectils...

Aquesta manera de calcular problemes físics té les seves limitacions. La principal limitació és la velocitat de l'ordinador. Si l'ordinador que s'utilitza per a fer els càlculs no és potent realitzarà els càlculs de forma relativament lenta i podria trigar dies a completar una simulació simple. Per solucionar aquest problema, científics han utilitzat diverses solucions **[Danese, et al. 2007]**:

- utilitzar superordinadors; uns ordinadors de la mida d'habitacions que poden ser un milió de vegades més potents que el portàtil més ràpid **[IBM. 2023]**;

- fer ús de clústers d'ordinadors; grups d'ordinadors (connectats amb una xarxa d'alta velocitat) que treballen junts per aconseguir una velocitat de processament alta a un cost baix **[Atiquzzaman & Srimani. 2000]**;

- implementant maquinari especialitzat per calcular simulacions; maquinària més barata que fa servir matrius de portes lògiques que poden ser programades específicament segons el camp en què s'utilitzen (Field Programmable Gate Arrays). Aquesta tecnologia relativament recent duplica la seva potència cada any. Comparat amb els microprocessadors dels

superordinadors i els clústers, que segueixen la llei de Moore, que la seva potència es duplica cada dos anys. **[Danese, et al. 2007]**.

La segona limitació de la física computacional és el mateix coneixement de la física. El programa només pot calcular simulacions usant fórmules que ja han sigut descobertes, si no hi ha una fórmula per solucionar un problema l'ordinador no ho podrà simular. Però, el coneixement necessari per a la majoria de simulacions mecàniques és molt bàsic i les fórmules han sigut descobertes fa més de cent anys.

3.2. L'estructura d'un model

3.2.1. Objectes i propietats

Un objecte és qualsevol entitat dins la simulació, des d'una pilota fins a una paret. Depenent de l'objecte tindrà propietats una mica diferents, però tots, més o menys, segueixen el mateix patró. **[Shiffman. 2012. {p.144}]**

Les propietats d'un objecte és informació que es necessita per fer els càlculs. S'emmagatzemen dintre variables que són com capses d'un codi on s'introdueixen dades que poden ser llegides i modificades.

Primer, l'objecte també té propietats físiques, elasticitat i densitat planar. L'elasticitat és l'habilitat d'un objecte per tornar a la seva forma original després d'una deformació. Però amb una simulació computacional, l'elasticitat es fa servir per saber el rendiment de l'energia quan rebota contra objectes. La variable de densitat està mesurada en kg/m^2. Un objecte inamovible tindria una densitat infinita. La massa és una propietat útil d'un objecte però el programa ho calcula a partir de la densitat i l'àrea de la forma de l'objecte.

Moltes altres propietats d'un objecte són vectors (velocitat, acceleració...). Els vectors es poden expressar com vectors polars o cartesians. Un vector polar consisteix en un angle i una magnitud, és més intuïtiu perquè és com és el món real. Però per un ordinador, manipular vectors cartesians de magnituds X i Y és molt més fàcil, fent-ho

també més fàcil pel programador. Per aquesta raó, la majoria de vegades, es fa servir únicament vectors cartesians. **[Shiffman. 2012. {p.31}]**

Un objecte té una posició, una velocitat i una acceleració, totes són expressades com un vector.

- La seva posició està mesurada en unitats reals. Típicament, s'utilitza el sistema internacional, de manera que la posició utilitza metres com a unitat.

- A la variable de velocitat també s'emmagatzema un vector, aquesta vegada en m/s. El primer valor del vector és la velocitat que té l'objecte a l'eix X i el segon valor és a l'eix Y **[Shiffman. 2012. {p.45}]**.

- L'acceleració és igual que la velocitat, fent servir el primer valor per l'eix X i el segon per l'eix Y, però la seva unitat és m/s^2.

L'objecte també té propietats angulars com l'angle en què està girat, mesurat en radians. Velocitat angular en rad/s i acceleració angular mesurat en rad/s^2.

Finalment, un objecte també té una geometria que, depenent de quina forma té, es defineix de diferent manera. Un cercle es defineix amb la variable de radi, sent el centre del cercle la posició de l'objecte. Un rectangle es defineix amb una amplada i una alçada. Per formes més complexes s'utilitza una llista de coordenades que representen vèrtexs. El programa uneix els vèrtexs amb una cadena i crea la forma de l'objecte.

3.2.2. Forces

Dins el món de la física computacional les forces són vectors d'energia que són transmesos a objectes. Igual que l'acceleració, la força es descompon en la força paral·lela a l'eix X i a l'eix Y.

Les forces es poden classificar com a forces constants i forces puntuals. Forces constants sempre són presents a l'objecte. La força de gravetat és constant per unitat

de massa, però no és la mateixa per a cada objecte i s'ha de calcular individualment **[Shiffman. 2012. {p.77-78}]**. Una força puntual és, per exemple, la que es produeix en una col·lisió. La força és aplicada durant un sol instant.

Finalment, hi ha forces externes que no s'originen de dintre la simulació, sinó que són introduïdes per un element exterior com un programa que supervisa la simulació o per l'usuari mateix. Aquestes forces són igual que qualsevol altra, només què es podria dir que apareixen màgicament.

3.2.3. El motor de la simulació

El motor d'una simulació és el director del món simulat. Ell dicta des de la velocitat en què el temps avança (mentre l'ordinador pugui seguir el ritme) fins a la posició de cada objecte. Aquests càlculs els repeteix cada cicle del bucle **[Figura 3.1]**. **[IBM. 2012]** Primer aplica la lògica de la simulació, això vol dir calcular el sumatori de totes les forces. Inclou les forces que han sigut afegides per fonts externes a la simulació i les de dins de la simulació. Quan té el sumatori de la força X i la força Y, fa servir la massa per calcular l'acceleració nova de l'objecte. Aquesta acceleració sobreescriu el valor d'acceleració antic **[Shiffman. 2012. {p.49-51}]**.

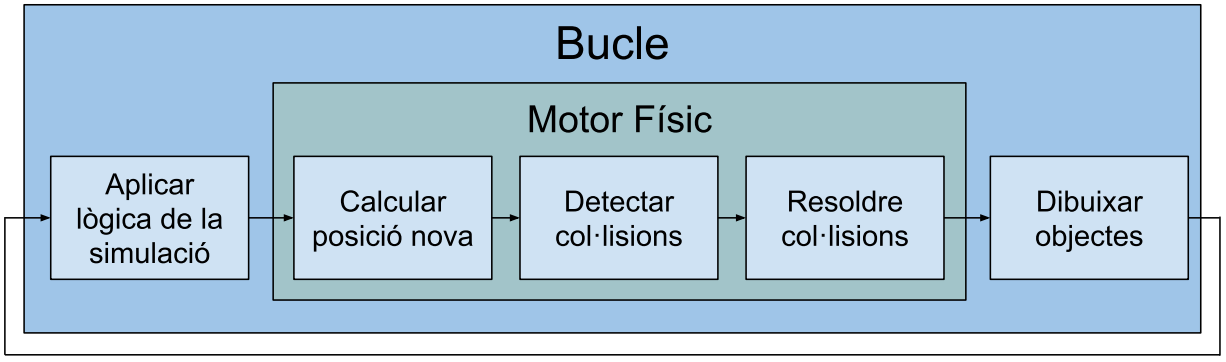


Figura 3.1 - Representació de la feina del motor físic dins el bucle d'una simulació computacional

Després utilitza el valor obtingut d'acceleració i l'increment de temps determinat per al motor per calcular l'increment de la velocitat. Aquest valor se suma al valor de la velocitat prèvia. I finalment, amb la velocitat nova i la posició prèvia calcula les coordenades de la posició nova de l'objecte **[Shiffman. 2012. {p.45-47}]**.

Com que les posicions són calculades independentment dels altres objectes és possible que dos objectes se superposin. Si això passa el motor ho arregla i acaba les seves tasques per aquest cicle.

Finalment, es dibuixen tots els objectes de la simulació. És un dels passos més simples però dels més intensius per l'ordinador, ja que els ordinadors estan fets pel càlcul aritmètic i dibuixar imatges els costa molts recursos. L'ordinador s'està 40% del temps del cicle dibuixant **[IBM. 2012]**, per aquesta raó moltes simulacions estalvien recursos no expressant els resultats gràficament sinó numèricament. I, per finalitzar el cicle, l'ordinador agafa les coordenades de la simulació, ho converteix en píxels i ho dibuixa a la pantalla de l'ordinador **[Shiffman. 2012. {p.193}]**.

3.3. Càlculs físics interns

3.3.1. Simulació dinàmica

Com que la simulació és d'un món que segueix les lleis de la física clàssica, per calcular la dinàmica d'un objecte s'utilitzen les fórmules físiques i es fan servir unitats del món real. Aquestes fórmules es van aplicant progressivament amb els valors obtinguts en la fórmula prèvia.

Per exemple, la força gravitatòria. Aquesta és la seva equació:

$$F_g = m \cdot g$$

Però amb codi es tradueix a això:

```
forceGravity = new PVector(0, object.getMass() * -GLOBAL_GRAVITY);
```

Amb totes les forces d'un objecte s'ha de calcular el sumatori. Com que els vectors són cartesians els càlculs es realitzen per separat per les components X i Y. Per exemple, la fórmula per la suma de totes les forces X seria aquesta:

$$\Sigma F_x = F_{1x} + F_{2x} + F_{3x} + ...$$

Però amb codi es programaria així:

```
for (int i = 0; i < object.forces.size(); i = i+1)
{
    object.totalForce.x =
        object.totalForceX + object.forces.get(i).x;
}
```

Aquest codi crea un bucle en què suma totes les forces de direcció X aplicades a l'objecte i ho guarda al vector de sumatori de forces.

A partir del sumatori de les forces, s'obté l'acceleració. Això ho fa amb la fórmula:

$$a_x = F_x \div m$$

Aquesta fórmula s'escriu així en el codi:

```
object.acceleration.x = object.totalForce.x / object.getMass();
```

Agafa el sumatori de forces de direcció X i ho divideix per la massa de l'objecte. La raó perquè a la fórmula la massa és obtinguda de manera diferent que la força és perquè, com s'ha esmentat a l'apartat 3.2.1, la massa s'obté a partir de l'àrea i la densitat planar. Així doncs, s'ha de cridar una funció que ho calculi.

A posteriori, s'aconsegueix la velocitat fent servir aquesta fórmula:

$$v_{1x} = v_{0x} + a_x \cdot \Delta t$$

Amb codi es representa així:

```
float timeIncrement = TIME_SCALE / frameRate;
object.velocity.x =
    object.velocity.x + object.acceleration.x * timeIncrement;
```

Aquest codi, primer, aconsegueix l'increment de temps. Per a fer això agafa la variable del coeficient de temps que està mesurat en segons de la simulació dividits pels segons de la vida real. Normalment, és igual a 1 perquè es vol que se simuli a una velocitat normal **[Shiffman. 2012. {p.47}]**.

Després es divideix pels fotogrames per segon, que són el nombre de cicles que ha de fer per segon. Amb això s'obté l'increment de temps. Multiplicant-lo per l'acceleració i sumant el resultat a la velocitat prèvia s'obté la nova velocitat.

Finalment, ha de calcular la posició nova, utilitzant la fórmula:

$$x_1 = x_0 + v_x \cdot \Delta t$$

Com que la velocitat es calcula dinàmicament a partir de l'acceleració, es pot fer servir la fórmula de MRU encara que el sistema tingui acceleració. La fórmula traduïda en codi és així:

```
object.position.x =
    object.position.x + object.velocity.x * timeIncrement;
```

Com que la variable *timeIncrement* ja ha estat calculada prèviament no cal tornar-la a calcular. Es multiplica l'increment de temps per la velocitat de direcció X i el resultat se suma a la posició X prèvia de l'objecte per a assolir la nova posició.

Quan acaba de calcular l'acceleració, la velocitat i la posició per l'eix X ho torna a calcular per l'eix Y.

De la mateixa manera es poden calcular les propietats angulars. És a dir, el parell s'aplica al moment d'inèrcia de l'objecte, s'obté l'acceleració angular, la velocitat angular i l'angle.

3.3.2. Col·lisions entre cossos

3.3.2.1 Detecció de col·lisions

Quan es calcula la posició nova d'un objecte no es té en compte la posició dels altres objectes. Això crea la possibilitat que dos objectes ocupin el mateix espai. Com que això és una vulneració de la física, significa que hi ha hagut una col·lisió. Però per poder corregir aquest problema primer s'ha de detectar.

Per a fer això s'han de fer uns càlculs que determinin si dos objectes estan superposats i realitzar aquests càlculs per totes les combinacions possibles de dos objectes.

El primer pas és crear un codi que aparella cada objecte amb cada altre objecte. Això es fa amb un bucle dins un bucle. El bucle exterior agafa el primer objecte i amb el segon bucle l'aparella amb cada objecte. Quan el primer objecte ha estat aparellat amb tots els altres, el bucle exterior acaba el seu cicle i ho repeteix per a la resta d'objectes.

```
for (int i = 0; i < objects.size(); i++)
{
    for (int j = 0; j <  objects.size(); j++)
    {
        checkCollision(objects.get(i), objects. get(j));
    }
}
```

El resultat d'aquest codi es pot representar amb una matriu **[Figura 3.2]**

Però, com es pot veure, aquest codi és molt ineficient. En primer lloc, comprova si un objecte està xocant amb ell mateix (caselles blaves), cosa que és una comprovació absurda. I, en segon lloc, comprova cada parella dues vegades, amb una parella (caselles verdes) i l'oposada (caselles taronges). Per exemple, comprova si l'objecte 1 col·lisiona amb l'objecte 2 i després comprova si l'objecte 2 col·lisiona amb l'objecte 1.

	Objecte 1	Objecte 2	Objecte 3	Objecte 4	Objecte 5
Objecte 1	1 - 1	2 - 1	3 - 1	4 - 1	5 - 1
Objecte 2	1 - 2	2 - 2	3 - 2	4 - 2	5 - 2
Objecte 3	1 - 3	2 - 3	3 - 3	4 - 3	5 - 3
Objecte 4	1 - 4	2 - 4	3 - 4	4 - 4	5 - 4
Objecte 5	1 - 5	2 - 5	3 - 5	4 - 5	5 - 5

Figura 3.2 - Matriu que representa totes les comprovacions fetes pel codi

Per arreglar aquests dos problemes la solució és simple, només cal obviar la meitat de les combinacions, mitjançant aquest canvi a la segona línia:

```
for (int j = 0; j < i; j++)
```

Aquest canvi millora l'eficiència del codi, duplicant la velocitat, i fa que la matriu de comprovacions ara quedi així **[Figura 3.3]**:

	Objecte 1	Objecte 2	Objecte 3	Objecte 4	Objecte 5
Objecte 1					
Objecte 2	1 - 2				
Objecte 3	1 - 3	2 - 3			
Objecte 4	1 - 4	2 - 4	3 - 4		
Objecte 5	1 - 5	2 - 5	3 - 5	4 - 5	

Figura 3.3 - Matriu que representa totes les comprovacions fetes pel codi després de millorar-lo

Ara, és clar com s'aparellen els objectes, però, com es fan els càlculs per detectar la col·lisió?

En el cas d'objectes de geometria circular és simple, es calcula la distància entre centres **[Figura 3.4]** i si és inferior a la suma dels radis significa que els objectes col·lisionen. La diferència entre la suma dels radis i la distància entre centres s'anomena distància de penetració, i dona una mesura del grau de superposició entre els dos objectes. **[Souto. 2016]**

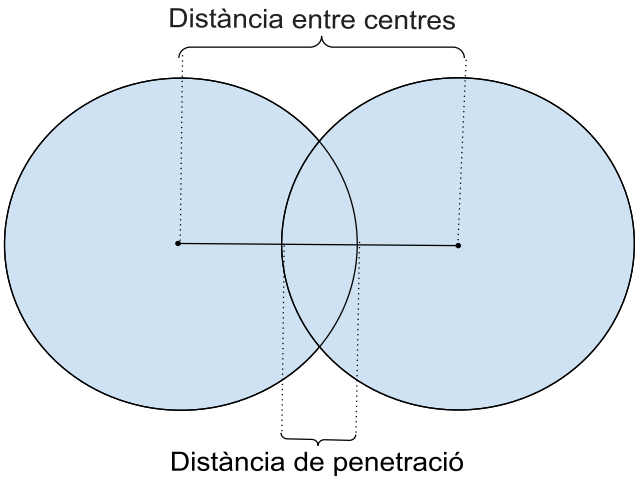


Figura 3.4 - Esquema de dos objectes que mostra la distància entre ells i la distància de penetració

El codi que fa aquest càlcul seria:

```
float getOverpenetration(objectA, objectB){
    float overpenetration;
    float distance;
    distance =
        sqrt(sq(objectA.getPosX() - objectB.getPosX()) +
        sq(objectB.getPosY() - circle2.getPosY()));
    overpenetration =
        objectA.getRadius() + circle2.getRadius() - distance;
  return(overpenetration);
}
```

En els casos d'objectes amb altres geometries és més complex però el principi és el mateix. Es calcula la distància entre el centre i el límit exterior, del primer objecte, en la direcció de l'altre objecte. Després es fa el mateix pel segon objecte. Si el sumatori d'aquestes dues distàncies és més gran que la distància entre els centres dels objectes vol dir que s'estan superposant.

3.3.2.2 Resolució de col·lisions

Una vegada el programa ha detectat que hi ha alguna superposició, hem de saber com ho arregla. Primer s'ha de fer que els objectes ja no es trepitgin. Per fer això el programa fa com si tirés el temps enrere, modificant les posicions dels dos objectes. Els desplaça enrere, en el sentit oposat al que anaven, fins que just no se solapen. Quan ja no s'estan superposant, el motor pot començar a calcular la col·lisió.

Les col·lisions són unes de les parts més complicades de programar. No només s'ha de calcular les forces i els seus sentits resultants, sinó que s'ha de calcular les forces angulars. Per aquesta raó, les col·lisions entre objectes circulars són més simples, ja que el seu angle és irrellevant.
Una col·lisió es calcula usant vectors polars, ja que és la manera més fàcil per aconseguir els valors de força i angle. La figura 3.5 mostra com la força de l'objecte A és distribuïda després de la col·lisió. Els mateixos càlculs es fan per a l'objecte B, i els dos resultats es combinen per obtenir les forces resultants de la col·lisió. **[Normani. 2010]**

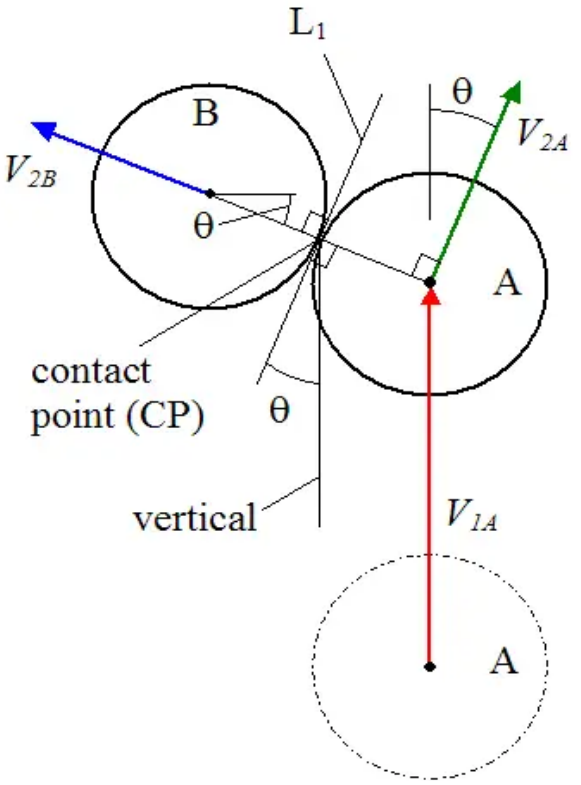


Figura 3.5 - Esquema de les forces durant una col·lisió entre dos cossos - *[Normani, F. 2010]*

Finalment, les forces de l'impacte són convertides en vectors cartesians i aplicades als objectes. Utilitza aquest codi per convertir-les:

```
objectA.force.x = collision.magnitude * cos(collision.angle);
objectA.force.y = collision.magnitude * sin(collision.angle);
```

3.3.3 Restriccions

Perquè un motor físic pugui simular un sistema mecànic cal que tingui en compte que els objectes interactuen entre ells en forma de restriccions. La modificació introduïda al punt anterior permet que el motor tingui en compte les col·lisions entre objectes, és un avanç important, però és la restricció mínima requerida per modelar un sistema mecànic.

Per modelar sistemes mecànics més complexos, que permetin lligar objectes amb cordes, unir-los amb pals, connectar-los amb frontisses, o utilitzar politges, cal incorporar altres restriccions a més de les col·lisions.

Les restriccions són condicions que té un objecte o un grup d'objectes, i que el motor computacional ha de vigilar que es compleixin contínuament. Les restriccions serveixen per controlar com es comporta al món i com interactua amb els altres objectes. Les restriccions s'associen a propietats específiques dels objectes, i per aquesta raó n'hi ha de molts tipus. Però les més típiques afecten les propietats de posició i angle, ja siguin absolutes o relatives a un altre objecte.
Alguns exemples de restriccions habituals són les següents:

- **Unió rígida (barra)**. Aquestes restriccions fixen la distància entre dos objectes perquè sigui constant. Com si estiguessin units per una barra rígida. El motor computacional comprova que la distància entre els dos objectes sigui exactament la distància establerta, i si no ho és, ho corregeix.

- **Unió flexible (corda).** Controla la distància màxima entre dos objectes. La restricció serà vulnerada quan la distància entre els objectes sigui major que l'establerta. És com si els dos objectes estiguin units per una corda perquè deixa que els dos objectes s'apropin, però no deixa que s'allunyin a més d'una certa distància.

- **Unió angular (frontissa)**. És una restricció d'angle i de posició. Uneix dos objectes per un punt determinat i fa que aquests dos punts sempre estiguin superposats. Però a diferència de la unió rígida, deixa que els objectes girin individualment respecte al punt d'unió. Addicionalment, també es pot definir

un angle relatiu màxim i mínim entre els dos objectes per limitar la rotació, igual que la frontissa d'una porta.

Si una restricció és vulnerada el motor computacional l'ha de corregir. Per restriccions de posició ho soluciona de manera semblant a com corregeix-les col·lisions: el motor mourà enrere l'objecte fins que estigui a una posició que no vulnera la restricció. De la mateixa manera, per restriccions d'angle, el motor girarà els objectes fins que no vulnerin la restricció.

3.4. Renderització del model físic

En una simulació 2D hi ha un pla que representa el món. Però com es mesura aquest pla? Depèn de qui estigui fent els càlculs. Si l'està manipulant el motor físic, el pla serà mesurat amb unitats del món real, normalment utilitzant el sistema mètric. Però si l'està dibuixant el programa, el pla serà mesurant en píxels.

Això crea una barrera de comunicació entre el motor físic i el codi que dibuixa la simulació. Això es pot solucionar fàcilment amb un factor de conversió que relacioni píxels amb metres, però en programar s'ha de tenir present si cal fer servir píxels o unitats del món real.

És a dir, aquest factor de conversió determina quants píxels ha d'utilitzar a la pantalla per representar un metre en la simulació. Si el factor de conversió és 1, un píxel representa un metre i, per tant, una pantalla típica de 1920 x 1080 píxels representarà 2 km^2. Però si el factor fos 100, la pantalla només representaria 207 m^2.

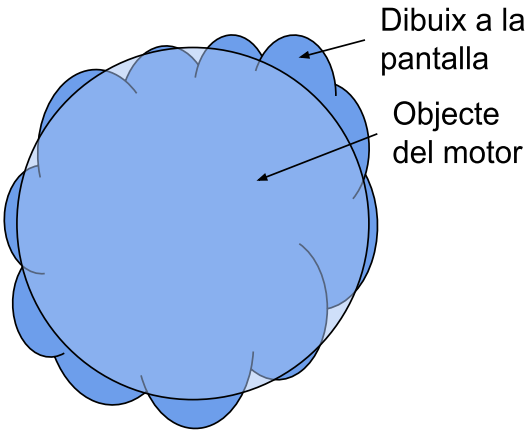


Figura 3.6 - Esquema d'un cos on en la pantalla es veu un núvol, però en el motor físic és una bola

Cal recordar que el motor físic només calcula la simulació però sovint el que volem és una animació, és a dir una representació visual del món simulat. Això

significa que tècnicament a la pantalla no es dibuixa exactament el que calcula el motor físic, sinó una representació del món simulat. Això permet que la simulació utilitzi elements computacionalment més simples però que a la pantalla es visualitzin formes més complexes **[Figura 3.6]**.

Això es pot veure a la figura 3.6, on a la pantalla es mostra un núvol però per facilitar els càlculs que ha de fer el motor, el motor ho calcula com si fos un cercle.
Aquesta tècnica és molt típica en videojocs, ja que el seu rendiment és molt important. Això es pot veure en el concepte *hitbox* que es refereix a la forma de l'objecte que el motor físic calcula. Aquesta tècnica també s'aplica en simulacions en 3D, per exemple el joc *Minecraft* que utilitza una *hitbox* amb forma de cub per calcular la física dels animals **[Figura 3.7]**.

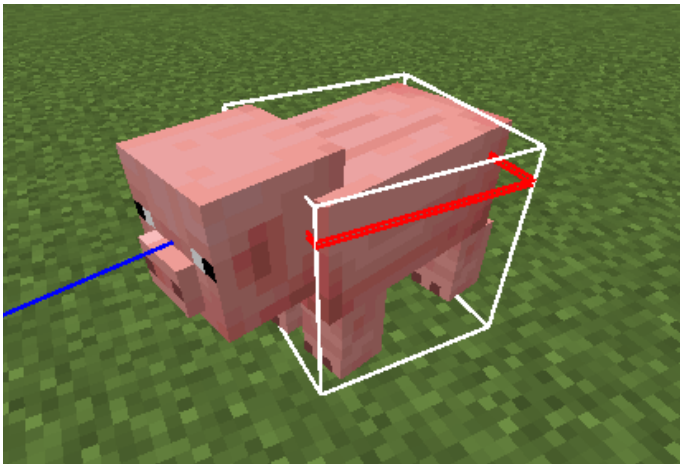


Figura 3.7 - Captura de pantalla del joc *Minecraft* on es veu un animal amb la seva *hitbox* de color blanc

II. Implementació pràctica

4. Motors computacionals físics

4.1. Introducció

Per poder modelar la contracció d'un múscul i els moviments que crea, cal tenir un motor físic que calculi els desplaçaments i moments que el múscul crea a partir de la seva contracció. Per aquesta raó vaig començar a crear un motor físic[4], desenvolupat amb Processing, un entorn de programació basat en Java. Com que un motor físic es crea gradualment, la complexitat del motor també va augmentar gradualment. Però quan vaig arribar al punt on havia de fer que els cossos col·lidissin el nivell requerit per programar això era més alt que el que jo posseïa. Si hagués posat molt de temps podria haver programat les col·lisions entre cercles. Però si volia que el motor calculés també els moments d'elements rectangulars caldria posar molt més temps i, tot i això, era possible que no ho aconseguís.

Dedicar tant temps a programar un motor físic era absurd, ja que se sortia de l'àmbit del treball de recerca que era programar un model muscular i no un motor físic. Finalment, vaig decidir fer servir una llibreria[5] especialitzada que fes la feina del motor físic. Vaig trobar dues llibreries que podien fer aquesta feina: una anomenada "Fisica" feta per Ricard Marxer i una altra anomenada "Box2D for Processing" feta per Daniel Shiffman. Totes dues eren semblants però la darrera estava una mica millor documentada i, per tant, aquesta és la que vaig acabar utilitzant en la programació del model muscular.

[4] Un motor físic és un programa que calcula la simulació d'un sistema físic.
[5] En programació, una llibreria és un conjunt de funcions que proporcionen serveis a un programa independent.

4.2. Motor propi

Encara que al final hagi fet servir un altre motor físic, fer el meu motor va servir per aprendre molt sobre com funcionen. I aspectes de com funcionava el meu motor els vaig poder aplicar a l'hora d'entendre com funciona el motor Box2D.

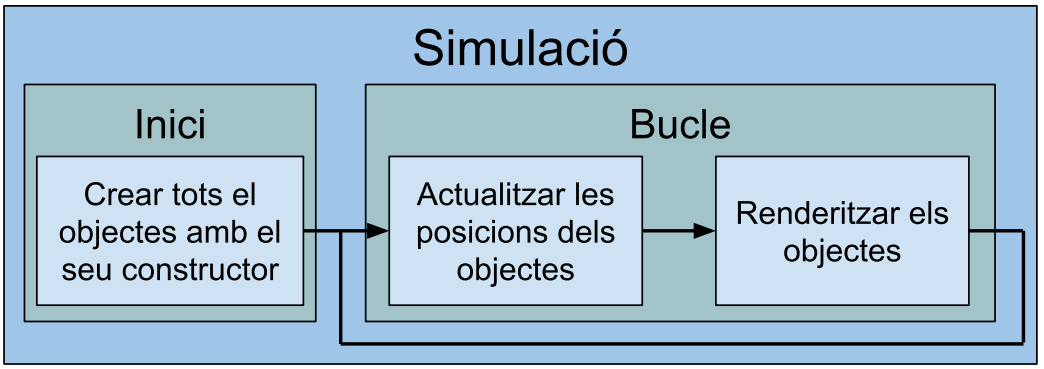


Figura 4.1 - Representació del funcionament d'una simulació

Simular un sistema mecànic consisteix a crear i inserir objectes a la simulació. I després crear un cicle infinit que actualitza les forces i posicions noves de cada objecte, i després visualitzar els objectes en les seves posicions noves. **[Figura 4.1]**

Cada objecte té una funció *update()* que calcula la seva posició nova a partir de la seva posició prèvia, velocitat i acceleració. I una funció *render()* que dibuixa l'objecte a la seva posició, aquesta funció és diferent per cada tipus d'objecte. Per exemple, tots els objectes circumferència tenen la mateixa funció *render()*, però diferent de la funció *render()* dels objectes rectangle.

4.2.1. Motor v0.1 - Propietats geomètriques

El primer pas per crear un model físic és crear objectes que actuïn en el motor físic. Vaig començar creant un objecte amb forma d'una circumferència **[Figura 4.2]** perquè és l'objecte més simple que hi ha, ja que el seu angle de rotació és irrellevant i calcular el seu rebot contra les parets és més simple.

Per crear un objecte s'ha de definir les propietats que té; vaig començar amb les propietats geomètriques. Totes les propietats d'un objecte les vaig emmagatzemar com a variables internes. La posició va ser la primera propietat que vaig crear; posX per l'eix X i posY per l'eix Y. La posició marca el centre de l'objecte, i els valors són a quants píxels està l'objecte del punt (0, 0). Matemàticament, el quadrant (positiu, positiu) és el de dalt a la dreta, però en el Processing aquest quadrant és el de baix a la dreta, això és perquè l'eix Y està invertit. Per no complicar el programa innecessàriament vaig fer servir els mateixos eixos que el Processing.

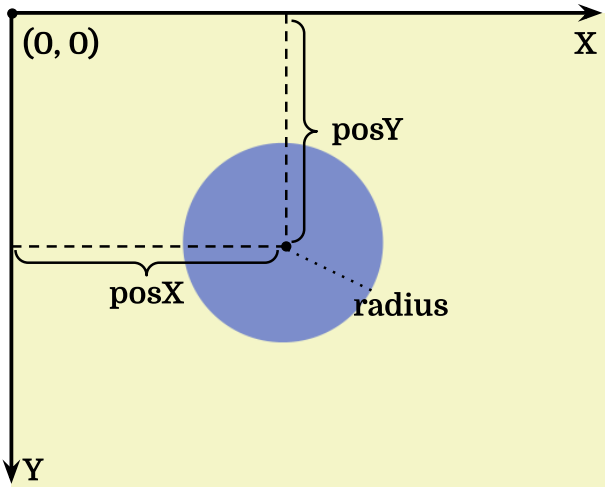


Figura 4.2 - Representació de les propietats geomètriques d'un cos de la simulació

També vaig afegir les propietats de radi i color. En que radi és la distància entre el centre de l'objecte a la seva circumferència. I el color està definit en RGB, on hi ha 3 valors que cada diu quant de vermell, verd o blau hi ha en l'objecte.

4.2.2. Motor v0.2 - Propietats dinàmiques

Ara que té les seves propietats geomètriques cal afegir les seves propietats dinàmiques, velocitat i acceleració.

La velocitat és un valor que emmagatzema quants píxels es mou per cada *frame*. Un *frame* (fotograma en anglès) és un cicle del programa. Com més frames calculi per segon, més ràpid està anant, això està mesurat en frames per segon (fps). La duració d'un frame es pot calcular amb aquesta equació:

$$t_{frame} = 1 / fps$$

Per exemple, en una simulació típica de 60 fps la durada de cada *frame* és de 1/60 s, és a dir 16,7 ms.

Finalment, l'acceleració és un valor de l'increment de la velocitat en cada frame. I es mesura en nombre de píxels per *frame* en que augmenta la velocitat cada *frame*.

Per poder accedir i modificar el valor de les propietats cal crear unes funcions anomenades *getters* i *setters* per cada propietat. Quan executes una funció *getter*, aquesta funció t'envia el valor que hi ha en la variable de la propietat. I quan executes una funció *setter* envies un valor a la funció, i la funció reescriu la variable de la propietat perquè tingui el valor que l'hi has enviat.

Per exemple, així és com ho vaig fer la propietat *posX*:

```
float posX, posY;

float getPosX(){
  return(posX);
}

void setPosX(float recievedValue){
  posX = recievedValue;
}
```

Com he explicat en el punt 4.2., cada objecte té una funció *update()* i una *render()*. En aquest punt, la funció *update()* només actualitzava la velocitat a partir de l'acceleració i després la posició a partir de la nova velocitat. La formula per calcular la nova velocitat és aquesta:

$$v_{1x} = v_{0x} + a_x \cdot \Delta t$$

Però com que l'increment de temps és sempre 1 frame, i no cal que ho multiplica per 1, puc ignorar l'increment de temps. Doncs la fórmula la puc programar com això.

$$v_{1x} = v_{0x} + a_x$$

I si aplico el mateix concepte en la fórmula de la posició, la funció d'update() queda així:

```
void update()
{
  velX = velX + accX;
  velY = velY + accY;

  posX = posX + velX;
  posY = posY + velY;
}
```

La funció render() és molt simple, ja que la seva única feina és dibuixar un cercle en la seva posició. La vaig programar així i no vaig haver de canviar-la durant totes les actualitzacions del codi.

```
void render()
{
  noStroke(); //Que no hi hagi un contorn al cercle
  fill(colour); //Que el color del cercle sigui el definit
  ellipseMode(RADIUS);
  //Dibuixa un cercle a la posició posX i posY del radi definit
  ellipse(posX, posY, radius, radius);
}
```

L'únic problema que té ara el model és que com que no hi ha parets els objectes poden sortir de la pantalla **[Figura 4.3]**. Per arreglar-ho vaig fer una funció que comprovi si l'objecte està fora de la pantalla, i si sí, el posiciona en l'instant abans que surti i modifica la seva velocitat perquè reboti de la paret.

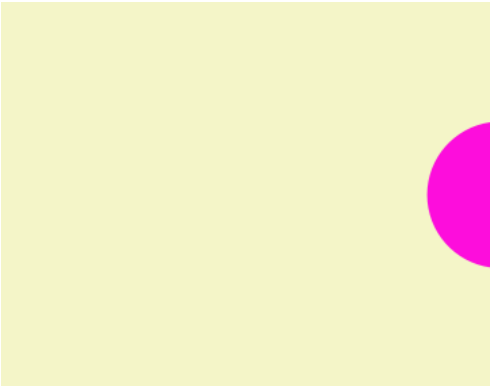


Figura 4.3 - Objecte de la simulació se surt de la finestra per culpa de no haver-hi parets

També vaig afegir una propietat que és el coeficient de restitució que s'aplica a la velocitat nova quan rebota. A continuació mostro la part de la funció per la paret superior de la pantalla, per cada paret el patró és igual però fa servir uns valors diferents:

```
if (c.getBoxTop() < 0)
{
  c.setPosY(c.getRadius());
  c.setVelY(-c.getVelY() * c.getRestitution());
}
```

Amb aquest codi, quan la part superior del cercle està més amunt que el sostre de la finestra, passarà això:

- La bola es mou just a la vora de la finestra;
- Es gira el sentit de la velocitat Y;
- Es multiplica les velocitats pel coeficient de restitució;

Ara els cercles no surten de la pantalla sinó que reboten dins la pantalla. **[Figura 4.4]** **[Figura 4.5]**

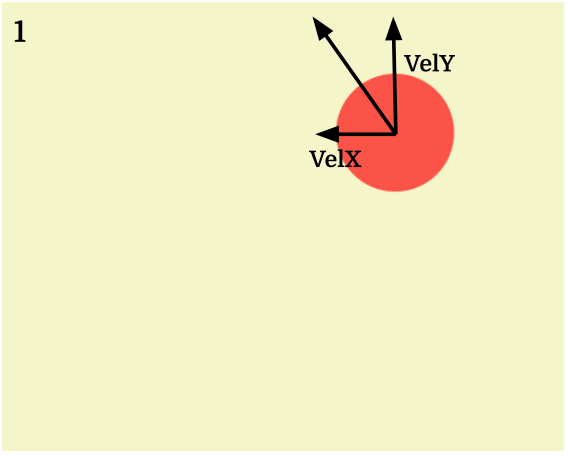


Figura 4.4 - Objecte de la simulació es mou cap a la paret superior de la simulació

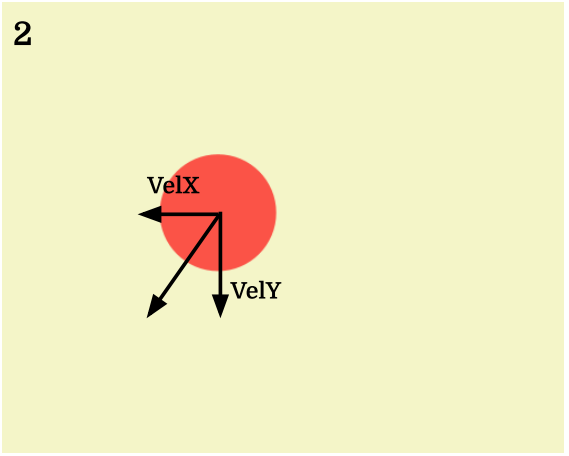


Figura 4.5 - Objecte de la simulació ha rebotat amb la paret superior de la simulació

4.2.3. Motor v0.3 - Massa i forces

Per la següent versió del motor vaig decidir afegir-hi forces, i conseqüentment una propietat de massa. Amb la propietat de massa vaig crear la propietat de densitat superficial (per ser en 2D), quan n'edites una, l'altra s'actualitza perquè concordin.

Per poder calcular les forces vaig crear dues variables: el sumatori de les forces en l'eix X (totalForceX) i el sumatori de les forces en l'eix Y (totalForceY). Aquestes variables s'utilitzen, després, en la funció *update()* per calcular l'acceleració.

Vaig crear unes funcions per poder gestionar les forces d'una manera simple. La primera va ser addForce(), aquesta funció serveix per afegir una força al total. Li envies el valor X i Y de la força nova i la suma al total. Així és com ho vaig fer per afegir la força de la gravetat als objectes:

```
circle.addForce(0, circle.getMass() * GLOBAL_GRAVITY);
```

El primer valor és 0, perquè no hi ha força de gravetat en l'eix X. El segon valor és la massa del cercle multiplicat per la gravetat, perquè la fórmula del pes és:

$$F_p = m \cdot g$$

Finalment, vaig afegir una funció resetTotalForces() que inicialitza les variables del sumatori de forces a 0. Aquesta funció la utilitzo al final del bucle per esborrar totes les forces al final de cada frame. Com que tracto totes les forces només durant un frame (al final del qual s'esborren), les forces constants com la de la gravetat s'han d'afegir al principi de cada frame.

Ara que ja he creat totes les funcions necessàries, la funció update() ha d'utilitzar les variables totalForceX i totalForceY per calcular l'acceleració. Només cal modificar la funció de la versió 0.1 perquè calculi l'acceleració utilitzant la segona llei de Newton:

$$F = m \cdot a$$

I aïllant l'acceleració s'obté la fórmula:

$$a = F \div m$$

Ara la funció update() és així:

```
void update()
{
  accX = totalForceX / mass;
  accY = totalForceY / mass;

  velX = velX + accX;
  velY = velY + accY;

  posX = posX + velX;
  posY = posY + velY;
}
```

En la versió 0.1 del motor l'acceleració estava declarada arbitràriament. Però ara, en la versió 0.2, l'acceleració està calculada a partir de la massa i les forces aplicades.

4.2.4. Motor v0.4 - Unitats reals

Fins ara el motor físic no utilitza les unitats del sistema internacional, sinó que utilitza píxels per distància i frames per temps. Perquè el motor físic serveixi per simular el món real, haig d'utilitzar unitats reals. Però això no és tan fàcil com sembla, perquè l'ordinador sempre treballa amb píxels i frames i no entén metres i segons.

Per arreglar això calen dues variables:

- Una és SPACE_SCALE que és l'escala espacial que dius quants píxels hi ha en un metre (píxels / metre).

- L'altre és TIME_SCALE i diu quants *frames* hi ha en un segon (fps). Aquesta variable serveix per dir quants segons simulats hi ha en cada segon real. Això fa que puguis simular la simulació a càmera ràpida o càmera lenta.

Ara que el motor físic **[Annex A]** té tots els factors de conversió, pot fer tots els càlculs físics en unitats del sistema internacional. Però l'increment de temps s'ha

d'ajustar amb la variable TIME_SCALE i a l'hora de visualitzar l'objecte, les dimensions i posicions s'han de multiplicar per SPACE_SCALE.

Això serveix per poder fer servir valors del món real per definir objectes i forces en la simulació. Per exemple el valor de la gravetat de la simulació no cal que sigui un valor arbitrari, sinó ara és $9{,}81\,m/s^2$.

4.2.5. Motor v0.5 - Col·lisions

La simulació de les col·lisions és un aspecte important dels motors físics, sense la qual els objectes es poden travessar i no interactuen entre ells. Però el seu càlcul no és trivial; després de fer recerca vaig decidir dividir la gestió de col·lisions en tres parts:

- **Detecció** - Aquesta part detecta quan dos objectes s'estan solapant. Si dos objectes es trepitgen, vol dir que hi hauria hagut d'haver una col·lisió.

- **Correcció del solapament** - En el món real els objectes no se solapen. Però com que la duració d'un *frame* mai serà prou petita, en les col·lisions sempre hi haurà una mica de solapament. Aquest pas arregla el solapament entre els cossos movent-los enrere just a l'instant abans de xocar.

- **Resolució** - Ara que els cossos estan just a l'instant abans de xocar, es pot calcular les velocitats noves de cada cos causat per la col·lisió.

Els càlculs matemàtics i el codi corresponent es mostra amb més detall en els següents punts.

4.2.5.1 Detecció de col·lisions

El primer pas és el de detecció. Per fer això creo una funció que mira el solapament. Si el solapament és positiu ha detectat una col·lisió, si el solapament és negatiu vol dir

que no hi ha solapament i, per tant, no hi ha col·lisió. **[Figura 4.6]**

Aquesta funció (*getOverpenetration()*) calcula el solapament de dos objectes. Per calcular el solapament, primer cal saber la distància entre els centres dels 2 objectes. La qual es calcula amb el teorema de Pitàgores.

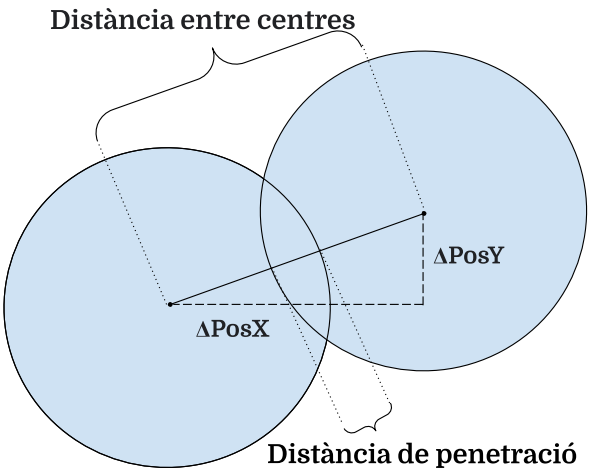


Figura 4.6 - Esquema de dos objectes que mostra la distància entre ells i la distància de penetració

$$Distància = \sqrt{((posX_a - posX_b)^2 + (posY_a - posY_b)^2)}$$

I després, la suma dels radis dels 2 cossos menys la distància és la magnitud de solapament entre els cossos. Aquesta funció la vaig fer així:

```
float getOverpenetration(RoundShape circle1, RoundShape circle2)
{
  float overpenetration;
  float distance;

  distance = sqrt(sq(circle1.getPosX() - circle2.getPosX()) +
                  sq(circle1.getPosY() - circle2.getPosY()));
  overpenetration = circle1.getRadius() + circle2.getRadius() -
                    distance;

  return(overpenetration);
}
```

En la línia 6 i 7, el codi, calcula la distància entre els dos cossos. I després en la línia 8 i 9, utilitza la distància per calcular el solapament. I quan el té calculat, retorna el resultat.

A partir d'aquesta funció vaig crear la funció *isCollisionDetected()* que retorna *true* o *false* segons si el solapament és positiu o negatiu.

```
boolean isCollisionDetected(RoundShape circle1,
                            RoundShape circle2)
{
  if (getOverpenetration(circle1, circle2) > 0.000)
  {
    return(true);
  }
  else
  {
    return(false);
  }
}
```

4.2.5.2 Correcció del solapament

Ara que el codi pot realitzar el primer pas, podem fer el segon. Per corregir el solapament, els dos objectes s'han de moure enrere fins que ja no se superposin. Calen dues variables per aconseguir això, el solapament en l'eix X i el solapament en l'eix Y. Anomenades SolS i SolY en la figura 4.7.

No volia fer servir les funcions trigonomètriques per calcular el solapament en cada eix, doncs ho vaig fer mitjançant un factor de conversió lineal entre la distància entre cercles i el solapament, ja que tots dos creen un triangle semblant i sé les magnituds de la hipotenusa **[Figura 4.7]**. La hipotenusa del triangle de posició és la distància entre els objectes i l'aconsegueixo igual que en el pas previ, amb el teorema de Pitàgores. I la hipotenusa del triangle de solapament és el solapament i l'aconsegueixo amb la mateixa funció que en el pas previ.

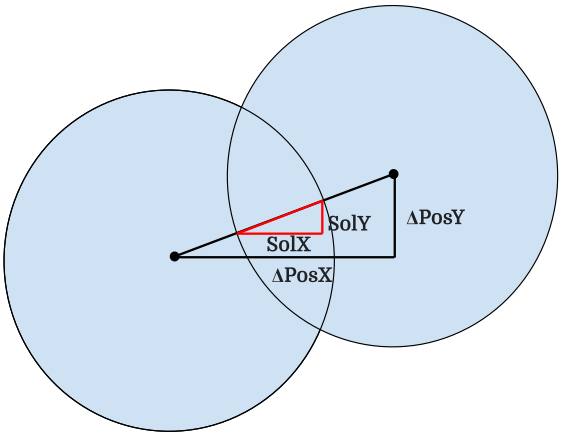


Figura 4.7 - Esquema de dos objectes que mostra que l'angle entre centres és igual que l'angle del solapament

$$\textit{Factor de conversió} = \textit{Solapament} \div \textit{Distància}$$

Ara, s'agafa la magnitud de la distància entre cossos en cada eix i es multiplica pel factor de conversió per obtenir el solapament en cada eix.

$$\textit{Solapament\_X} = |\Delta \textit{Pos\_X}| \cdot \textit{Factor\_de\_conversió}$$

En codi ho vaig escriure així:

```
totalClipX = abs(circle2.getPosX()-circle1.getPosX())
            *linialScaleFactor;
```

La funció abs() agafa el valor absolut d'un nombre.
Tècnicament, ara només s'ha de moure cada bola la meitat de SolX en l'eix X i la meitat de SolY en l'eix Y per corregir el solapament. Però si un objecte té un moment menor que l'altre no té sentit que els mogui la mateixa distància.

Per arreglar això, en calcular quan un objecte s'ha de moure, SolX i SolY es multiplica pel moment de l'objecte dividit entre el sumatori dels moments. Per tant, vaig crear una funció a l'objecte que calcula el moment:

```
float getMomentum(){
  float momentumX = mass * velX;
  float momentumY = mass * velY;
  return(sqrt(sq(momentumX) + sq(momentumY)));
}
```

I vaig fer servir la funció getMomentum() per calcular el moment de l'objecte 1 entre el sumatori de moments:

```
momentumRatio = circle1.getMomentum()
               /(circle1.getMomentum() + circle2.getMomentum());
```

Finalment, a l'hora de moure els cossos a les seves posicions noves vaig fer servir la funció, creada en la versió v0.1, *setPosX()*.

```
if (circle1.getPosX()<circle2.getPosX()){
  circle1.setPosX(circle1.getPosX()-(SolX*momentumRatio));
  circle2.setPosX(circle2.getPosX()+(SolX*(1-momentumRatio)));
}
else{
  circle1.setPosX(circle1.getPosX()+(SolX*momentumRatio));
  circle2.setPosX(circle2.getPosX()-(SolX*(1-momentumRatio)));
}
```

La primera línia comprova si l'objecte 1 està a l'esquerra de l'objecte 2. L'objecte que estigui més a la dreta es mourà a la dreta i l'objecte que està més a l'esquerra es mourà a l'esquerra. A l'hora de moure l'objecte 2, en comptes de tornar a calcular la proporció de moment, resto la proporció de l'objecte 1 d'1.

$$Proporció\_de\_moment\_del\_cos\_2 = 1 - Proporció\_de\_moment\_del\_cos\_1$$

Aquest codi s'ha de repetir per l'eix Y.

4.2.5.3 Resolució de la col·lisió

Ara el motor sap detectar quan hi ha una col·lisió i corregir el solapament. Només cal calcular les velocitats noves dels cossos causades pel rebot de la col·lisió. Les noves velocitats es calculen a partir de llei de la conservació del moment:

$$m_a \cdot |v_a| + m_b \cdot |v_b| = m_a \cdot |v_a'| + m_b \cdot |v_b'|$$

I l'equació del coeficient de restitució:

$$k\,(v_a - v_b) = -v_a' + v_b'$$

On k és el coeficient de restitució que està determinat arbitràriament en crear l'objecte.

Amb aquestes dues fórmules, es pot obtenir les fórmules per la velocitat dels dos cossos després de les col·lisions:

$$v_a' = (m_b \cdot v_b + m_a \cdot v_a) \div (k \cdot v_a \cdot m_b - k \cdot v_b + m_b + m_a)$$

$$v_b' = (m_a \cdot v_a + m_b \cdot v_b + m_a \cdot k \cdot v_a - m_a \cdot k \cdot v_b) \div (m_a + m_b)$$

Quan les velocitats noves s'han calculat, s'ha d'extreure els seus components X i Y utilitzant el nou angle i aplicar-los a cada objecte amb la funció *setVelocityX()* i *setVelocityY()*.

```
circle1.setVelocityX(newVelocity.x);
circle1.setVelocityY(newVelocity.y);
```

En arribar a aquest punt la programació es va començar a complicar i els resultats no van ser satisfactoris.

4.2.6. Resultat i conclusió

Es va dedicar molt temps a mirar de millorar el motor però a partir d'un punt va quedar clar que no valia la pena avançar en aquesta direcció. Fins i tot si arreglava l'error, encara calia afegir-hi la simulació de col·lisions per altres elements amb geometries més complexes, que tenien rotació i moment.

Per tant, no semblava raonable dedicar més temps a crear un model físic, ja que se sortia de l'àmbit del treball de recerca que era modelar un model musculoesquelètic.

Tot i això, el resultat va ser positiu, vaig poder desenvolupar un motor físic que podia simular la dinàmica i interacció d'un conjunt de circumferències sota l'efecte de la gravetat i forces inicials **[Figura 4.8]**.

A més, haver dedicat tot aquest temps a programar un motor físic propi em va permetre aprofundir molt el meu coneixement sobre el funcionament intern dels motors físics, cosa que em va ajudar molt en el següent punt.

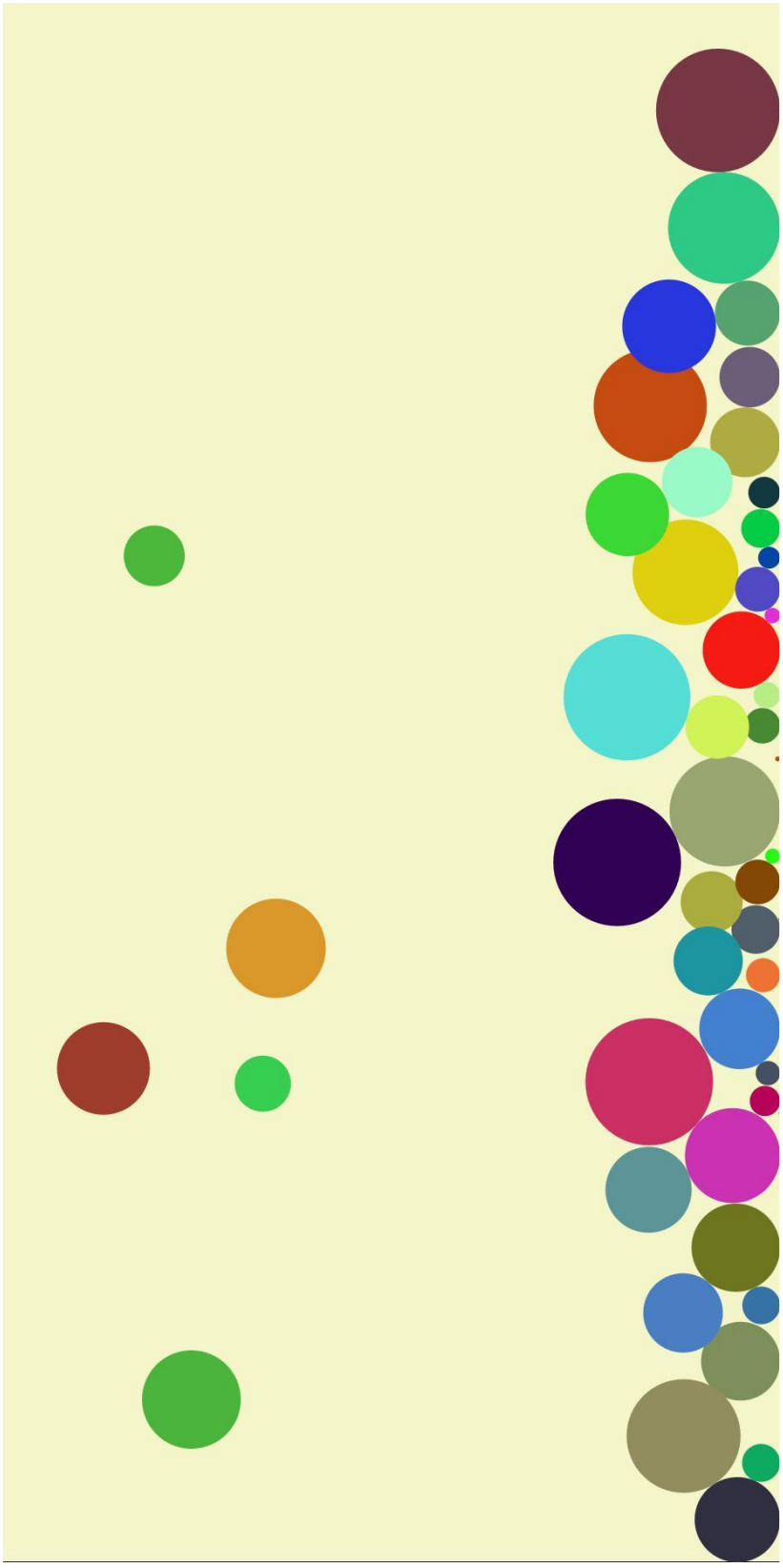


Figura 4.8 - Simulació de 50 boles utilitzant l'última versió del motor propi

4.3. Motor PBox2D

El Box2D és un motor físic en 2D creat per Erin Catto en C++, que va ser publicat *open source* en l'any 2007 i des de llavors no ha parat de ser actualitzada. És un motor relativament simple però molt potent i versàtil per la raó que només fa càlculs físics. Si vols que el teu programa faci una cosa més que càlculs físics ho hauràs de fer tot tu. Aquest motor físic ha sigut utilitzat per crear jocs com *Happy Wheels* i *Angry Birds*.

Com que era un motor tan bo, ràpidament, altres programadors van traduir-lo a altres llenguatges de programació. Un d'aquests llenguatges en que van convertir el motor va ser Java amb el nom de JBox2D. I com que el llenguatge Processing està basat en Java, va ser fàcil crear un *wrapper*[6] de manera que el Processing pot entendre's amb el motor JBox2D. Això és el que va fer el programador Daniel Shiffman en crear la llibreria PBox2D.

Com que el PBox2D és una llibreria, per fer-la servir només cal importar-la:

```
import shiffman.box2d.*;
```

I després s'ha d'importar el JBox2D, en què no s'importa la llibreria sencera, sinó els segments que necessitis:

```
import org.jbox2d.segment.*;
```

4.3.1. Objectes i propietats

El PBox2D funciona semblant al meu motor físic, però la manera de controlar el motor i defineixes els objectes és molt diferent. Una de les diferències claus és com defineix un objecte. El PBox2D defineix un objecte amb 3 parts; un *body* (cos), una

[6] Un *wrapper* és un programa embolcall que fa d'intermediari entre un codi i una llibreria.

shape (forma) i un *fixture* (enllaç). Cada una de les quals té les seves propietats. **[Shiffman. 2012. {p.194}]**

Un *body* té la propietat del tipus de cos que sigui. Hi ha 3 tipus de cossos **[Shiffman. 2012. {p.198-199}]**:

- Un cos **dinàmic** és un objecte normal. Pot caure, rebotar i empènyer altres objectes.

- Un cos **estàtic** no pot ser mogut. No cau per causa de la gravetat i quan interactua amb altres objectes es comporta com si tingués una massa infinita.

- Un cos **cinemàtic** és com un cos estàtic, però pot ser mogut. No és mogut per altres cossos, sinó manualment, i això modifica les propietats d'acceleració, velocitat i posició de l'objecte.

El *body* també té altres propietats que es poden definir al començament i editar i llegir durant la simulació.

Tipus de Variable	Nom	Que és
BodyType	*type*	Tipus de cos
Vec2	*position*	Vector de posició del cos
Vec2	*linearVelocity*	Vector de la velocitat lineal
float	*linearDamping*	Valor de l'amortiment lineal
float	*gravityScale*	Escala de la gravetat de l'objecte

La *shape* d'un objecte conté la geometria de l'objecte, és com la carcassa de l'objecte que interacciona amb l'exterior. S'encarrega d'emmagatzemar la informació de la geometria que serveix per calcular les col·lisions contra altres objectes **[Shiffman. 2012. {p.194}]**. Segons la forma que tingui l'objecte es crearà d'una manera o d'una altra.

Nom del Constructor	Que és	Paràmetres de definició
ChainShape	Cadena de punts	Matriu de vectors de posició
CircleShape	Cercle	Radi
EdgeShape	Pla entre 2 punts	2 vectors de posició
PolygonShape	Cadena tancada de punts	Matriu de vectors de posició

Finalment, hi ha la *fixture* que connecta el *body* amb la *shape* i emmagatzema les propietats de densitat, fricció i elasticitat. **[Shiffman. 2012. {p.194 & 201}]**

Tipus de Variable	Nom	Que és
float	*friction*	Coeficient de fricció
float	*restitution*	Coeficient de restitució
float	*density*	Densitat de l'objecte en kg/m^2
Filter	*filter*	Filtre de col·lisions

Que un objecte es defineixi amb tres parts serveix per optimitzar el codi, ja que objectes diferent poden compartir parts. Si vols crear dues pilotes iguals però una ha de rebotar molt i l'altre gens, només cal que tinguin un *fixture* diferent perquè poden compartir el *body* i la *shape*.

4.3.2. Col·lisions

Aquest motor físic és capaç de calcular les col·lisions entre objectes, i molt més eficientment que el meu motor ho podria haver fet. Això es pot observar en l'exemple de la llibreria *CollisionsAndControlInterface* on calcula les col·lisions entre molts cercles i un quadrat sense reduir la velocitat de l'ordinador significament.

A més a més, no està limitat a cercles **[Figura 4.9]**, també pot calcular col·lisions entre polígons irregulars **[Figura 4.10]**.

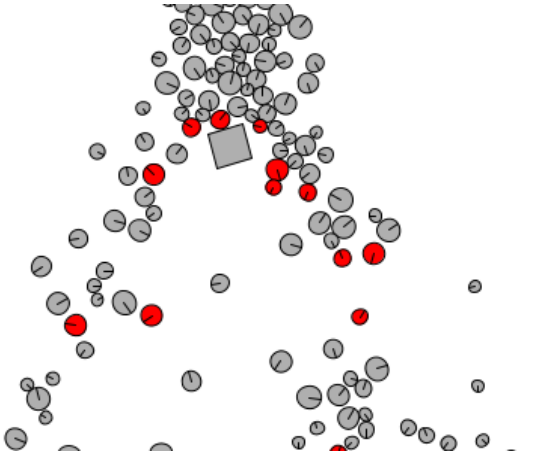


Figura 4.9 - Simulació física del PBox2D on boles cauen i xoquen entre elles i amb un quadrat al centre

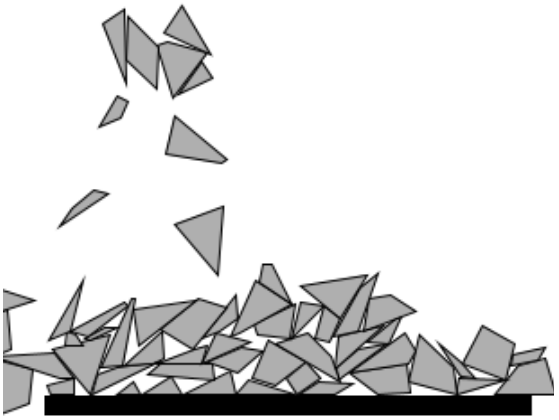


Figura 4.10 - Simulació física del PBox2D on polígons irregulars cauen i xoquen entre elles i amb el terra

El PBox2D gestiona les col·lisions ell mateix, però proporciona funcions i variables que pots fer servir per tenir més control sobre el motor físic. Per exemple, si vols que no calculi les col·lisions d'un objecte, en crear la *fixture*, has de posar la variable *isSensor* com a *true*.

```
fixtureDef.isSensor = true;
```

Si vols detectar quan hi ha col·lisions, per exemple per canviar el color de l'objecte, ho pots fer. Senzillament cridant la funció *listenForCollisions()* dins la funció *setup()*.

```
void setup(){
  box2d.listenForCollisions();
}
```

Una vegada fet això, pots definir la funció *beginContact()* que es crida cada vegada que hi ha una col·lisió i envia la informació dels dos objectes que han xocat.

4.3.3. Rotació i moment

Quan un cos xoca amb un altre, si la força del xoc no ha sigut directament al centre, es produirà un moment de rotació. Per aquesta raó la dinàmica i les forces angulars

són importants a l'hora de simular col·lisions. La dinàmica angular és calculada pel PBox2D però, igual que les col·lisions, el PBox2D proporciona variables i funcions que serveixen per poder controlar aquest apartat de la simulació.

Tipus de Variable	Nom	Que és
float	*angle*	Angle del cos
float	*angularVelocity*	Velocitat angular del cos
float	*angularDamping*	Amortiment lineal

Les primeres dues variables de la taula són les propietats que tindrà l'objecte just a l'inici de la simulació. Quan la simulació avanci aquests valors s'actualitzaran. El tercer paràmetre és constant durant tota la simulació, i canviarà el comportament angular del cos.

Tipus de Funció	Nom	Que és
void	*applyTorque()*	Afegir parell al cos
void	*applyAngularImpulse()*	Afegir un impuls al cos
float	*getInertia()*	Obté la inèrcia del cos

També inclou funcions que es poden utilitzar en qualsevol moment durant la simulació. Les dues funcions primers de la taula apliquen una força angular a l'objecte i, per tant, afecten a la simulació. La tercera és una funció *getter* que només serveix per llegir el valor de la inèrcia del cos, i doncs, no canvia la simulació.

4.3.4. Eficiència computacional

Els motors físics han de fer molts càlculs per segon; això fa que ordinadors normals els hi pugui costar executar la simulació. Però si la quantitat de càlculs es pot reduir, ordinadors menys potents podrien executar simulacions més complexes. Per aquesta raó el PBox2D ofereix maneres de programar la simulació perquè utilitzi menys recursos computacionals.

Una de les maneres que es pot millorar l'eficiència de la simulació és no comprovant les col·lisions de tots els objectes. Si un objecte està estàtic i no té res a prop seu no cal que es comprovi per si xoca amb una cosa. Per fer permetre això, els cossos es poden posar a 'dormir'. Si un cos està dormint, el motor físic no calcularà les seves col·lisions, forces, acceleració i velocitat. Per posar un cos a dormir només cal cridar la funció *setAwake()* amb el paràmetre *false*:

```
setAwake(false);
```

I per tornar-lo a despertar, es crida la mateixa funció però amb el paràmetre *true*:

```
setAwake(true);
```

D'una manera similar, si vols que no calculi col·lisions entre uns cossos però que encara calculi les forces, acceleració i velocitat es pot crear l'enllaç de l'objecte amb un filtre. Aquest filtre és una llista de tipus d'objectes amb que l'objecte no xocarà. El motor físic no perdrà temps comprovant col·lisions entre aquests tipus d'objectes i farà que la simulació sigui més eficient.

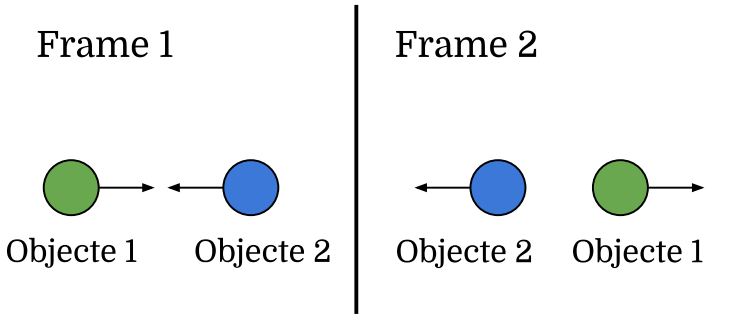


Figura 4.11 - Representació de com dos objectes es poden travessar quan hi ha un fps vaig

L'última manera per millorar la simulació, és una solució més eficient a la detecció de col·lisions, que és un problema habitual en les simulacions. Concretament, si un objecte (objecte-1) que es mou molt de pressa i va a xocar amb un altre (objecte-2), és possible que en el temps d'un frame l'objecte-1 travessi l'objecte-2 **[Figura 4.11]**.

La manera més simple d'arreglar això és fer que un frame tardi menys temps, doncs en la mateixa quantitat de temps que abans hi haurà més frames. Això fa que la

probabilitat que l'objecté-1 se superposi amb l'objecte-2 en algun d'aquests frames sigui major i, per tant, també de què el motor detecti la col·lisió **[Figura 4.12]**.

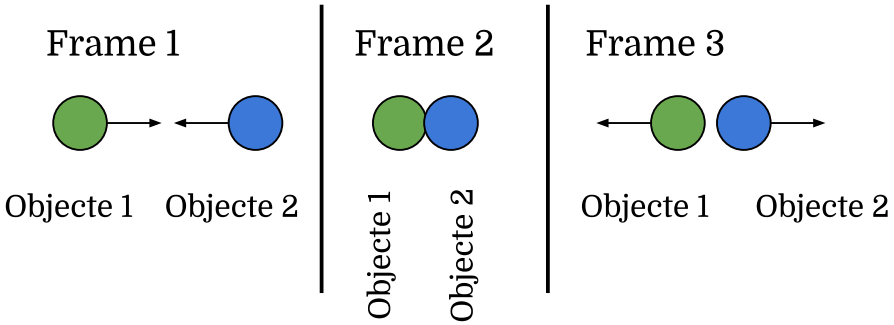


Figura 4.12 - Representació de com dos objectes reboten (no es travessen) quan el fps és alt

El problema d'aquesta solució és que no és gens eficient, ja que suposa calcular i actualitzar la simulació més vegades per segon. El consum computacional de la simulació augmenta molt per la petita millora que produeix. I com més ràpid sigui l'objecte, més recursos haurà de consumir per calcular-ho. Afortunadament, el PBox2D ofereix una altra solució més eficient: definir determinats objectes com de tipus *bullet*.

Pels objectes *bullet*, el motor físic detecta una col·lisió extrapolant el seu camí matemàticament. És a dir, calcula l'equació de la trajectòria entre dos *frames* i resol analíticament si dues trajectòries es creuen. Un *body* de tipus *bullet* utilitza més recursos de l'ordinador que un cos normal, però menys recursos que la primera opció de calcular més frames per segon.

Totes aquestes maneres per optimitzar l'eficiència computacional del cos fan que l'ordinador on s'executa la simulació pugui executar simulacions més complexes amb el PBox2D que amb un motor físic més simple. O que en la mateixa simulació, el programa tingui recursos computacionals de sobres per fer altres coses com millorar la qualitat de la visualització o augmentar els frames per segon i tenir una simulació més precisa.

5. Implementació del model musculoesquelètic

5.1. Introducció del *framework*

Un model musculoesquelètic és un sistema mecànic d'ossos, músculs, tendons i lligaments que treballen junts per crear forces i realitzar moviments. Per programar aquest sistema, la manera més intuïtiva és utilitzant la programació orientada a objectes.

La programació orientada a objectes és un paradigma de programació que està basat en concepte de classes. Les classes són un conjunt de propietats i accions, els quals es poden accedir des de fora si estan declarades com a públiques. Un objecte és una instància d'una classe general que defineix els objectes i accions.

Per tant, utilitzant la programació orientada a objectes, es van crear classes que modelin la seva contrapart real **[Figura 5.1]**. Concretament, hi ha tres classes que representen els cossos físics: l'os, el pes i el múscul.

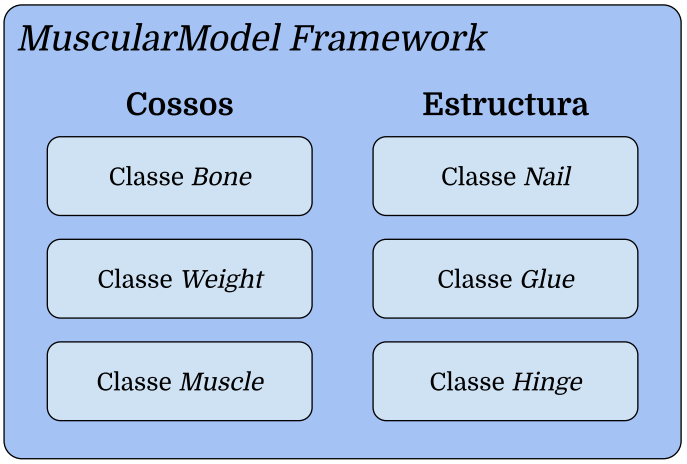


Figura 5.1 - Diagrama de les classes que formen el *MuscularModel Framework*

Un os (*Bone*) està modelat com un objecte allargat rígid amb massa i inèrcia. Un pes (*Weight*) és modelat com un disc sòlid que té una massa. Finalment, un múscul (*Muscle*) es modela com un objecte actiu amb forma d'el·lipse. Es pot contraure i generar tensió com una molla.

També hi ha tres classes més que modelen elements estructurals. La classe *Nail*, que serveix per fixar el model al món, es representa com un punt estàtic. La classe *Glue* modela una unió fixa entre dos objectes. I l'última classe és la *Hinge* que modela una unió entre dos objectes que permet la rotació relativa entre ells.

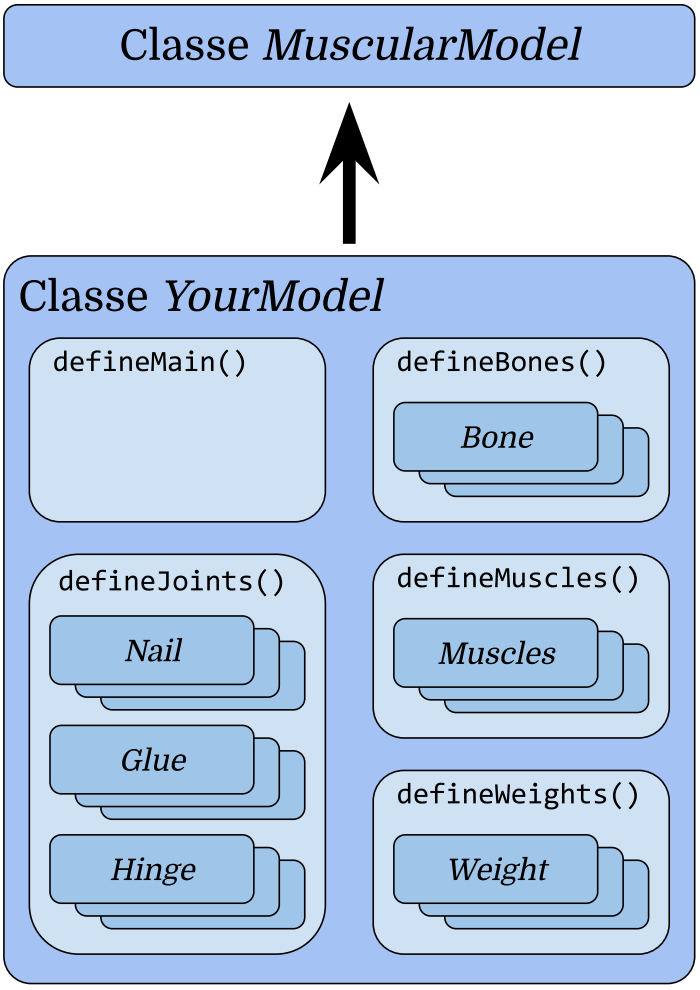


Figura 5.2 - Diagrama del contingut de la classe abstracta del model que estén la classe *MuscularModel*

Addicionalment, per facilitar la creació d'un model, s'ha creat una classe abstracta (*MuscularModel*) que conté la majoria de propietats i funcions que cal per crear un model. Però, algunes funcions s'han deixat en blanc perquè són específiques del model concret que es vulgui crear.

Quan es vol crear un model, només cal crear una classe nova **[Figura 5.2]** que estengui la classe abstracta *MuscularModel* i després, en la teva classe, definir les funcions buides. Dintre de cada funció només cal crear i definir els elements corresponents del teu model.

Per tant, el nucli d'aquest projecte ha consistit a desenvolupar un *framework*[7] de programació que permet crear models musculoesquelètics i simular-los físicament d'una manera senzilla només amb nocions bàsiques d'anatomia i programació orientada a objectes.

[7] Un *framework* és una infraestructura de classes i funcions que proporciona un esquelet per crear una aplicació.

5.2. Els objectes

5.2.1. La classe *Nail*

Introducció:

Aquesta classe serveix per fixar un objecte al món de forma que no caigui. És un objecte imprescindible per poder ancorar una part del model musculoesquelètic al món.

En el model físic es modela com un quadrat minúscul i immòbil. En visualitzar-lo a la pantalla es mostra com un cercle amb un diàmetre de 5 píxels.

Constructor:

El constructor[8] bàsic de l'objecte *Nail* requereix com a paràmetres la seva posició (X i Y), i opcionalment el seu nom.

`Nail (`	Valor per defecte	Descripció
`String name,`	`"Unknown Nail Joint #000"`	Nom identificatiu del clau.
`float posX,`		Coordenada X de la posició inicial.
`float posY)`		Coordenada Y de la posició inicial.

Atributs:

Constants	Valor	Descripció
`color NAIL_COLOR`	`color(255, 255, 255)`	Color del clau.

[8] Un constructor és una funció especial d'una classe que serveix per crear un objecte de la.

Propietats	Descripció
`Body body`	Box2D body (objecte estàtic de la simulació).
`String name`	Nom identificatiu del clau.
`float posX`	Coordenada X de la posició actual.
`float posY`	Coordenada Y de la posició actual.

Mètodes:

Mètode	Descripció
`void display(float displayScale)`	Visualitza el clau en la posició calculada pel PBox2D.

5.2.2. La classe *Bone*

Introducció:

Aquesta classe serveix per modelar els ossos del model musculoesquelètic. És un objecte imprescindible tant com a element estructural com perquè és l'element on els músculs s'uneixen i apliquen les forces.

El model físic el representa com una capsa rectangular amb un pes calculat a partir del volum, derivat de les seves dimensions, i la densitat mitjana dels ossos (1900 kg/m^3 **[Fathima. 2019]**). En visualitzar-lo a la pantalla es mostra com un rectangle blanc amb les cantonades arrodonides **[Figura 5.3]**.

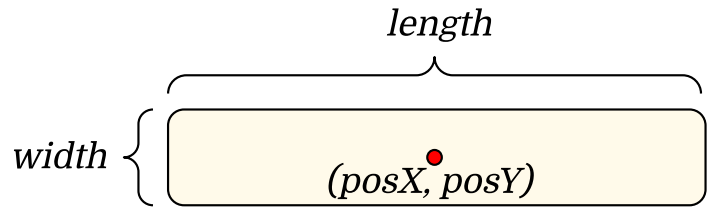


Figura 5.3 - Diagrama de l'objecte *Bone* amb les propietats de posició i mida

Constructor:

El constructor bàsic de l'objecte *Bone* requereix com a paràmetres la seva posició (X i Y) i les seves dimensions (llargada i amplada), i opcionalment el seu nom, el seu gruix i si l'os és estàtic o no.

`Bone (`	Valor per defecte	Descripció
`String name,`	`"Unknown Bone #000"`	Nom identificatiu de l'os.
`float posX,`		Coordenada X de la posició inicial.
`float posY,`		Coordenada Y de la posició inicial.
`float length,`		Llargada de l'os.
`float width,`		Amplada de l'os.
`float depth,`	`width`	Gruix de l'os.
`boolean isStatic)`	`false`	Si l'os és estàtic o no

Tot i que el model musculoesquelètic és bidimensional, cal tenir en compte el gruix per poder calcular el seu pes a partir de la densitat. Per defecte el gruix té el mateix valor que l'amplada, perquè assumeix que la secció és quadrada, però si un os és molt pla se li pot definir un gruix específic.
Per defecte els ossos són dinàmics, és a dir, la seva posició i angle pot ser modificat per altres objectes i forces de la simulació. Per contra, els ossos estàtics es mantenen fixes i no canvien ni de posició i ni d'angle.

Atributs:

Constants	Valor	Descripció
`color BONE_COLOR`	`color(255, 255, 240)`	Color de l'os.
`float BONE_FRICTION`	`8`	Coeficient fregament.
`float BONE_ELASTICITY`	`0,5`	Coeficient d'elasticitat.
`float BONE_DENSITY`	`1900 kg/m`^{3}	Densitat de l'os.

Propietats	Descripció
`Body body`	Box2D *Body* (objecte dinàmic de la simulació).
`String name`	Nom identificatiu de l'os.
`float posX`	Coordenada X de la posició actual.
`float posY`	Coordenada Y de la posició actual.
`float sizeLength`	Llargada de l'os.
`float sizeWidth`	Amplada de l'os.
`float sizeDepth`	Gruix de l'os.

Mètodes:

Mètode	Descripció
`void display(float displayScale)`	Visualitza l'os en la posició i angle calculat pel PBox2D.

5.2.3. La classe *Weight*

Introducció:

Aquesta classe s'utilitza per modelar pesos que s'uneixen al model musculoesquelètic a l'hora de fer experiments. És un objecte que habitualment s'uneix als ossos, i segons la seva posició servirà per aplicar resistència a uns músculs o uns altres. La seva massa pot variar per realitzar diferents experiments en un mateix esquelet.

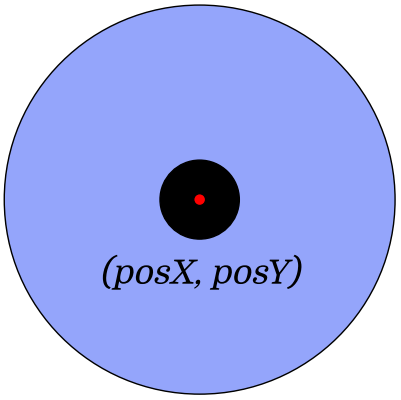


Figura 5.4 - Diagrama de l'objecte *Weight* amb les propietats de posició

El model físic el representa com un cercle amb un radi calculat a partir de la seva massa, la densitat del ferro (7874 kg/m^3 **[kg-m3. 2020]**) i suposant un gruix de 10 cm. En visualitzar-lo a la pantalla es mostra com un cercle blau amb un forat negre al centre de 2,5 cm que representa el forat on es posa a la barra **[Figura 5.4]**.

Constructor:

El constructor bàsic de l'objecte *Weight* requereix com a paràmetres la seva posició (X i Y) i la seva massa, i opcionalment el seu nom.

`Weight (`	Valor per defecte	Descripció
`String name,`	`"Unknown Weight #000"`	Nom identificatiu del pes.
`float posX,`		Coordenada X de la posició inicial.
`float posY,`		Coordenada Y de la posició inicial
`float mass)`		Massa del pes.

Atributs:

Constants	Valor	Descripció
`color WEIGHT_COLOR`	`color(#7F82BC)`	Color del pes.
`float WEIGHT_FRICTION`	`8`	Coeficient fregament.
`float WEIGHT_ELASTICITY`	`0,5`	Coeficient d'elasticitat.
`float WEIGHT_DENSITY`	`7874 kg/m`^{3}	Densitat del pes (ferro).

Propietats	Descripció
`Body body`	Box2D *Body* (objecte dinàmic de la simulació).
`String name`	Nom identificatiu del pes.
`float posX`	Coordenada X de la posició actual.
`float posY`	Coordenada Y de la posició actual.
`float sizeRadius`	Radi del pes.
`float sizeDepth`	Gruix del pes (0,05 m).
`float mass`	Massa del pes.

Mètodes:

Mètode	Descripció
`void display(float displayScale)`	Visualitza el pes en la posició i angle calculat pel PBox2D.

5.3. Les unions

5.3.1. La classe *Glue*

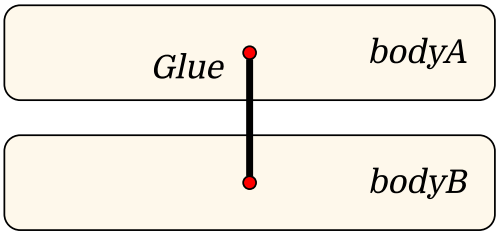


Figura 5.5 - Diagrama de la classe *Glue*

Introducció:

Aquesta classe s'utilitza per unir dos objectes del model musculoesquelètic. La Unió de dos ossos modela una articulació fibrosa que manté els ossos fixos entre ells. També serveix per fixar un os a un clau o a un pes, per exemple en unir un pes a una mà.

El model físic el representa com una *WeldJoint*. La visualització a la pantalla és opcional i dibuixa una línia entre els punts d'unió. **[Figura 5.5]**

Constructor:

El constructor bàsic de la unió *Glue* requereix com a paràmetres els dos cossos que uneix, i opcionalment el seu nom i la seva posició relativa.

`Glue (`	Valor per defecte	Descripció
`String name,`	`"Unknown Glue Joint #000"`	Nom identificatiu de la unió.
`Body bodyA,`		Cos de l'objecte A.
`Body bodyB,`		Cos de l'objecte B.
`Vec2 relativePosition)`	`Vec2(0, 0)`	Posició relativa entre els dos cossos.

Atributs:

Constants	Valor	Descripció
`color GLUE_COLOR`	`color(255, 0, 0)`	Color de la unió.

Propietats	Descripció
`WeldJoint weldJoint`	Box2D *WeldJoint* (unió rígida del PBox2D).
`String name`	Nom identificatiu de la unió.

Mètodes:

Mètode	Descripció
`void display(float displayScale)`	Visualitza la unió en la posició calculada pel PBox2D.

5.3.2. La classe *Hinge*

Introducció:

Aquesta classe serveix per poder unir dos objectes del model musculoesquelètic mitjançant una unió angular que funciona com una frontissa. Si s'utilitza per connectar dos ossos representa una articulació sinovial.

El model físic el representa com una *RevoluteJoint*. En visualitzar-lo a la pantalla es mostra com una circumferència del radi establert amb un punt de 2 píxels de diàmetre al centre per indicar l'eix de rotació. **[Figura 5.6]**

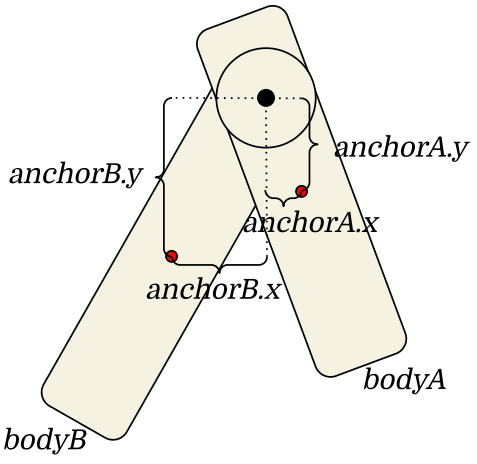


Figura 5.6 - Diagrama de la classe *Hinge* amb els punts d'unió i els cossos anomenats

Constructor:

El constructor bàsic de la unió *Hinge* requereix com a paràmetres els dos cossos que uneix, i opcionalment els punts d'unió dels dos cossos. El punt d'unió d'un cos és un vector posició que determina on l'eix estarà localitzat al cos. L'origen d'aquest vector és el centre del cos.

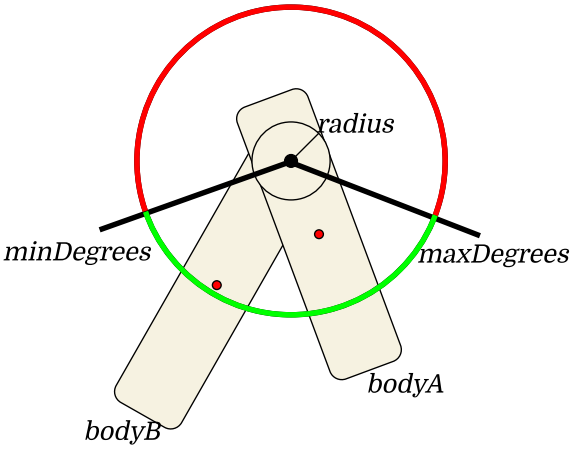


Figura 5.7 - Diagrama de la classe Hinge amb l'angle màxim i mínim anomenats

`Hinge (`	Valor per defecte	Descripció
`String name,`	`"Unknown Hinge Joint #000"`	Nom identificatiu de la unió.
`Body bodyA,`		Cos de l'objecte A.
`Body bodyB,`		Cos de l'objecte B.
`Vec2 anchorA,`	`Vec2(0, 0)`	Punt d'unió del cos A.
`Vec2 anchorB,`	`Vec2(0, 0)`	Punt d'unió del cos B.
`float radius,`		Radi de la unió.
`float minDegrees,`	`0`	Angle mínim de la unió.
`float maxDegrees)`	`0`	Angle màxim de la unió.

Opcionalment, també pots definir el nom identificatiu de la unió i l'angle relatiu mínim i màxim entre els dos cossos **[Figura 5.7]**. Si no es defineixen, els dos cossos podran girar lliurement. Finalment, la unió es visualitza com un cercle localitzat al punt d'unió, així doncs també s'ha de definir el radi d'aquest cercle.

Atributs:

Constants	Valor	Descripció
`color HINGE_COLOR`	`color(255, 255, 255)`	Color de la unió.

Propietats	Descripció
`RevoluteJoint revoluteJoint`	Box2D *RevoluteJoint* (unió rígida del PBox2D).
`String name`	Nom identificatiu de la unió.
`float radius`	Radi de la unió.
`minDegrees`	Angle mínim de la unió.
`maxDegrees`	Angle màxim de la unió.

Mètodes:

Mètode	Descripció
`void display(float displayScale)`	Visualitza la unió en la posició i angle calculat pel PBox2D.

5.4. Els músculs

5.4.1. La classe *Muscle*

Introducció:

Aquesta classe s'ha programat per modelar els músculs en el model musculoesquelètic. És l'objecte més important i complex del model musculoesquelètic, ja que ha de calcular la força que genera segons el seu nivell d'activació, utilitzant el model de Hill, i aplicar-la als objectes on està connectat. Representa un múscul estriat, perquè es controla voluntàriament.

Com que cap dels objectes bàsics existents en el PBox2D permet aproximar el funcionament d'un múscul, ha calgut programar un objecte des de zero. Per això, les instàncies de la classe *Muscle* són un objecte virtual que genera forces que s'apliquen als objectes del PBox2D però sense tenir un *body* dins el món PBox2D.

A la pantalla es visualitza com una el·lipse d'àrea constant ("isovolumètric" en 2D) que varia de color segons del seu nivell d'activació i els tendons com unes línies blanques entre els punts d'unió i el múscul. **[Figura 5.8]**

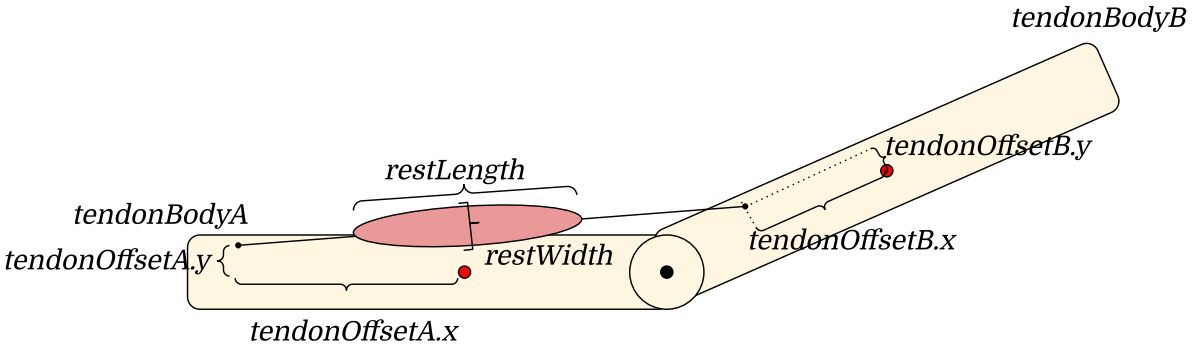


Figura 5.8 - Diagrama de l'objecte *Muscle* amb les mides i punts d'unió anomenats

Constructor:

El constructor bàsic de l'objecte *Muscle* requereix com a paràmetres la seva llargada i amplada en repòs i els dos cossos on es connecta, i opcionalment el seu nom, la llargada inicial normalitzada i els punts d'unió dels dos cossos.

`Muscle (`	Valor per defecte	Descripció
`String name,`	`"Unknown Muscle #000"`	Nom identificatiu del múscul.
`float restlength,`		Llargada en repòs del múscul.
`float restWidth,`		Amplada en repòs del múscul.
`float startNormalizedLength,`		Llargada inicial normalitzada del múscul.
`Body tendonBodyA,`		Cos on se situa l'origen del múscul.
`Body tendonBodyB,`		Cos on se situa la inserció del múscul.

`Vec2 tendonOffsetA,`	`Vec2(0, 0)`	Posició relativa d'unió del cos A.
`Vec2 tendonOffsetB)`	`Vec2(0, 0)`	Posició relativa d'unió del cos B.

Definició de politges:

Figura 5.9 - Politja al tendó de l'origen del Biceps Brachii llarg - *https://epos.myesr.org/posterimage/esr/seram2012/111271/mediagallery/421751*

Les politges són una extensió a l'origen o la inserció dels músculs que s'utilitzen en llocs on la direcció del tendó canvia, per exemple el tendó del *Biceps Brachii* llarg té una politja al seu tendó de l'origen del múscul crear per l'*Humerus* **[Figura 5.9]**.

Per simplificar el constructor del múscul, quan les politges són necessàries són definides mitjançant una funció a part. Aquesta funció *setPulleys()* requereix quatre paràmetres: dos cossos i dos punts d'unió, un per l'*origen* i l'altre per la *inserció* del múscul. Si un costat no té politja, els seus paràmetres s'han d'omplir amb '*null*'. **[Figura 5.10]**

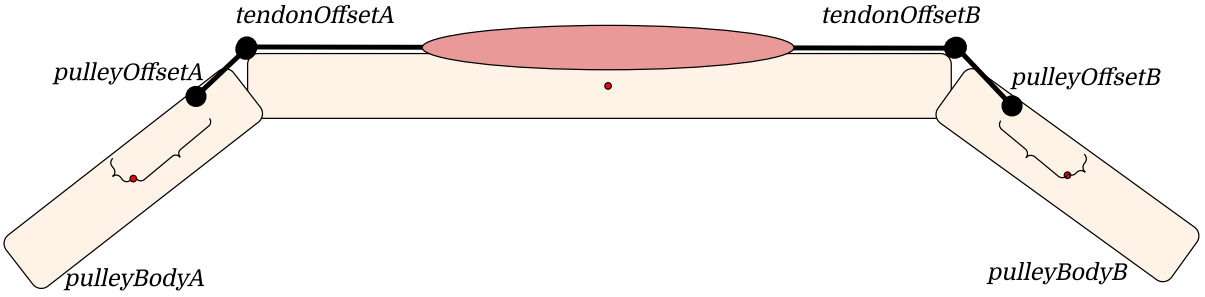


Figura 5.10 - Diagrama de l'objecte *Muscle* amb els punts d'unió de les politges anomenats

`void setPulleys (`	Valor per defecte	Descripció
`Body pulleyBodyA,`	`null`	Cos on s'uneix l'extensió de l'*origen* del múscul.

`Body pulleyBodyB`	`null`	Cos on s'uneix l'extensió de la *inserció* del múscul.
`Vec2 pulleyOffsetA`	`null`	Posició relativa del punt d'unió al cos de l'extensió A.
`Vec2 pulleyOffsetB`	`null`	Posició relativa del punt d'unió al cos de l'extensió B.

Atributs:

Constants	Valor	Descripció
`color MUSCLE_COLOR_LOW`	`color(223,177,177)`	Color del múscul quan l'activació és 0.
`color MUSCLE_COLOR_HIGH`	`color(161,44,44)`	Color del múscul quan l'activació és 1.
`color TENDON_COLOR`	`color(223,177,177)`	Color del tendó.
`float MUSCLE_REST_LENGTH`	`1`	Llargada en repòs normalitzada.
`float MUSCLE_MIN_LENGTH`	`0,5`	Llargada mínima normalitzada.
`float MUSCLE_MAX_LENGTH`	`1,5`	Llargada màxima normalitzada.
`float MUSCLE_ZERO_CONTRACTION`	`0,00001 m`	Llindar per 0 contracció.
`float MUSCLE_MAX_VELOCITY`	`0,25 m/s`	Velocitat de contracció màxima del múscul.
`float MUSCLE_SPECIFIC_TENSION`	`300000 N/m`^{2}	Esforç unitari màxim.
`float MUSCLE_FRICTION`	`8`	Coeficient fregament.
`float MUSCLE_ELASTICITY`	`0,5`	Coeficient d'elasticitat.
`float MUSCLE_DENSITY`	`1055 kg/m`^{3}	Densitat del múscul.

Aquesta classe té moltes propietats, les quals es poden dividir entre les que s'actualitzen contínuament durant la simulació, i les que es mantenen constant però

són específiques per cada múscul. Les que s'actualitzen són, per exemple les de posició, que contínuament són recalculades. Les constants, per exemple, són les mides en repòs, que són definides al constructor i es mantenen constants durant tota la simulació.

Propietats	Constant	Descripció
`String name`	Si	Nom identificatiu del múscul.
`float posX`	No	Coordenada X de la posició actual.
`float posY`	No	Coordenada Y de la posició actual.
`float angle`	No	Angle actual.
`float anglePulleyA`	No	Angle de la politja A.
`float anglePulleyB`	No	Angle de la politja B.

La variable *angle* és l'angle actual del múscul que s'actualitza segons la posició dels ossos on es connecta. Les variables *anglePulleyA* i *anglePulleyB* són l'angle actual de l'extensió dels tendons, aquestes variables també s'actualitzaran durant la simulació. **[Figura 5.11]**

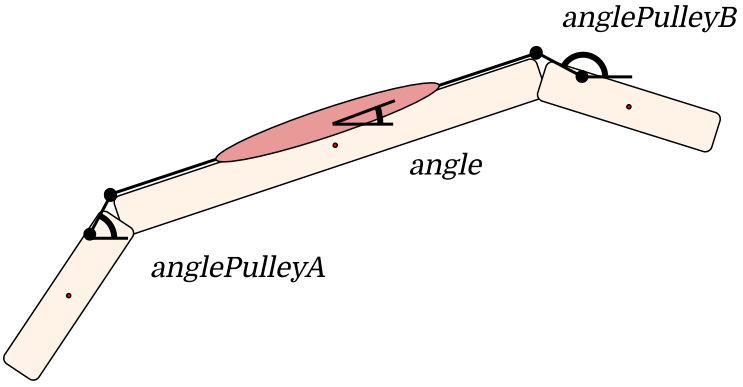


Figura 5.11 - Diagrama de l'objecte *Muscle* amb els angles dels músculs i les politges anomenats

Propietats	Constant	Descripció
`float currentWidth`	No	Amplada actual del múscul.
`float currentLength`	No	Llargada actual del múscul.
`float lastLength`	No	Llargada del múscul en el *frame* passat.
`float minLength`	Si	Llargada mínima del múscul.

`float maxLength`	Si	Llargada màxima del múscul.
`float restLength`	Si	Llargada del múscul en repòs.
`float restWidth`	Si	Amplada del múscul en repòs.
`float restDrawArea`	Si	Superfície del múscul.
`float restCrossSection`	Si	Secció del múscul en repòs.
`float restVolume`	Si	Volum del múscul en repòs.

Les variables *restLength* i *restWidth* són definides al constructor, amb el producte d'aquests dos valors s'obté el valor de la variable *restDrawArea*. Aquesta variable es mantindrà constant durant tota la simulació, ja que els músculs tenen un volum constant i en una simulació 2D tenen una superfície constant. Finalment, la variable *restCrossSection* és la secció del múscul en repòs que s'utilitza per calcular la força màxima del múscul.

Propietats	**Constant**	**Descripció**
`float muscleSymmetry`	No	La simetria en la llargada dels tendons a l'hora de dibuixar-los
`float tendonsLength`	Si	Llargada dels tendons.

Aquesta darrera variable és la llargada dels tendons. És calculada al constructor i es manté constant durant tota la simulació. Els tendons reben una deformació normal màxima de 0.049 **[Delp. 2022 (3)],** de manera que la seva elongació deguda a la tensió és negligible.

Propietats	**Constant**	**Descripció**
`float activation`	No	Activació del múscul.
`float tensionGoal`	No	Objectiu de tensió del múscul.
`float contraction`	No	Contracció del múscul.
`float normalizedContraction`	No	Contracció normalitzada del múscul.

`float contractionVelocity`	No	Velocitat de la contracció.
`float normalizedVelocity`	No	Velocitat de la contracció normalitzada.
`float forceMax`	Si	Força màxima del múscul.
`float forceActive`	No	Força activa del múscul.
`float forcePasive`	No	Força passiva del múscul.
`float forceTotal`	No	Força total del múscul.
`float forceNow`	No	Força instantània del múscul.

Quan el múscul es controla mitjançant la tensió en comptes d'amb l'activació, la variable *tensionGoal* emmagatzema la quantitat de newtons que es vol que el múscul produeixi de força activa. El valor -1 significa que el controlador de tensió està desactivat.

La variable *forceActive* proporciona quants newtons de força activa està generant el múscul, de la mateixa manera, la variable *forcePassive* proporciona quants newtons de força passiva està generant el múscul. El sumatori de les dues és el valor de la variable *forceTotal*.

Mètodes:

Mètode	**Descripció**
`void setActivation(float activation)`	Defineix l'activació del múscul.
`void setTension(float tension)`	Defineix la tensió desitjada del múscul.
`void setMuscleSymmetry(float muscleSymmetry)`	Defineix la simetria dels tendons del múscul (per defecte és 0,5)
`void display(float displayScale)`	Visualitza el múscul en la posició, angle i amb les dimensions calculades.
`void step(float timeIncrement)`	Simula el comportament del múscul durant l'increment de temps *timeIncrement*.

La funció *setActivation()* és una funció molt important que s'utilitza per poder controlar el múscul. Se li envia un valor entre 0 i 1. Biològicament, l'activació està directament correlacionat amb el senyal que envia el cervell, així doncs, l'activació és el coeficient utilitzat en la funció de Hill per calcular les forces generades.

La funció *setTension()* és l'altra manera de controlar el múscul. En comptes de controlar l'activació manualment, l'activació serà controlada automàticament mitjançant un controlador proporcional per obtenir una tensió muscular determinada.

A més, existeixen altres funcions internes que són cridades per altres parts del codi:

Mètode	**Descripció**
`Vec2 rotatePoint(Vec2 point, Vec2 center, float angle)`	Rota el primer valor al voltant del segon valor tants radiants com el tercer valor. Retorna la posició nova del primer valor.
`void setCurrentLength(float currentLength)`	Defineix la llargada actual del múscul.
`PVector updateMusclePositionBetweenAnchorPoints()`	Actualitza la posició del múscul perquè estigui centrat entre l'origen i la inserció. Retorna el vector entre l'origen i la inserció.
`void updateTendonPositions()`	Actualitza les posicions dels tendons.
`void updatePulleyPositions()`	Actualitza les posicions de les politges.

5.4.2. Funcions de Hill

Per calcular la força generada per un múscul s'utilitza el model de Hill. Com s'ha explicat en l'apartat 2.3.2, el model de Hill utilitza 3 funcions on cada una retorna un coeficient respecte a la força màxima del múscul. Per tant, era necessari obtenir aquestes 3 funcions computacionalment.

El primer problema és que no hi ha una funció matemàtica que representi les línies generades per aquestes funcions. Per aquesta raó vaig decidir obtenir la corba a partir de la interpolació de dades empíriques publicades a un article del 2018 **[Ross, et al. 2018]**.

Aquest estudi utilitza els resultats empírics d'altres dos estudis anteriors per obtenir les corbes de les funcions de Hill: velocitat-força activa, llargada-força activa i llargada-força passiva **[Figura 5.12]**.

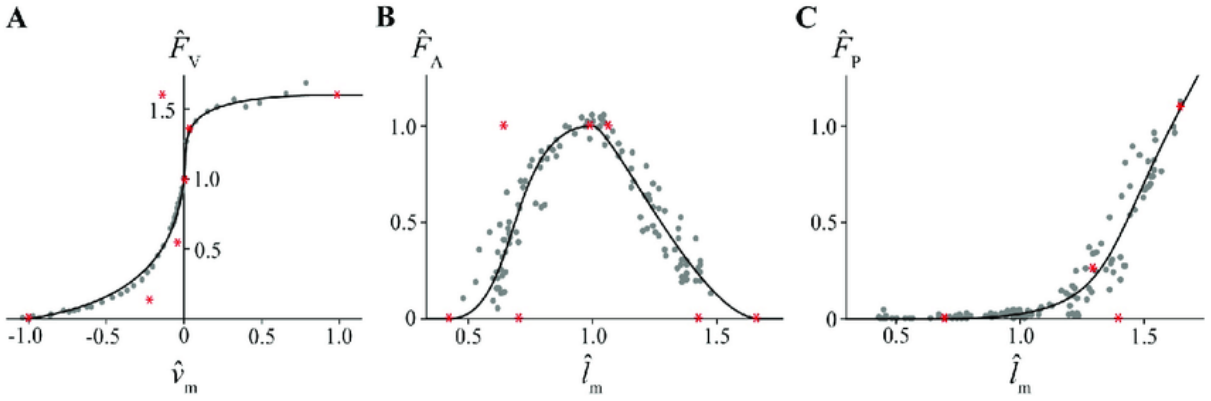


Figura 5.12 - Funcions de Hill a partir de valors empírics - *[Ross, et al. 2018]*

El seguent problema era que les dades empíriques d'aquests dos estudis no estaven disponibles públicament, més enllà de les gràfiques incloses a l'article.

La solució d'això va ser utilitzar l'eina online *WebPlotDigitizer* (https://automeris.io/WebPlotDigitizer). Aquesta eina permet obtenir les coordenades precises de qualsevol punt en una imatge referenciades a un eix de coordenades prèviament calibrats. Vaig carregar les gràfiques de l'estudi i vaig obtenir 3 taules de punts de les funcions de llargada-força activa **[Annex B.1]**, velocitat-força activa **[Annex B.2]** i llargada-força passiva **[Annex B.3]**.

Una vegada obtinguts els punts de la corba calia crear un codi que interpolés les dades. És a dir, que a partir d'un valor X obtingui el valor corresponent de l'eix Y. Per fer això, el codi ha d'interpolar el valor de l'eix Y dins del segment més proper. **[Figura 5.13]**

El primer pas és trobar els dos punts més propers. Per fer això faig servir un algoritme de cerca binària. Aquest algoritme és l'algoritme de cerca més eficient per llistes ordenades, per sort, les llistes dels punts són llistes ordenades per l'eix x.

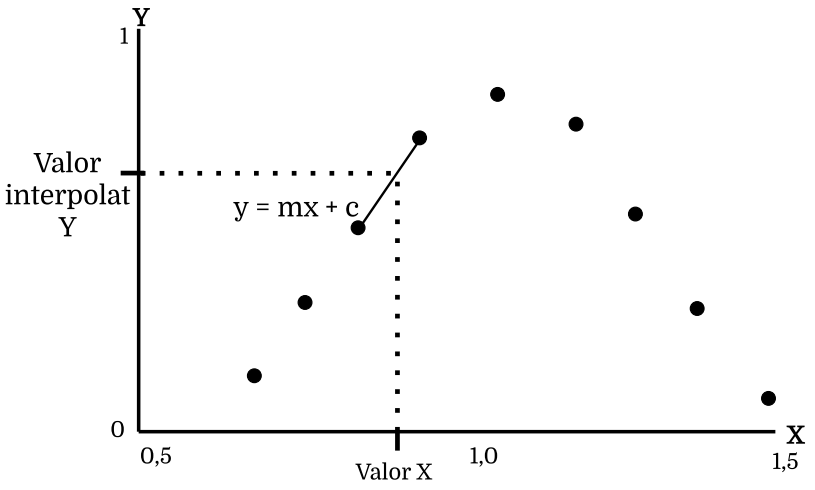


Figura 5.13 - Representació gràfica de la interpolació de punts d'una gràfica

```
int findByBinarySearch(float xValue, ArrayList<Float> xList)
{
  int topEnd = xList.size()-1;
  int bottomEnd = 0;
  int index = (topEnd+bottomEnd)/2;
  boolean foundIndex = false;

  while(!foundIndex)
  {
    index = floor((topEnd+bottomEnd)/2);
    if (xValue >= xList.get(index)
        && xValue < xList.get(index + 1))
    {foundIndex = true;}

    else
    {
      if(xValue < xList.get(index))
      {topEnd = index;}
      else
      {bottomEnd = index;}
    }
  }
  return(index);
}
```

Quan el codi troba els dos punts que rodegen el valor de l'eix x que s'ha demanat, ha d'interpolar el valor de l'eix y. Per fer això, obté la funció del segment que uneix aquests dos punts i utilitza el valor de l'eix x per obtenir el valor de l'eix y en aquest segment.

Aquest codi és genèric i serveix per a qualsevol taula, aplicat a les tres taules anteriors permet obtenir computacionalment les tres funcions de Hill. **[Figura 5.14] [Figura 5.15] [Figura 5.16]**

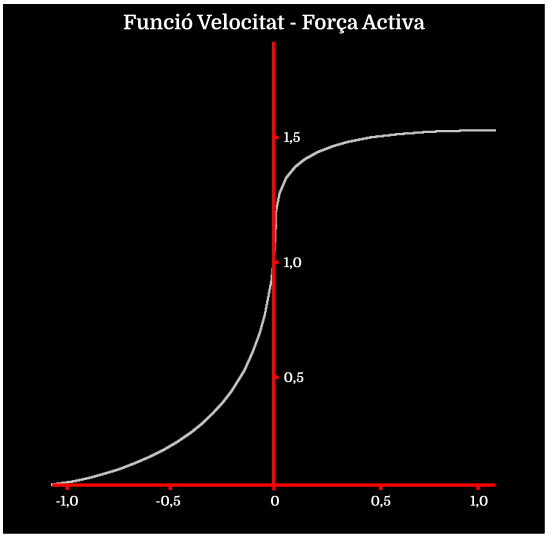


Figura 5.14 - Interpolació de la gràfica velocitat - força activa

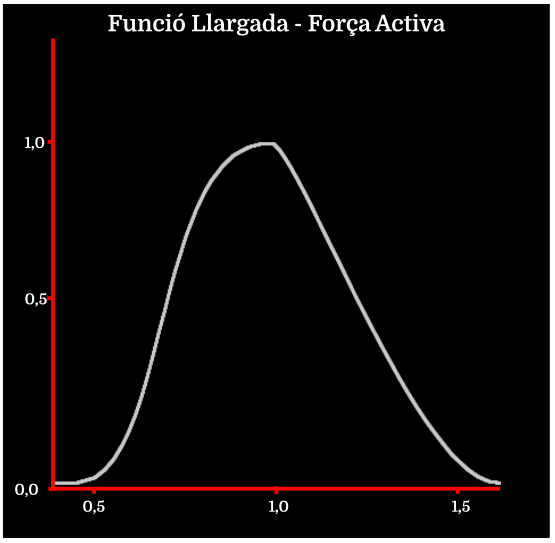


Figura 5.15 - Interpolació de la gràfica llargada - força activa

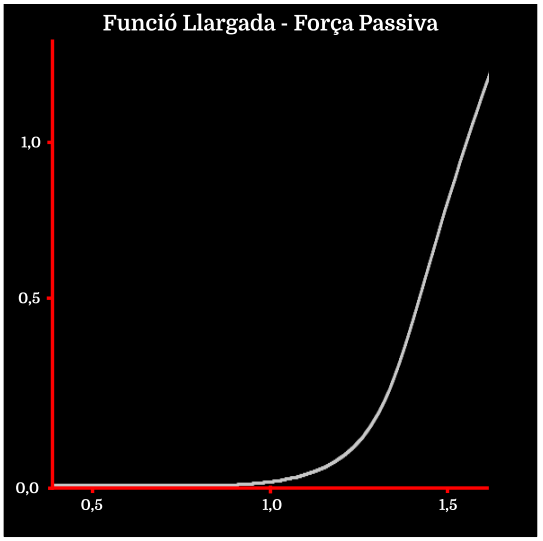


Figura 5.16 - Interpolació de la gràfica llargada - força passiva

5.4.3. Implementació del model muscular de Hill

Una vegada obtingudes les tres funcions de Hill ja és possible implementar el funcionament d'un múscul segons el model de Hill. Aquest codi és una de les parts fonamentals d'aquest projecte de recerca, ja que és el que prediu la força del múscul a partir del nivell d'activació.

Per fer-ho, per cada frame (de durada *timeIncrement*) de la simulació, es realitzen els següents càlculs per actualitzar les propietats del múscul:

- Geometria: posició, llargada i angles del múscul
- Dinàmica: velocitats, contraccions musculars
- Forces: activa i passiva generada pel múscul

I finalment:

- Aplicació de les forces als punts de connexió

5.4.3.1. Càlculs geomètrics del múscul

El primer pas és obtenir la llargada del múscul. Aquesta es calcula geomètricament mesurant la distància entre els extrems interiors dels tendons.

Per fer això, es crida a la funció *updateMusclePositionBetweenAnchorPoints()* que actualitza les posicions de les unions i extensions dels tendons a partir de les posicions dels ossos calculades pel motor físic. Quan els ha actualitzat, la funció retornarà el vector que uneix els punts d'*origen* i d'*inserció* del múscul.

La magnitud d'aquest vector és la distància entre aquests punts, que és igual a la suma de la llargada dels tendons i la del múscul. Per tant, si l'hi restes la llargada dels tendons obtens la llargada del múscul.

Aquesta llargada s'actualitza mitjançant la funció *setCurrentLength()* que també actualitza l'amplada del múscul mantenint l'àrea constant. Addicionalment, es calcula la llargada normalitzada dividint la llargada actual entre la llargada del múscul en repòs.

```
// Set muscle position and angle
PVector lineAB = updateMusclePositionBetweenAnchorPoints();

// Calculate muscle length
anchorDistance += (lineAB.mag() - anchorDistance) * 0.2;
float muscleLength = anchorDistance - tendonsLength;
setCurrentLength(muscleLength);
float normalizedSizeLength = this.currentLength
                           / this.restLength;
```

5.4.3.2. Càlculs dinàmics del múscul

El segon pas és calcular la velocitat normalitzada. Per fer-ho primer es calcula l'increment de la llargada, restant la llargada del múscul en el *frame* anterior de la llargada actual. Aleshores, es calcula la velocitat de contracció dividint l'increment de llargada per la duració d'un *frame* (*timeIncrement*). Finalment, s'obté la velocitat de contracció normalitzada dividint la velocitat de contracció per la velocitat màxima.

```
float lengthVariation = currentLength - lastLength;
if (lengthVariation<MUSCLE_ZERO_CONTRACTION)
  { lengthVariation = 0.0; }

this.contractionVelocity = lengthVariation / timeIncrement;
this.normalizedVelocity = contractionVelocity
                        / MUSCLE_MAX_VELOCITY;
```

També es calcula la contracció i la contracció normalitzada per actualitzar les propietats corresponents:

```
    this.contraction = restLength - currentLength;
    this.normalizedContraction = contraction/restLength;
```

El següent fragment de codi només s'executa quan el múscul es controla mitjançant la tensió. Per comprovar-ho, es mira si la variable *tensionGoal* és major que 0:

```
// Tension Controller (Proportional)
if(this.tensionGoal > 0)
{
  float error = (this.tensionGoal - this.forceActive)
                / this.forceMax;
  float correction = error * 0.08; // Kp = 0.08
  this.activation += correction;
  if (this.activation > 1) { this.activation = 1; }
  if (this.activation < 0) { this.activation = 0; }
}
```

Per aconseguir que el múscul tingui la tensió desitjada, l'activació s'ha de regular dinàmicament. Per fer això s'ha fet servir un sistema de realimentació negativa conegut com a control proporcional. Primer aconsegueixo l'error normalitzat entre la tensió desitjada i la tensió actual. Després calculo la correcció de l'activació multiplicant aquest error pel K_p, un valor empíric, que en aquest cas és 0,08. Ara la correcció és suma a l'activació actual, i es comprova que el nou valor està entre 0 i 1.

5.4.3.3. Càlculs de les forces generades

Finalment, ja s'han calculat tots els valors necessaris per aplicar les funcions de Hill i calcular la força total generada que és la suma de la força activa i la força passiva.

Per calcular la força activa s'utilitza l'equació de Hill:

$$F^a = F_{max}\,(a \cdot f^l{}_a(\bar{l}) \cdot f^v(\bar{v}))$$

Que dins el codi es programa així:

```
this.forceActive = this.forceMax * (this.activation
                * getActiveForceFromLength(normalizedSizeLength)
                * getForceFromVelocity(normalizedVelocity));
```

De la mateixa manera, per calcular la força passiva s'utilitza l'equació de Hill:

$$F^p = F_{max} \cdot f^l_p(\bar{l})$$

Que en el codi és així:

```
this.forcePasive = this.forceMax
            * getPassiveForceFromLength(normalizedSizeLength);
```

Com es pot veure, per obtenir les funcions de Hill s'utilitza el codi de l'apartat 5.4.2. que interpola els valors a partir de les taules empíriques.

Per acabar, se suma la força activa i la força passiva per obtenir la força total que el múscul està generant en aquest moment.

```
this.forceTotal = forceActive + forcePasive;
```

5.4.3.4. Aplicació de la força generada

Una vegada es coneix la força que genera el múscul ja es pot aplicar al model musculoesquelètic mitjançant el motor físic.

Durant la contracció del múscul s'aplica la força total (F) als punts d'unió del múscul amb els ossos, és a dir, als extrems exteriors dels tendons. Aquestes forces tenen direccions oposades i són paral·leles als tendons.

A més, en el cas que els tendons tinguin politges, s'aplica la mateixa força (F) a l'extrem exterior de l'extensió i una força de reacció (R) a la politja. Aquestes forces tenen la mateixa magnitud i sentit contrari, i totes dues paral·leles a l'extensió. **[Figura 5.17]**

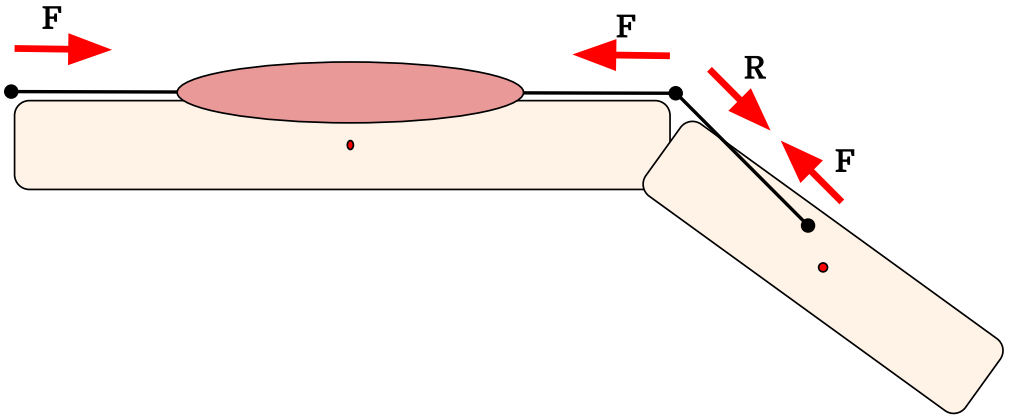


Figura 5.17 - Diagrama de forces del múscul

Per aplicar forces a la simulació s'han de definir com un vector, on la magnitud és quants newtons té i l'angle del vector és l'angle de la força, i a més s'ha d'especificar a quin cos i punt s'aplica.

```
// Apply Force A to TendonAnchor A
Vec2 forceA = rotatePoint(new Vec2(forceNow, 0), angle);
tendonBodyA.applyForce(forceA, posTendonAnchorA);

// If pulley exists apply action and reaction forces
if (pulleyBodyA!=null)
{
  // Force Applied to TendonPulley
  Vec2 actionForceA = rotatePoint(new Vec2(forceNow, 0),
                                  anglePulleyA);
  pulleyBodyA.applyForce(actionForceA, posPulleyAnchorA);

  // Reverse Force Applied to TendonAnchor
  Vec2 reactionForceA = rotatePoint(new Vec2(forceNow, 0),
                                    anglePulleyA+PI);
  tendonBodyA.applyForce(reactionForceA, posTendonAnchorA);
}
```

Com mostra el codi anterior, la primera força que s'aplica és la força del tendó A. Es crea un vector de magnitud igual a la força total i es rota segons la variable *angle* (orientació del múscul). Després s'aplica al punt *posTendonAnchorA* utilitzant la funció *applyForce()* per introduir-la al motor físic.

Després, comprova si el costat A del múscul té una extensió. Si no té extensió no farà res més però si en té, aplicarà dues forces addicionals de la mateixa magnitud. La primera s'aplica al punt d'unió extern de l'extensió (*posPulleyAnchorA*) amb un angle paral·lel a l'extensió (*anglePulleyA*). I la segona, és la força de reacció, i per tant amb sentit contrari (*anglePulleyA+PI*), que s'aplica al punt d'unió intern de l'extensió (posTendonAnchorA). **[Figura 5.10]**

Quan ha aplicat les forcés pel cantó A, repetirà exactament el mateix però pel cantó B:

```
// Apply Force B to TendonAnchor B
Vec2 forceB = rotatePoint(new Vec2(forceNow, 0), angle+PI);
tendonBodyB.applyForce(forceB, posTendonAnchorB);

// If pulley exists apply action and reaction forces
if (pulleyBodyB!=null) {
  // Force Applied to TendonPulley
  Vec2 actionForceB = rotatePoint(new Vec2(forceNow, 0),
                                  anglePulleyB);
  pulleyBodyB.applyForce(actionForceB, posPulleyAnchorB);

  // Reverse Force Applied to TendonAnchor
  Vec2 reactionForceB = rotatePoint(new Vec2(forceNow, 0),
                                    anglePulleyB+PI);
  tendonBodyB.applyForce(reactionForceB, posTendonAnchorB);
}
```

5.5. La classe *MuscularModel*

Introducció:

Amb els objectes anteriors n'hi ha prou per poder crear qualsevol model musculoesquelètic en 2D. Un programador que vulgui crear un model hauria de crear els diferents objectes i unions, i encarregar-se de cridar les funcions que actualitzen i visualitzen els diferents elements de la simulació per cada cicle de la simulació.

Si el programador volgués crear més models veuria que comparteixen molt del seu codi. Per aquesta raó, s'ha programat una classe abstracta *MuscularModel* que generalitza qualsevol model muscular com unes llistes d'ossos, de músculs, de pesos i d'unions. Aquesta classe s'encarrega d'actualitzar i visualitzar el model en cada cicle de la simulació.

A més a més, ofereix funcions que creen grups de músculs. Els quals permeten controlar molts músculs amb una sola ordre. Per exemple, la funció *setMuscleGroupActivation()* permet definir alhora l'activació de tots els músculs d'un grup muscular.

Aquesta classe abstracta permet crear qualsevol model mitjançant la creació d'una subclasse, que estén la classe mare *MuscularModel*, i omplint les cinc classes virtuals que serveixen per definir els objectes del model.

Addicionalment, s'ha creat funcions que faciliten la creació de variants d'un mateix model. I també, unes funcions que permeten seleccionar pesos diferents per la simulació. Les dues coses són útils perquè sovint en un entorn experimental s'han de comparar petites variacions d'un mateix model.

Constructor:

`MuscularModel (`	Valor per defecte	Descripció
`PApplet p,`		Pantalla de la llibreria G4P on se simula el model.

`float x,`		Coordenada X del model muscular.
`float y,`		Coordenada Y del model muscular.
`float scale`	`100`	Escala visual per defecte de la simulació.
`int fps`	`60`	*Frames* per segon de la simulació.
`int level`	`SIMULATION_LEVEL_WEIGHTS`	Índex del nivell de la simulació.
`int variantIndex`	`0`	Índex de la variant de la simulació.
`int weightIndex`	`0`	Índex del pes de la simulació.

Els primers paràmetres del constructor són els valors X i Y del punt de referència utilitzat per dibuixar el model musculoesquelètic. El paràmetre *scale* permet ampliar o reduir l'escala de la visualització, i el paràmetre *fps* el nombre d'actualitzacions per segon de la simulació.

El paràmetre *level* controla el nivell de la simulació. Hi ha 5 nivells i cada un determina quins tipus d'objecte del model se simularan. El primer nivell és el 0 on no se simularà res, i l'últim és el nivell 4 on se simularan tots els objectes del model.

El següent paràmetre, *variantIndex*, determina quina variant del model se simularà. Diferents variants d'un mateix model comparteixen la majoria d'objectes però tenen petites diferències com quants músculs té o petits canvis a la mida dels ossos... Les variants s'han de definir manualment pel programador durant la creació del model.

Finalment, el paràmetre *weightIndex* determina la massa dels pesos utilitzats. Un mateix model i variant pot tenir un pes diferent per experimentar el seu efecte en un model. Igual que les variants, els pesos s'han de definir manualment pel programador durant la creació del model.

Atributs:

Constants	Valor	Descripció
`int SIMULATION_LEVEL_NONE`	`0`	Índex del nivell de simulació amb res.
`int SIMULATION_LEVEL_BONES`	`1`	Índex del nivell de simulació amb ossos.
`int SIMULATION_LEVEL_JOINTS`	`2`	Índex del nivell de simulació amb ossos i articulacions.
`int SIMULATION_LEVEL_MUSCLES`	`3`	Índex del nivell de simulació amb ossos, articulacions i músculs.
`int SIMULATION_LEVEL_WEIGHTS`	`4`	Índex del nivell de simulació amb ossos, articulacions, músculs i pesos.

Els següents *ArrayList* emmagatzemant les llistes d'objectes que formen el model musculoesquelètic:

Llistes	Descripció
`ArrayList<Nail> allNails`	Tots els objectes *Nail* de la simulació.
`ArrayList<Bone> allBones`	Tots els objectes *Bone* de la simulació.
`ArrayList<Muscle> allMuscles`	Tots els objectes *Muscle* de la simulació.
`ArrayList<Hinge> allHinges`	Tots els objectes *Hinge* de la simulació.
`ArrayList<Glue> allGlues`	Tots els objectes *Glue* de la simulació.
`ArrayList<Weight> allWeights`	Tots els objectes *Weight* de la simulació.
`ArrayList<int[]> allMuscleGroups`	Tots els grups de músculs de la simulació.
`ArrayList<String> variantNames`	Els noms de totes les variants de la simulació.
`ArrayList<String> weightNames`	Els noms de tots els pesos de la simulació.

Propietats	Descripció
`int currentSimulationLevel`	Índex del nivell de la simulació.
`int currentModelVariant`	Índex de la variant de la simulació.
`boolean isSimulationRunning`	*true* quan la simulació estigui en marxa, *false* quan la simulació estigui parada.
`float displayScale`	Escala actual de la visualització de la simulació.
`int FRAME_RATE`	*Frames* per segon de la simulació.

Mètodes Virtuals:

Una classe abstracta té funcions que només ha declarat però no ha omplert, aquestes funcions es diuen funcions virtuals. Per tant, quan un programador crea una subclasse ha d'omplir les funcions virtuals amb el contingut corresponent. En aquest cas s'han d'omplir amb el codi que crea els diferents objectes del model.

La majoria d'aquestes funcions virtuals reben els valors x i y. Aquests valors corresponen al punt d'origen del model, ja que tots els elements del model estan posicionats relatius a aquest punt.

Finalment, també reben l'índex de la variant de la simulació. La funció s'encarregarà de generar el model correcte segons la variant obtinguda creant certs objectes o no.

Mètode Virtual	Paràmetres	Descripció
`abstract void defineMain()`		Omple les llistes de *variantNames* i *weightNames* amb els objectes correctes.
`abstract void defineBones()`	`float x, float y, int variantIndex`	Crea els ossos i els afegeix a la llista *allBones*.
`abstract void defineJoints()`	`float x, float y, int variantIndex`	Crea les articulacions i les afegeix a les llistes *allNails, allHinges o allGlues*.

`abstract void defineMuscles()`	`float x, float y, int variantIndex`	Crea els músculs i els afegeix a la llista *allMuscles*, també afegeix grups de músculs a la llista *allMuscleGroups*.
`abstract void defineWeights()`	`float x, float y, int variantIndex, int weightIndex`	Crea els pesos i els afegeix a la llista *allWeights*.

Mètodes:

Aquestes funcions serveixen per controlar el bucle del motor físic. El motor físic pot estar engegat o parat.

Mètode	Descripció
`void playSimulation()`	Engegar la simulació.
`void stopSimulation()`	Parar la simulació.
`void togglePauseSimulation()`	Alternar l'estat de la simulació.

També hi ha la funció *setGravity()* que serveix per definir la gravetat de la simulació com un vector. Si no es defineix la gravetat serà la del planeta Terra.

Mètode	Descripció
`void setGravity(Vec2 gravity)`	Defineix la gravetat de la simulació amb un vector. Per exemple la gravetat de la Terra és (0, -9,8).

El programador pot controlar la contracció del múscul modificant la seva activació. Per modificar l'activació pot utilitzar qualsevol de les següents funcions, una determina quin múscul es vol modificar a partir del seu índex numèric i l'altre a partir del seu nom identificatiu.

Mètode	Descripció
`void setMuscleActivation(int muscleIndex, float activationLevel)`	Defineix l'activació d'un múscul a partir del seu índex en la llista *allMuscles*.
`void setMuscleActivation(String muscleName, float activationLevel)`	Defineix l'activació d'un múscul a partir del seu nom identificatiu.

Una altra manera de controlar el múscul és definint la tensió desitjada del múscul. Es pot modificar amb dues funcions, una que determina el múscul amb el seu índex i una altra que utilitza el seu nom identificatiu.

Mètode	Descripció
`void setMuscleTension(int muscleIndex, float tension)`	Defineix la tensió desitjada d'un múscul a partir del seu índex en la llista *allMuscles*.
`void setMuscleTension(String muscleName, float tension)`	Defineix la tensió desitjada d'un múscul a partir del seu nom identificatiu.

Totes aquestes funcions permeten utilitzar grups musculars. Els grups musculars es defineixen cridant a la funció *createMuscleGroup()* amb una llista de músculs. Un cop el grup està creat pots controlar tots els músculs del grup amb una sola ordre. Per exemple es pot agrupar els tres tríceps en un sol grup i activar-los alhora.

Mètode	Descripció
`int createMuscleGroup(String[] muscleNames)`	Crea un grup muscular a partir d'una llista dels noms dels músculs
`String getMuscleGroupName(int groupIndex)`	Obté el nom del grup muscular a partir del seu índex a la llista *allMuscleGroups*.
`void setMuscleGroupActivation(int groupIndex, float activationLevel)`	Modifica l'activació de tots els músculs d'un grup.
`float getMuscleGroupActivation(int groupIndex)`	Obté l'activació mitjana de tots els músculs d'un grup.

Per poder llegir o modificar les propietats dels objectes de la simulació a vegades cal accedir a un objecte concret. Les següents funcions permeten obtenir un objecte o unió a partir del seu nom identificatiu:

Mètode	Descripció
`Nail getNailByName(String name)`	Obtenir un objecte *Nail* a partir del seu nom identificatiu.
`Bone getBoneByName(String name)`	Obtenir un objecte *Bone* a partir del seu nom identificatiu.
`Hinge getHingeByName(String name)`	Obtenir un objecte *Hinge* a partir del seu nom identificatiu.
`Glue getGlueByName(String name)`	Obtenir un objecte *Glue* a partir del seu nom identificatiu.
`Weight getWeightByName(String name)`	Obtenir un objecte *Weight* a partir del seu nom identificatiu.
`Muscle getMuscleByName(String name)`	Obtenir un objecte *Muscle* a partir del seu nom identificatiu.
`int getMuscleIndexByName(String name)`	Obtenir un objecte *Muscle* a partir del seu índex a la llista *allMuscles*.

Finalment, la classe *MuscularModel* ofereix funcions per controlar la visualització de la simulació. No afecten el motor físic però serveixen per modificar l'aspecte de la simulació.

Mètode	Descripció
`void setGridProperties(float divisionLength, int numberOfQuadrantDivisions)`	Defineix les propietats de la matriu. El primer valor és la separació entre les divisions i el segon és la quantitat de divisions en la matriu.
`void resetDisplayScale()`	Redimensiona la simulació a l'escala visual inicial.
`void incrementDisplayScale(float increment)`	Incrementa l'escala visual de la simulació dinàmicament.

5.6. Exemple d'un model mínim

5.6.1 Introducció

Per entendre millor tot el que s'ha explicat, en aquesta secció es mostra el codi necessari per programar un petit model musculoesquelètic utilitzant el *framework* creat en aquest treball.

Com s'ha explicat, a l'hora de crear un model només s'ha d'omplir la plantilla de la subclasse que s'ha de crear. Primer es declaren tots els objectes necessaris per a la simulació. Després es programa la funció *defineMain()* creant les variants i pesos diferents. I finalment es declaren els objectes dins la seva funció corresponent.

```
class SampleModel extends MuscularModel
{
  // Global objects of the model
  ...
  // CONSTRUCTOR ================================================
  SampleModel(PApplet p, float x, float y,
              float scale, int fps,
              int simulationLevel)
  {
    super(p, x, y, scale, fps, simulationLevel);
  }
  SampleModel(PApplet p, float x, float y,
              float scale, int fps,
              int simulationLevel, int variantIndex,
              int weightIndex)
  {
    super(p, x, y, scale, fps, simulationLevel, variantIndex, weightIndex);
  }
  void defineMain()
  {
   ...
  }
  void defineBones(float x, float y, int variantIndex)
  {
   ...
  }
```

```
  void defineJoints(float x, float y, int variantIndex)
  {
   ...
  }
  void defineMuscles(float x, float y, int variantIndex)
  {
   ...
  }
  void defineWeights(float x, float y, int variantIndex, int weightIndex)
  {
   ...
  }
} // End of Class
```

5.6.2 Descripció del sistema d'exemple

Per mostrar com funciona aquest framework he volgut crear un model musculoesquelètic molt simple que contingui un objecte de cada tipus i serveixi d'exemple.

A partir de la plantilla de codi de l'apartat anterior es pot crear qualsevol model musculoesquelètic. Però abans d'explicar com programar el model cal definir el sistema que es vol simular.

En aquest exemple hi ha un os (20 cm x 5 cm) que penja d'una articulació sinovial, és a dir, que pot rotar. Aquest os està unit a un altre os més petit (5 cm x 5 cm) mitjançant un múscul (30 cm x 5 cm). I perquè aquest sistema musculoesquelètic no caigui a causa de la gravetat s'ha fixat l'articulació inicial mitjançant un clau. A més, per causar una oscil·lació l'os inferior s'ha desplaçat 20 cm a la dreta. **[Figura 5.18]**

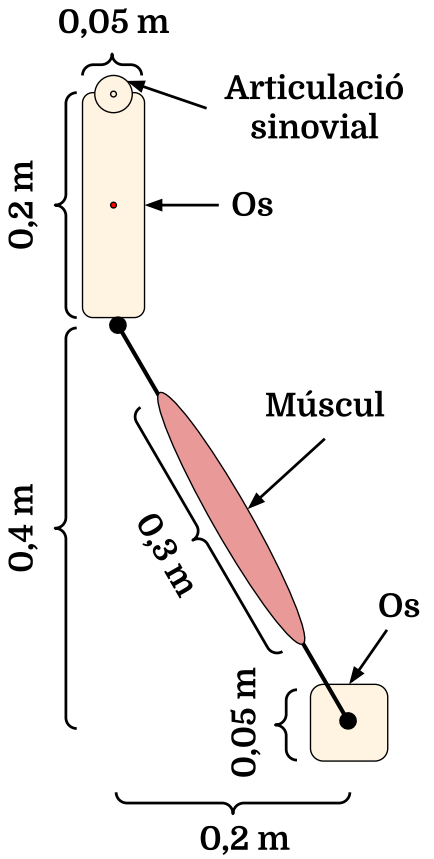


Figura 5.18 - Diagrama del sistema musculoesquelètic del model mínim

Per models de sistemes anatòmics reals s'ha de fer recerca sobre les mides i posicions dels ossos i músculs del sistema. Però, en aquest exemple fictici les mides i posicions s'han inventat arbitràriament.

5.6.3 Programació del model d'exemple

El primer pas és declarar tots els objectes que formen part de la simulació. Aquest model té:

- 1 objecte *Nail* per fixar el model al món;
- 2 objectes *Bone* que fan d'os;
- 1 objecte *Hinge* que fa d'articulació sinovial;
- 1 objecte *Muscle* que fa de múscul;

I a més:

- 1 objecte *Weight* que fa de pes;
- 1 objecte *Glue* per enganxar el pes a l'os inferior.

```
// Global objects of the model
Nail sampleNail;
Bone sampleBone1, sampleBone2;
Hinge sampleHinge;
Muscle sampleMuscle;
Weight sampleWeight;
Glue sampleGlue;
```

Una vegada declarats tots els objectes es pot començar a omplir les funcions virtuals.

La primera és la funció *defineMain()* que s'utilitza per definir les variants i els pesos. Aquest model només tindrà una variant, doncs n'afegeixo una amb el nom "Default". I de pesos en tindrà cinc: un de 0 kg, 1 kg, 5 kg, 10 kg i 50 kg.

```
void defineMain()
{
  variantNames.add("Default");
  weightNames.add("0 kg");
  weightNames.add("1 kg");
  weightNames.add("5 kg");
  weightNames.add("10 kg");
  weightNames.add("50 kg");
}
```

El següent pas és definir els ossos dins la funció *defineBones()*. Utilitzant les mides i posicions del diagrama vaig inicialitzar els ossos amb les seves propietats correctes. Després de definir els ossos s'afegeixen a la llista *allBones* perquè la classe *MuscularModel* els tingui en compte.

```
void defineBones(float x, float y, int variantIndex)
{
  sampleBone1 = new Bone("Bone1", x+0, y-0.1, 0.05, 0.2);
  allBones.add(sampleBone1);
  sampleBone2 = new Bone("Bone2", x+0.2, y-0.6, 0.05, 0.05);
  allBones.add(sampleBone2);
}
```

El proper pas és definir les unions amb la funció *defineJoints()*. La posició de l'objecte *Nail* és l'origen del model, i quan està definida s'afegeix a la llista *allNails*. La unió *Hinge* uneix el *Nail* amb l'extrem superior del primer os, i quan està creada s'afegeix a la llista *allHinges*.

```
void defineJoints(float x, float y, int variantIndex)
{
  // FIXED JOINT
  sampleNail = new Nail("Nail", x, y);
  allNails.add(sampleNail);

  // HINGE JOINT
  sampleHinge = new Hinge("Hinge", sampleNail.body,
                          sampleBone1.body, new Vec2(0,0),
                          new Vec2(0, 0.1), 0.025, 0, 0);
```

```
  allHinges.add(sampleHinge);
}
```

Per definir els músculs es fa dins la funció *defineMuscles()*. Mirant al diagrama es pot veure que el múscul té una llargada de 30 cm i una amplada de 5 cm i s'uneix a l'extrem inferior del primer os i al centre del segon. Quan s'ha definit s'afegeix a la llista *allMuscles*.

```
void defineMuscles(float x, float y, int variantIndex)
{
  // MUSCLE
  sampleMuscle = new Muscle("Muscle", 0.3, 0.05,
                            Muscle.MUSCLE_REST_LENGTH,
                            sampleBone1.body, sampleBone2.body,
                            new Vec2(0, -0.1), new Vec2(0, 0));
  allMuscles.add(sampleMuscle);
}
```

Finalment, es defineixen els pesos dins la funció *defineWeights()*. Aquesta funció rep la variable *weightIndex* que determina com serà el pes. Això es pot fer de moltes maneres però jo ho vaig fer amb una llista. Aquesta llista conté les masses dels pesos en el mateix ordre que s'han afegit en la funció *defineMain()*.

Quan s'ha creat la llista es pot crear el pes, utilitzant la variable weightIndex per saber l'índex de la massa que té pes. Després s'afegeix a la llista *allWeights()*.

Ara es crea la unió de tipus *Glue* que uneix el pes amb un os inferior. Com que els pesos són opcionals aquesta unió també ha de ser opcional. Per això, es defineix dins aquesta funció i no la funció *defineJoints()*.

Finalment, es crea la unió indicant quins dos cossos uneix (el pes i el segon os) i s'afegeix a la llista *allGlues*.

```
void defineWeights(float x, float y, int variantIndex,
                   int weightIndex)
{
```

```
  float[] variantWeights = {0.0, 1.0, 5.0, 10.0, 50.0};

  // WEIGHT
  sampleWeight = new Weight("Weight", x+0.2, y-0.6,
                            variantWeights[weightIndex]);
  allWeights.add(sampleWeight);

  //WEIGHT-RADIUS
  sampleGlue = new Glue(sampleBone2.body, sampleWeight.body);
  allGlues.add(sampleGlue);
}
```

5.6.4 Utilització del model d'exemple

Ara el model computacional està creat, només cal programar el codi que inicialitza i controla la simulació.

Primer, s'ha de declarar una instància de la classe *MuscularModel*, aquesta instància serà on s'emmagatzema el model.

```
MuscularModel model;
```

Dins la funció del Processing *setup()* pots definir els paràmetres de la finestra del Processing que vulguis, en aquest exemple només he definit la mida de la finestra.

```
void setup()
{
  size(500,500);
```

Després cal definir la variable *box2D* i crear el món. És important que aquestes dues línies de codi estiguin en la funció *setup()*, sense elles el motor físic no serà creat i la simulació no funcionarà.

```
  //Box2D setup
  box2d = new Box2DProcessing(this);
  box2d.createWorld();
```

Ara, es pot inicialitzar la instància de *MuscularModel* que s'ha declarat prèviament. Per fer això és crida al constructor de la subclasse *SampleModel* creada en la secció 5.6.. També es pot cridar a la funció *setGridProperties()* i definir les propietats de la matriu.

```
  model = new SampleModel(this, 0.0, 0.3, 50, 60,
                          MuscularModel.SIMULATION_LEVEL_WEIGHTS,
                          0, 1);
  model.setGridProperties(0.05, 10);
}
```

Finalment, dins la funció *draw()* s'ha de cridar les funcions *step()* del model perquè calculi el següent pas de la simulació i després a la funció *display()* perquè ho visualitzi. És important 'esborrar' la pantalla, amb la funció *background()*, al principi de la funció *draw()* perquè la simulació es dibuixi sobre una pantalla buida.

```
void draw()
{
  background(0);
  model.step();
  model.display();
}
```

Tot el codi del *framework* es pot consultar en l'annex C o descarregar-lo al GitHub sota la llicència de codi obert, *GNU General Public License v3.0*.

6. Demostració i resultats

6.1. Introducció

El *framework* presentat al capítol 5 és una eina potent i molt flexible, però en ser una llibreria especialitzada, està orientada a programadors. De manera que per algú sense coneixements de programació no és fàcil fer-se una idea del potencial i utilitat que permet el framework.

Per aquesta raó vaig crear un programa que serveixi com a demostració per la simulació de models musculoesquelètics. Aquest programa utilitza el *framework* però addicionalment inclou una interfície gràfica per interactuar més intuïtivament amb la simulació.

El programa de demostració inclou tres models musculoesquelètics: el model mínim d'exemple del punt 5.6. del capítol anterior, i dos models més complexos que representen l'anatomia real d'un braç i d'una cama. Aquest programa també inclou una interfície gràfica per controlar la simulació (utilitzant botons, llistes delegables i controls lliscants) i interactuar amb els músculs ajustant els seus nivells d'activació. D'aquesta manera utilitzar la simulació és molt fàcil i intuïtiu. Finalment, el programa inclou un sistema de visualització gràfica en temps real que facilita observar els senyals interns del múscul, i que permet l'exportació d'aquestes dades.

Finalment, utilitzant aquesta eina s'han realitzat una sèrie de demostracions i experiments per avaluar la qualitat de la simulació.

6.2. Exemples de models complexos

6.2.1. La subclasse *ArmModel*

El primer model anatòmicament realista que inclou la demostració és la simulació de la part superior d'un braç. A continuació es presenta l'estructura del model i com s'ha programat utilitzant el *framework*.

6.2.1.1. Descripció del sistema *ArmModel*

Com que aquest model representa un sistema musculoesquelètic real, perquè la seva física sigui realista, les mides i posicions han de ser les mateixes del sistema real. Per això vaig haver de realitzar molta recerca per obtenir un diagrama de com és un braç mecànicament.

El primer pas era obtenir les mides dels ossos del braç. A partir de diferents estudis i, en el cas de la mà, de mesures empíriques, es va poder obtenir aquesta taula **[Taula 6.1]**:

Os	**Llargada**	**Amplada**
Húmer	0,304 m **[Khan. 2020]**	0,048 m **[Milner. 2011]**
Radi	0,229 m **[Hong. 2021]**	0,024 m
Cúbit	0,245 m **[Hong. 2021]**	0,019 m
Palmell	0,08 m	0,06 m
Dit	0,07 m	0,01 m

Taula 6.1 - Llargada i amplada dels ossos de l'*ArmModel*

Cal tenir en compte que, en el model, els ossos de la ma han estat simplificats per millorar l'aparença. Com la ma no forma part del model muscular que es vol simular, s'han unit els ossos del palmell en un únic bloc al qual s'han unit als ossos dels dits. Obtenir les mides dels ossos va ser relativament fàcil, però aconseguir mides de músculs va resultar molt més complicat. Hi ha moltes raons per això, una era que hi ha molts menys estudis que mesurin la llargada d'un múscul que la d'un os. També és molt més difícil mesurar la llargada d'un múscul perquè és difícil distingir el múscul del tendó, per això, molts estudis utilitzen la llargada del múscul més la del tendó sense especificar-ho. Això es pot veure en un estudi **[Podgórski. 2019]** que va mesurar la llargada del bíceps brachii com 31 cm, cosa que només és possible si també es mesura la llargada dels tendons.

I, una altra raó és perquè com que la llargada dels músculs es mesura en cadàvers, aquests músculs poden estar passant pels processos *rigor mortis* o *flaccidiat secundària* **[Shrestha. 2019]** que varien la seva llargada.

Una vegada obtingudes les mides dels ossos i músculs del model cal saber les seves posicions relatives. A partir de diagrames anatòmics del braç es va poder definir aquest diagrama del braç **[Figura 6.1]**:

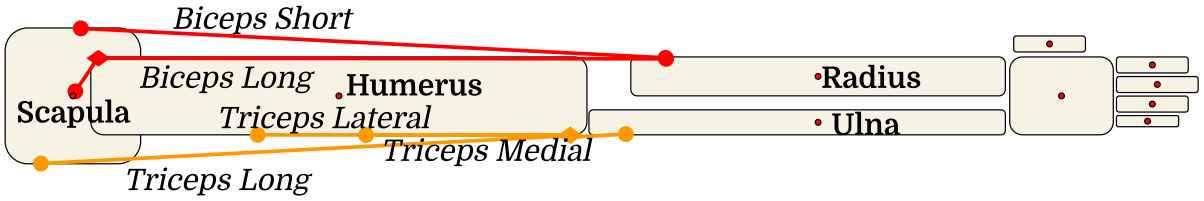


Figura 6.1 - Diagrama anatòmic d'un braç amb els ossos i músculs anomenats

6.2.1.2. Programació de la subclasse *ArmModel*

Com s'ha explicat en el punt 5.5., per programar un model musculoesquelètic només cal omplir cinc mètodes virtuals, que defineixen els diferents elements que el componen.

El primer mètode és el *defineMain()*, en aquest vaig definir les variants **[Taula 6.2]** i els pesos **[Taula 6.3]**.

Índex	Variants
0	*All Muscles*
1	*Biceps Short*
2	*All Biceps*
3	*All Triceps*
4	*All Biceps + Triceps*

Taula 6.2 - Variants del model *ArmModel*

Índex	Músculs
0	*0 kg*
1	*1 kg*
2	*5 kg*
3	*10 kg*
4	*50 kg*
5	*100 kg*

Taula 6.3 - Pesos del model *ArmModel*

L'última variant, a la pràctica, és idèntica a la primera. Però, si algun dia algú afegeix altres músculs del braç igualment haurà d'haver-hi una variant de només bíceps i tríceps, ja que són músculs complementaris.

El segon mètode és el *defineBones()*. Per definir les mides dels ossos vaig fer servir la taula 6.1, i per definir les posicions entre ells vaig utilitzar el diagrama del punt anterior **[Figura 6.1]**.

El següent mètode és el *defineJoints()*. Aquí vaig definir les unions entre els ossos. L'escàpula es fixa al món unint-se a un objecte *Nail* amb una unió de tipus *Glue*. Per unir l'escàpula i l'húmer vaig fer servir una unió angular (tipus *Hinge*) amb uns límits d'angle iguals que en una espatlla real. El cúbit i l'húmer també es van unir amb una altra unió angular amb els límits d'angle iguals que un colze real. La resta d'ossos es van unir al cúbit amb una unió de tipus *Glue*. No vaig posar una unió angular entre el palmell i el cúbit, ja que el seu moviment és irrellevant pels músculs creats i afegiria complexitat computacional innecessària al model.

En el mètode *defineMuscles()* vaig crear els músculs. Per definir les posicions dels tendons dels músculs i les extensions vaig fer servir aquest diagrama **[Figura 6.2]**.

També vaig fer que segons la variant es creïn diferents músculs, segons s'ha definit en el mètode *defineMain()*.

Finalment, en el mètode *defineWeights()* vaig fer un codi que crea un objecte de tipus *Weight* amb la massa definida indirectament per la variable *weightIndex*. Aquest pes s'uneix a la ma amb una unió de tipus *Glue*.

Ara el model musculoesquelètic *ArmModel* ja està definit i es pot simular. Com es pot veure en la figura 6.3 el model és òbviament el d'un braç. Els bíceps doblegaven el braç i els tríceps l'estenen. El codi d'aquest model es pot consultar en l'annex **[Annex D.6]**.

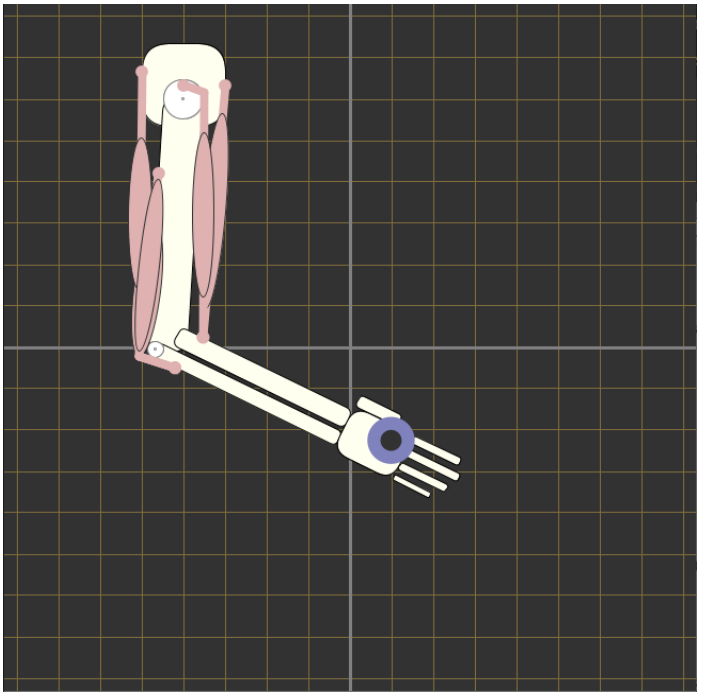


Figura 6.3 - Captura de pantalla de la simulació del model *ArmModel*

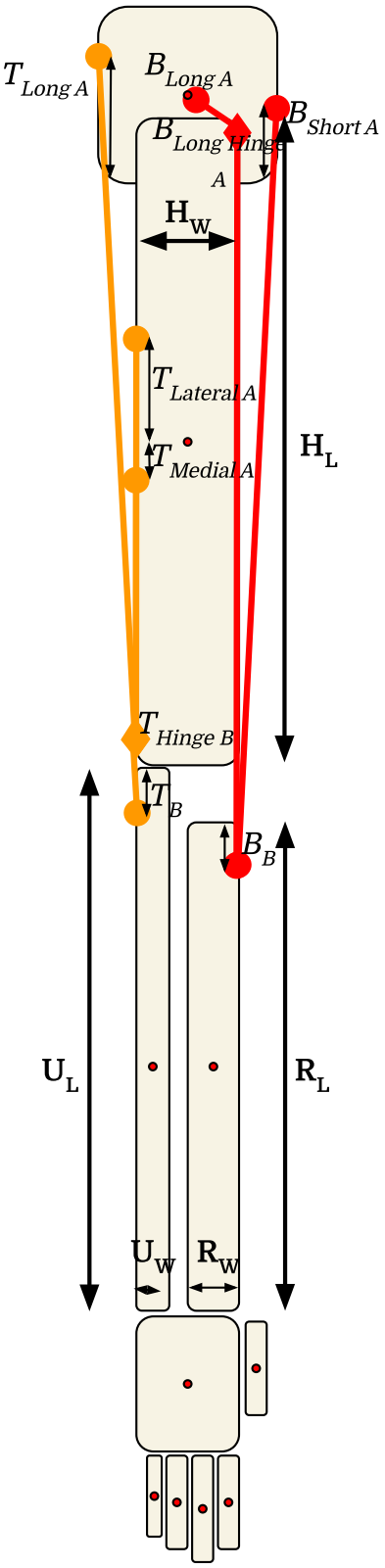


Figura 6.2 - Diagrama del model del braç amb les mides relatives anomenades

6.2.2. La subclasse *LegModel*

El segon model anatòmic que inclou la demostració és la simulació de la part inferior d'una cama. En els dos punts següents es presenta l'estructura del model i com s'ha utilitzat el *framework* per programar-ho.

6.2.2.1. Descripció del sistema *LegModel*

Igual que el sistema *ArmModel*, el sistema *LegModel* és una representació d'un sistema musculoesquelètic real. Això vol dir que les mides i posicions han de ser com més correctes possibles.

Per obtenir les mides dels ossos **[Taula 6.4]** vaig utilitzar articles científics quan va ser possible i mesures empíriques en cas contrari.

Os	**Llargada**	**Amplada**
Pelvis	0,17 m	0,22 m **[Giroux. 2008]**
Fèmur	0,48 m **[OpenOregon. 2023]**	0,0537 m **[Terzidis, 2012]** **[OpenOregon. 2023]**
Peroné	0,35 m **[Ide. 2015]**	0,0175 m **[Lubek. 2017]**
Tíbia	0,36 m **[Hrdlicka. 1898]**	0,0258 m **[Karimi. 2019]**
Peu	0,17 m	0,04 m
Dit	0,04 m	0,02 m

Taula 6.4 - Llargada i amplada dels ossos del *LegModel*

El fèmur és un os que té els extrems molt gruixuts i el centre relativament prim. Per aquesta raó, com que l'os es modela com un rectangle, el gruix que vaig fer servir per al model va ser la mitjana entre l'extrem i el centre.

Igual que en el braç hi ha molta poca informació sobre les dimensions dels músculs. Però aquesta vegada vaig trobar un model en 3D de la cama inferior **[Figura 6.4] [TMA. 2023]**. Com que sabia les llargades dels ossos vaig poder aproximar les llargades i posicions dels músculs. Al final vaig poder definir el diagrama següent **[Figura 6.5]**.

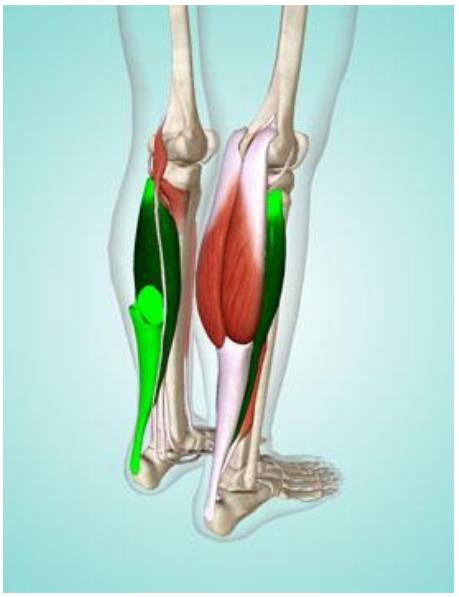


Figura 6.4 - Model 3D de la cama inferior

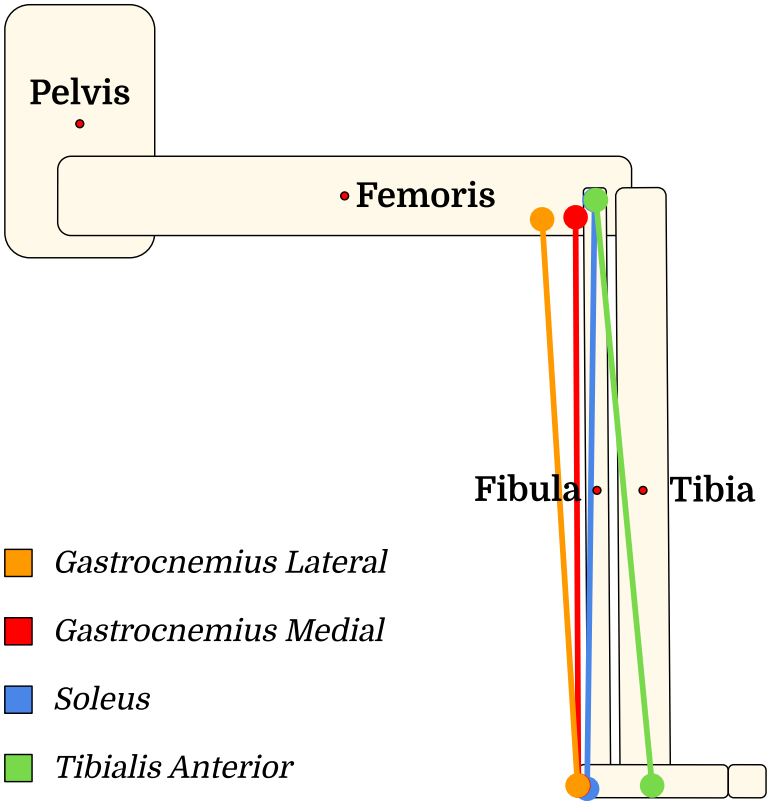


Figura 6.5 - Diagrama anatòmic d'un braç amb els ossos i músculs anomenats

6.2.2.2. Programació de la subclasse *LegModel*

Per programar aquest model vaig tornar a definir els 5 mètodes virtuals de la classe *MuscularModel*.

En el primer mètode, *defineMain()*, vaig definir les variants **[Taula 6.5]** i pesos **[Taula 6.6]** que tindrà aquest model. Aquest model té uns pesos molt més grans que el del braç perquè els bessons són uns músculs molt forts i, per tant, calen pesos grans per comprovar la seva força màxima.

Índex	Variants
0	*All Muscles*
1	*All Gastrocnemius*
2	*All Gas + Soleus*
3	*Tibialis Anterior*
4	*Soleus + Tibialis Anterior*

Taula 6.5 - Variants del model *LegModel*

Índex	Músculs
0	*10 kg*
1	*50 kg*
2	*100 kg*
3	*200 kg*

Taula 6.6 - Pesos del model *LegModel*

El següent mètode és el *defineBones()*. Vaig fer servir l'esquema de la figura 6.5 i la taula 6.4 per definir les posicions relatives i les dimensions dels ossos. En aquest model la pelvis i el dit estan fixats, però quan fixava el peu a un objecte *Nail* el model era inestable. Per això, finalment vaig fixar-lo des del constructor mitjançant el paràmetre '*isStatic = true*', cosa que ho va arreglar.

En el mètode *defineJoints()* vaig fer les unions entre els ossos. La pelvis es fixa al món unint-se a un objecte *Nail* amb una unió de tipus *Glue*. Per unir la pelvis i el fèmur vaig fer servir una unió angular (tipus *Hinge*). La tíbia i el fèmur també es van unir amb una unió angular que representa el genoll. El peroné es va unir a la tíbia amb una unió de tipus *Glue*. El peu es va unir a la tíbia amb una altra unió angular amb uns límits d'angle iguals que un el turmell de veritat. Finalment, el dit es va unir al peu amb una unió angular.

El pròxim mètode és el *defineMuscles()*. Vaig fer servir la figura 6.5 i el model en 3D **[Figura 6.4]** per definir les posicions i mides dels músculs. En aquest model no calia cap extensió en els músculs. El mètode està definit perquè creï els músculs corresponents dependentment de la variant del model.

Com que els bessons tenen el tendó inferior molt més llarg que el superior, el múscul no està centrat. L'objecte *Muscle*, per defecte, fa els tendons simètrics i, per tant, el múscul queda centrat. Per això, en aquest cas, vaig haver d'utilitzar la funció *setMuscleSymmetry()* amb un valor de 0,25 per posicionar-lo.

Finalment, en el mètode *defineWeights()* vaig fer el codi que crea l'objecte *Weight* segons el pes escollit. Per sort el codi era molt semblant al del model *ArmModel* i, per tant, vaig poder reutilitzar-lo. El pes s'uneix al final del fèmur amb una unió de tipus *Glue*, de manera que queda sobre el genoll.

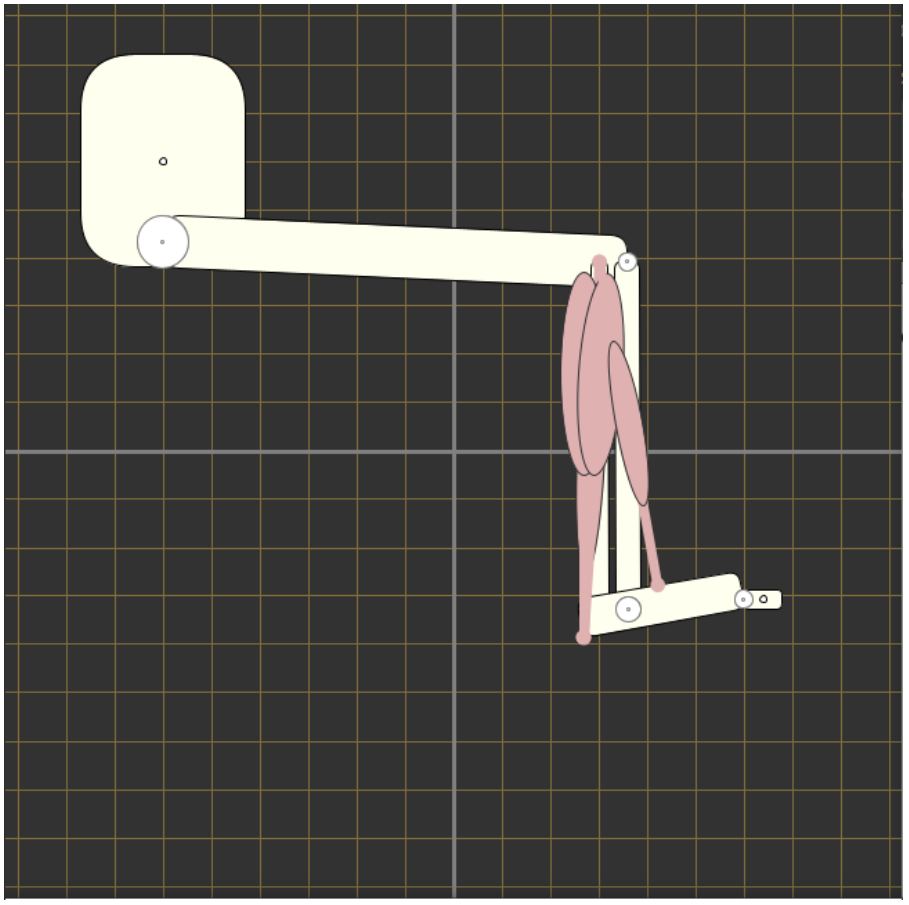


Figura 6.6 - Captura de pantalla de la simulació del model *LegModel*

Ara el model musculoesquelètic *LegModel* ja està definit. El resultat **[Figura 6.6]** ha quedat molt semblant a una cama real i els músculs generen els moviments correctes. El codi d'aquest model es pot consultar en l'annex **[Annex C.7]**.

6.3. Demostració interactiva

6.3.1. La interfície gràfica

Una vegada programats els models musculoesquelètics anteriors, és possible engegar la simulació i veure la dinàmica dels models. Però, al no haver-hi forces, el model només oscil·la fins a arribar a un punt d'equilibri en el qual es queda quiet. Això és així perquè no s'ha produït cap activació muscular.

A més, la simulació només és utilitzada per visualitzar el moviment del sistema musculoesquelètic, però molts valors interns calculats en el model del múscul no són accessibles. Per aquestes raons cal una interfície gràfica que, a més, permeti triar la variant, el pes, i el nivell de simulació.

Aquesta interfície gràfica, també anomenada GUI (*Graphical User Interface*), serveix per controlar i visualitzar la simulació més intuïtivament i, alhora, amb més precisió.

La interfície d'aquesta demostració té una estructura multifinestra **[Figura 6.7]** amb tres seccions: una que mostra la simulació, una pel control interactiu, i l'última per mostrar les gràfiques.

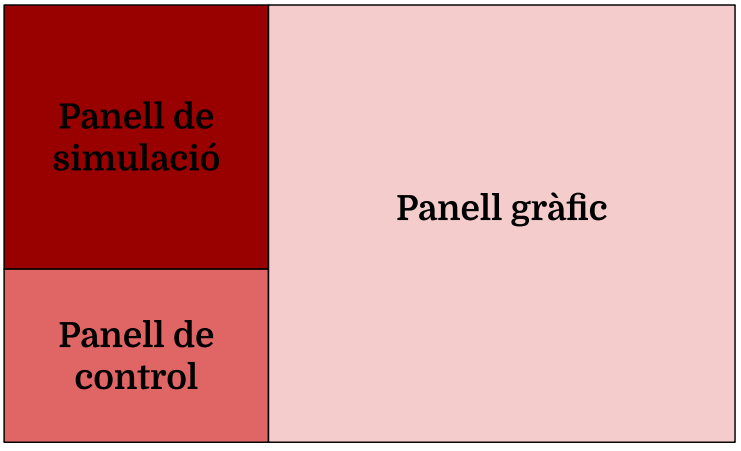


Figura 6.7 - Diagrama de les diferents finestres de la interfície gràfica

El panell de simulació està controlat per la classe *MuscularModel*. Utilitza la funció *display()* del *MuscularModel* per dibuixar en temps real el model de la

simulació. És útil perquè l'usuari pugui veure el que està passant dins la simulació sense haver d'interpretar gràfiques.

La següent finestra és el panell de control **[veure apartat 6.3.2.]**. El panell de control conté botons i llistes desplegables per controlar la simulació, des de si està en marxa o en pausa fins a quin model exacte ha de carregar. També, hi ha un control lliscant per controlar l'activació de qualsevol múscul. D'aquesta manera controlar un múscul és més intuïtiu i ràpid per l'usuari. Finalment, hi ha botons per exportar els valors exactes de les gràfiques i poder fer servir-los en qualsevol programa extern.

L'última finestra és el panell gràfic, que serveix per visualitzar en temps real com varien els valors interns dels músculs mitjançant una sèrie d'objectes *LiveGraph*. Aquests objectes serveixen per crear gràfiques i afegir-hi punts dinàmicament. La classe *LiveGraph* **[veure apartat 6.3.3.]** ha sigut programada específicament per aquest projecte però és molt versàtil i, per tant, pot ser utilitzat en qualsevol altre projecte.

6.3.2. Panell de control

El panell inferior a la demostració inclou els elements de control que permet a l'usuari interactuar amb el programa. Hi ha moltes maneres de plantejar el tipus i distribució d'aquests elements gràfics per controlar els models i interactuar amb els músculs durant un experiment. Per això, abans de programar un GUI és recomanable provar diferents dissenys fins a obtenir un diagrama que compleixi els objectius necessaris i sigui intuïtiu.

Inicialment, es va dissenyar aquesta interfície **[Figura 6.8]**. Aquest disseny estava bé com a primer esbós, però tenia alguns problemes. Un dels problemes és que dona molt espai al control dels gràfics. Quan això no és realment necessari, ja que per la majoria d'experiments n'hi ha prou observant quatre gràfics fixos (activació, tensió muscular, forces activa i passiva, i contracció).

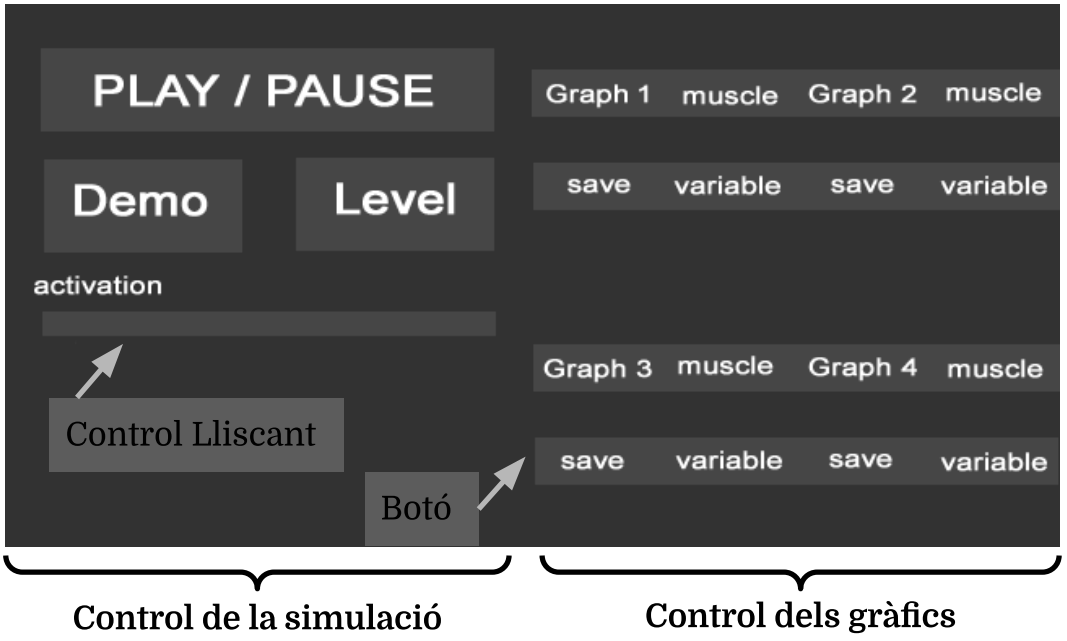


Figura 6.8 - Primer diseny del panell de control

L'altre problema és l'invers, no dona prou espai al control a la simulació. El control lliscant d'activació no permet controlar a quin múscul afecta, cosa que és important en qualsevol model multimuscular. També falten controls per triar específicament quin model, variant i pes es vol simular.

Per tot això es va fer un altre disseny en què es van eliminar la majoria dels controls dels gràfics, ja que amb quatre botons per exportar els gràfics era suficient. I aquest espai recuperat va servir per afegir controls per la simulació sense que la finestra quedés massa plena.

Com es pot veure al disseny final **[Figura 6.9]**, aquesta interfície té molt més control sobre la simulació que el primer disseny.
Per fer servir el panell de control, primer s'ha d'utilitzar les llistes desplegables taronges per especificar el model exacte que vols carregar i la variant que vols simular. A més, pots definir la massa del pes que s'utilitzarà en la simulació i, opcionalment, els elements que se simularan (ossos, articulacions, músculs, pesos).

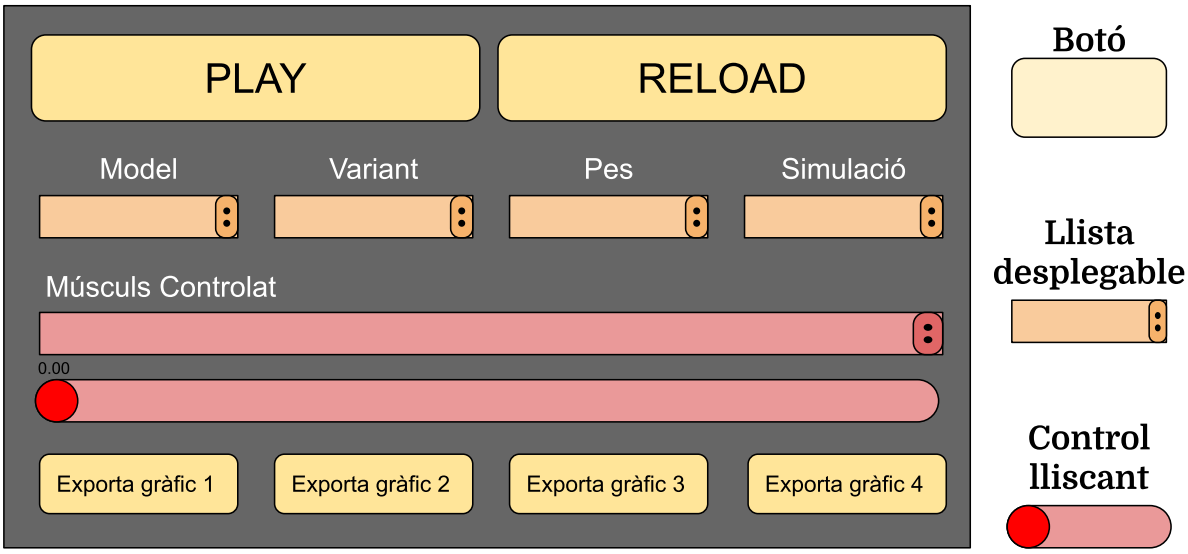


Figura 6.9 - Segon i últim disseny del panell de control

Una vegada definits, es clica el botó *reload* per carregar el model a la simulació. Després es clica el botó *play* per engegar el motor físic, quan estigui en marxa, el botó canviarà a *pause* i servirà per posar-lo en pausa en qualsevol moment.

Ara que el motor està simulant el model ja es pot començar l'experiment que es vulgui fer. Per triar quin múscul es controla s'utilitza la llista desplegable vermella, dins la qual hi ha tots els músculs del model i també els grups de músculs definits. Una vegada seleccionat, es pot fer servir el control lliscant vermell per controlar l'activació del múscul, l'extrem esquerre és 0 i el dret és 1.

Una alternativa per activar el múscul és fer servir, en comptes del control lliscant, la barra d'espai. Cada vegada que es prem l'espai es genera un pols en el nivell d'activació amb una durada d'un *frame* (16,67 ms) i una intensitat màxima.

Quan s'hagi complert l'experiment es poden guardar els valors que s'han obtingut en les gràfiques. Per fer això es clica botó groc de baix que correspongui al gràfic que vulguis guardar. Totes les corbes de la gràfica s'exportaran com un fitxer de text que es pot obrir des de qualsevol full de càlcul.

Ara que el disseny de la finestra està fet, només calia programar-ho. Vaig mirar llibreries que permetessin crear botons, llistes desplegables i controls lliscants. Vaig

trobar dues llibreries: *ControlP5* i *G4P*, al final em vaig decantar per la llibreria *G4P* perquè era més intuïtiva d'utilitzar i permetia més control sobre el comportament dels objectes.

El resultat final de codi, de la demostració i del panell de control, pot consultar-se en els annexos. **[Annex D] [Annex D.2]**

6.3.3. La classe *LiveGraph*

Com s'ha explicat abans, el nucli del panell gràfic és la classe LiveGraph. Aquest objecte visualitza en temps real un gràfic dinàmic que permet rebre una quantitat indefinida de dades. Això és així, perquè quan la quantitat de dades es fa més llarg que la finestra, el gràfic es desplaçarà perquè sempre mostri els valors més recents. També permet mostrar diverses corbes simultàniament, les quals comparteixen l'eix X (temps).

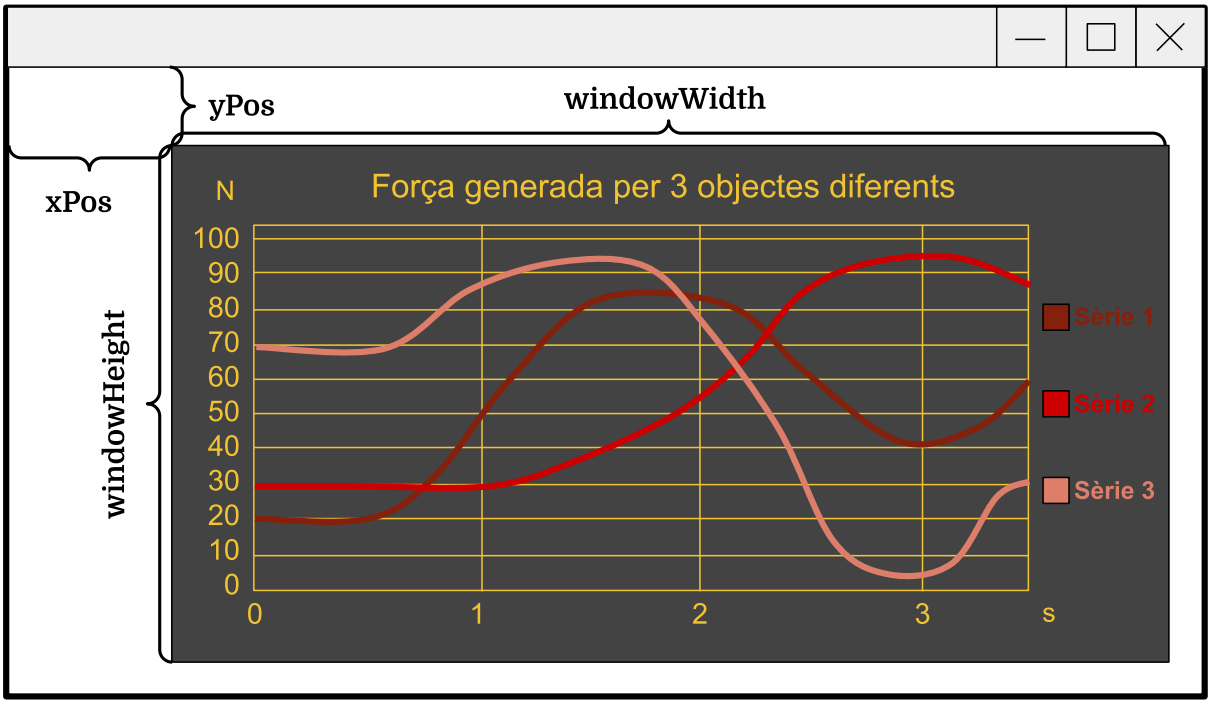


Figura 6.10 - Representació d'un objecte *LiveGraph* amb les variables de posició i mida marcades

La majoria dels seus paràmetres són definibles de manera que es pot personalitzar el seu aspecte i comportament per adaptar-se a diferents aplicacions.

Constructor

Per un objecte *LiveGraph* **[Figura 6.10]** es crida el seu constructor amb els paràmetres corresponents:

`LiveGraph (`	Valor per defecte	Descripció
`float x,`		Coordenada X de la cantonada de la finestra.
`float y,`		Coordenada X de la cantonada de la finestra.
`float windowWidth,`		Llargada de la finestra.
`float windowHeight,`		Amplada de la finestra.
`PApplet window,`		Finestra de la llibreria G4P on es visualitza el gràfic.
`int numOfChannels)`	`1`	Quantitat de canals en el gràfic.

Els primers paràmetres *x* i *y* defineixen la posició de la cantonada superior esquerra, i *windowWidth* i *windowHeight* defineixen les seves mides.

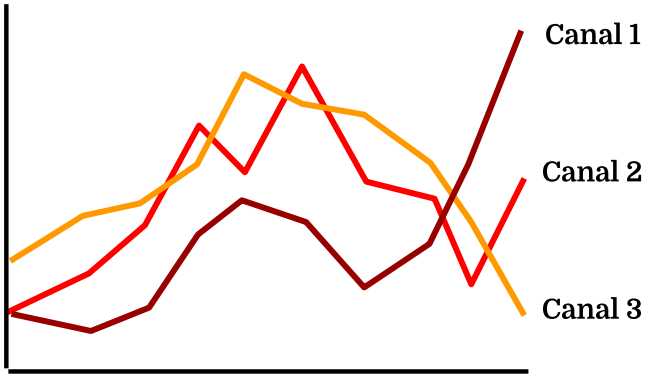


Figura 6.11 - Representació dels diferents canals de l'objecte LiveGraph

El paràmetre *window* és una variable del tipus *PApplet*, que es troba en la llibreria del Processing *G4P*. Aquesta llibreria permet crear diverses finestres simultàniament, per tant, a l'hora de visualitzar el gràfic cal especificar en quina finestra es visualitzarà. Això vol dir que per utilitzar la classe *LiveGraph* cal tenir instal·lada la llibreria *G4P*.

La quantitat de canals del gràfic (*numOfChannels*) són quantes corbes seran dibuixades en el mateix gràfic **[Figura 6.11]**.

Mètodes

Amb el constructor s'han definit els paràmetres bàsics. Però després hi ha les funcions: *setAxisProperties()*, *serGraphProperties()*, *setGridProperties()*, *setTextProperties()* i *setTimeIncrement()* per definir les propietats i l'aspecte del gràfic.

Mètode	Descripció
`void setAxisProperties(float tMax, float yMin, float yMax, float timeIncrement, String yUnit)`	Defineix les propietats dels eixos. **tMax** = El temps que es visualitza. **yMin** = El valor mínim de l'eix Y. timeIncrement = L'increment de temps entre els punts **yUnits** = Unitats de l'eix Y.
`void setGraphProperties(color backgroundColor, color lineColor, float lineThickness)`	Defineix les propietats de l'estil del gràfic.
`void setGridProperties(float tGridIncrement, float yGridIncrement, color gridColor)`	Defineix les propietats de la matriu del gràfic.
`void setTextProperties(float textSize, int decimalPlaces, String title, float titleSize, String[] channelNames, color textColor)`	Defineix les propietats dels texts del gràfic.
`void addPoint(float num)`	Afegeix el següent punt del gràfic.
`void addPoint(float[] nums)`	Afegeix el següent punt (una llista de punts) del gràfic. Utilitzat quan hi ha diversos canals.
`void resetData()`	Esborrar les dades del gràfic.
`void export(String tableName)`	Guarda les dades del gràfic com un fitxer CSV.

`void display()`	Visualitzar el gràfic en la finestra definida.

Una vegada creat l'objecte *LiveGraph*, els valors s'afegeixen amb la funció *addPoint()*, la qual afegeix un punt al final de les dades. Si un gràfic té diversos canals, els valors s'afegeixen amb una llista de valors.

Quan l'usuari hagi acabat amb el seu experiment, pot fer servir la funció *export()* per guardar les dades que s'han obtingut durant la simulació. Les guarda com un fitxer CSV (*Comma-Seperated Values*) que és un format que permet guardar taules en un fitxer de text. Com que aquest format està molt estandarditzat, la majoria de programes poden llegir-lo.

Aquest format utilitza comes per separar columnes i *returns* per separar files. Per exemple, aquesta taula **[Taula 6.7]** seria així convertida en el format CSV **[Figura 6.12]**.

Activació	Tensió
0	0
0.2	100
0.4	200
0.6	300
0.8	400
1	500

```
Activació, Tensió
0,0
0.2,100
0.4,200
0.6,300
0.8,400
1,500
```

Taula 6.7 - Representació de com una taula és exportada en un fitxer CSV

Figura 6.12 - Representació de com una taula és exportada en un fitxer CSV

Per tant, aquesta funció agafa la taula que conté les dades de la simulació, ho converteix en el format correcte i ho guarda amb un fitxer en la mateixa carpeta de la simulació.

Utilització

Per utilitzar la classe *LiveGraph* primer s'ha de declarar una instància de la classe que serà el gràfic.

```
LiveGraph graph;
```

Quan la instància és declarada, s'inicialitza amb el constructor i es defineix tots els seus valors amb les funcions següents.

```
void setup()
{
  graph = new LiveGraph( ... );
  graph.setAxisProperties( ... );
  graph.setGridProperties( ... );
  graph.setGraphProperties( ... );
  graph.setTextProperties( ... );
}
```

Ara el gràfic ja està definit i es pot fer servir. Per utilitzar-lo, en la funció draw, es pot cridar la funció *addPoint()* per afegir el següent valor del gràfic. I quan s'ha afegit el valor, es visualitza el gràfic amb la funció *display()*.

```
void draw()
{
  background(0);
  graph.addPoint(value);
  graph.display();
}
```

El codi de la classe *LiveGraph* es pot consultar en l'annex D.4 o descarregar al GitHub sota la llicència de codi obert, *GNU General Public License v3.0*.

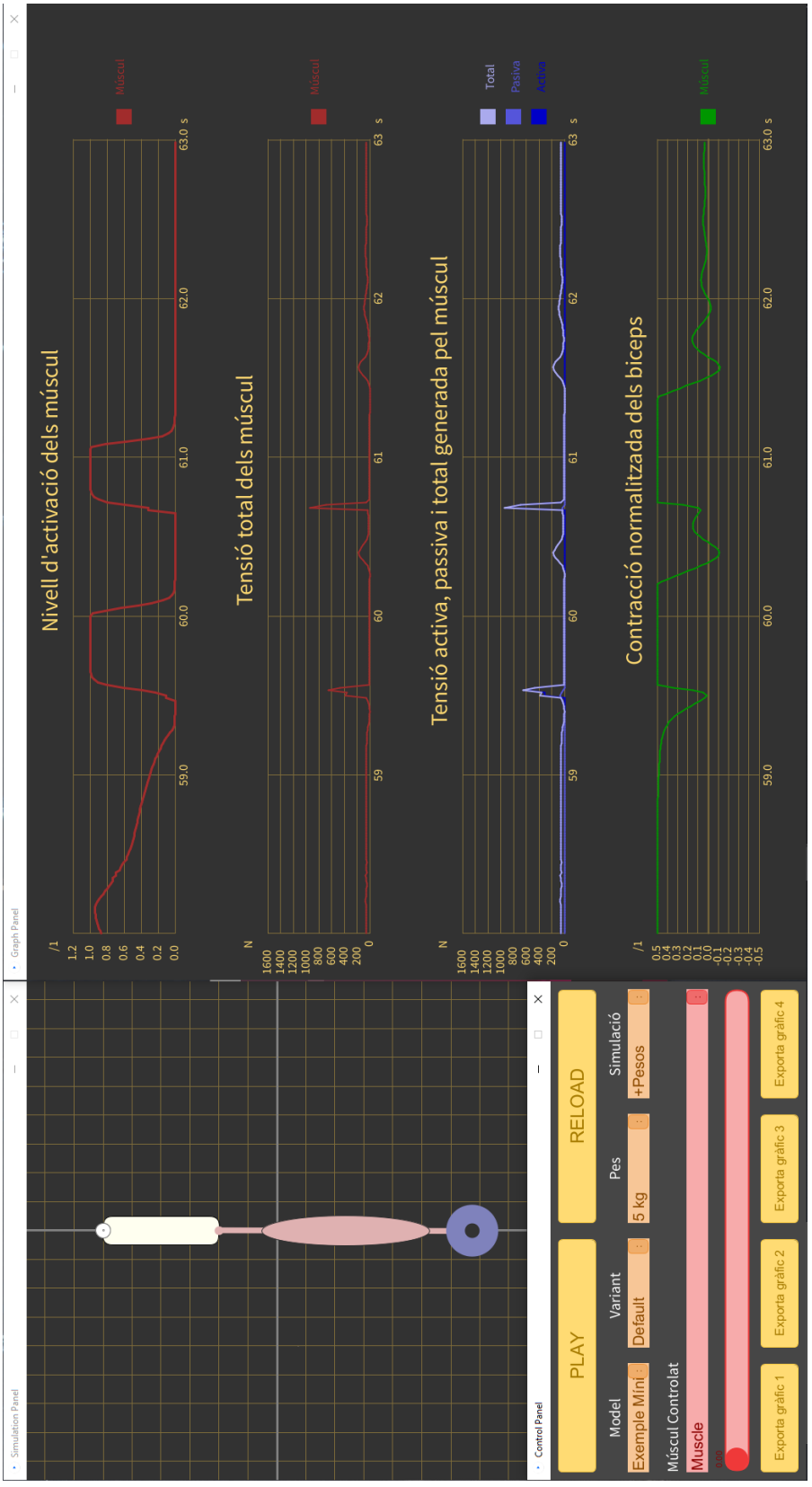


Figura 6.13 - Captura de pantalla de la interfície gràfica final

6.4. Experiments i resultats

6.4.1. Introducció

En la secció anterior s'ha explicat com funciona el programa de demostració. Amb aquesta eina es pot veure que la simulació dels models musculars es visualitza d'una manera fluida i realista. Però també és important observar el comportament intern del múscul durant la simulació i comprovar la seva similitud amb els comportaments de músculs reals.

En aquesta secció es mostren diferents experiments i comparacions entre els comportaments de músculs biològics en situacions determinades i els comportaments dels músculs simulats en les mateixes situacions.

6.4.2. Contracció muscular

La primera comparació és d'una contracció simple **[Figura 6.14]**. Quan el múscul rep un pols d'activació el múscul es contrau i després es relaxa. El temps que triga el múscul a començar la contracció es diu el període de latència. El període on el múscul està en contracció es diu el temps de contracció, i just després comença el temps de relaxació que dura fins que torna a la llargada inicial. **[BYU, 2023]**

El temps de contracció i relaxació és molt més llarg que el període de latència.

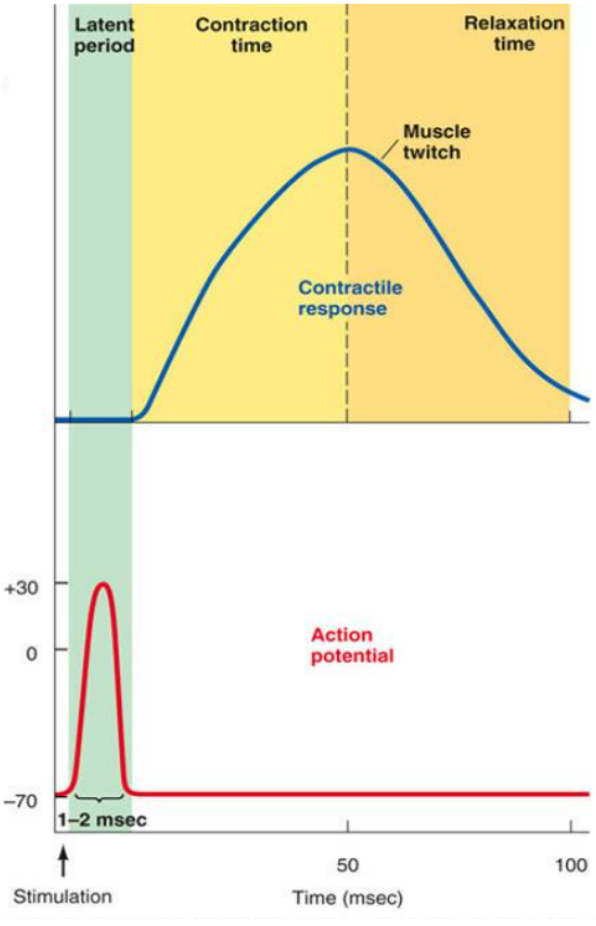


Figura 6.14 - Demostració 1: Contracció muscular a partir d'un pols - *www.austincc.edu/apreview/PhysText/Muscle*

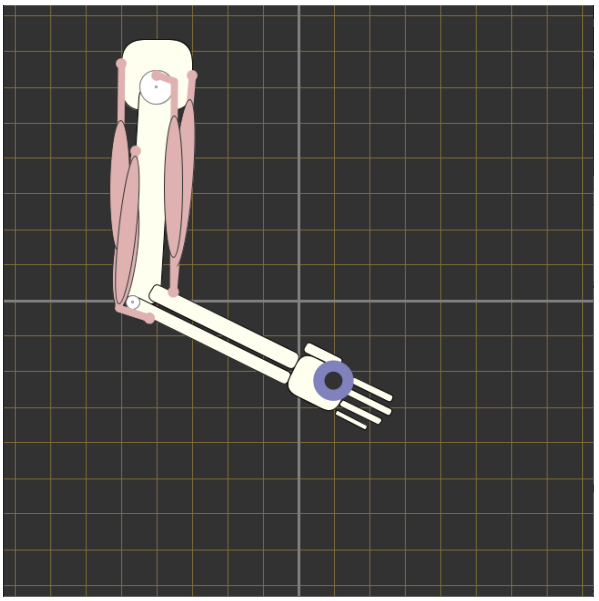


Figura 6.15 - Captura de pantalla de la simulació durant la demostració

Per realitzar aquest experiment vaig fer servir aquest model amb les especificacions següents:

Model:	Braç Superior
Variant:	All Muscles
Pes:	1 kg
Simulació:	+Pesos

Quan ja tenia el model carregat a la simulació **[Figura 6.15]** vaig simular-lo durant uns segons fins que estigués en la seva posició de repòs. Després vaig activar un pols d'una duració d'un *frame* (16,7 ms) que va fer que el múscul es contragués. Finalment, vaig esperar fins que el braç tornés a la seva posició de repòs i vaig parar la simulació.

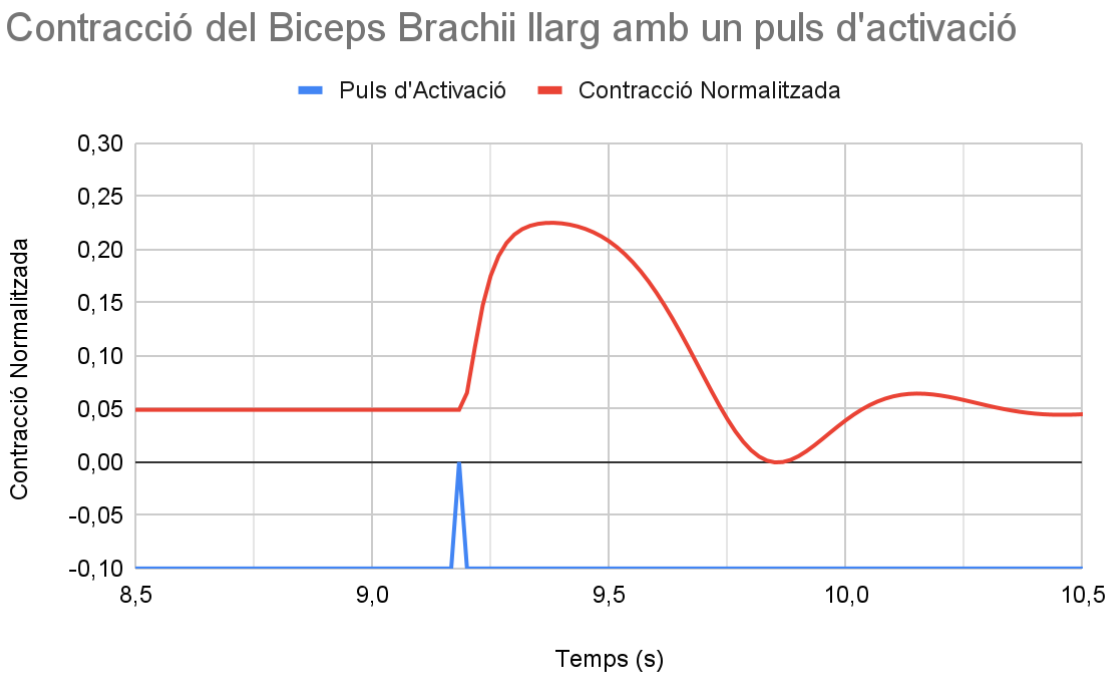


Figura 6.16 - Demostració 1: Gràfica obtinguda a patir del fitxer CSV de la simulació

Ara que l'experiment estava complet vaig exportar les dades dels gràfics de la interfície. A partir de les dades vaig poder obtenir una gràfica **[Figura 6.16]** més nítida i ampliada que la gràfica de la interfície.

Com es pot veure, la simulació muscular mostra un comportament molt similar als músculs biològics. En rebre el pols d'activació el múscul comença la contracció, i triga un temps a arribar a la màxima contracció. Aquest comportament és igual que el de la figura 6.14.

Al final de la contracció, el múscul no és estable instantàniament, sinó que oscil·la durant mig segon. Això és diferent que en la figura 6.14 perquè aquesta és una simplificació, però la simulació inclou la inèrcia del múscul.

Finalment, la latència de la contracció no és present a la simulació perquè el treball implementa un model muscular, no neurològic, on el nivell d'activació del múscul es controla directament.

6.4.3. Força activa i força passiva

Aquest experiment observa la quantitat de força activa i força passiva durant la relaxació d'un múscul. Durant la relaxació del múscul la seva llargada augmenta i, com s'explica en el capítol 2.3.1., la seva força activa es redueix. Per contra, la força passiva augmenta, ja que el múscul actua com una molla. Aquest comportament es pot veure en la part d'allargament de la figura 6.17.

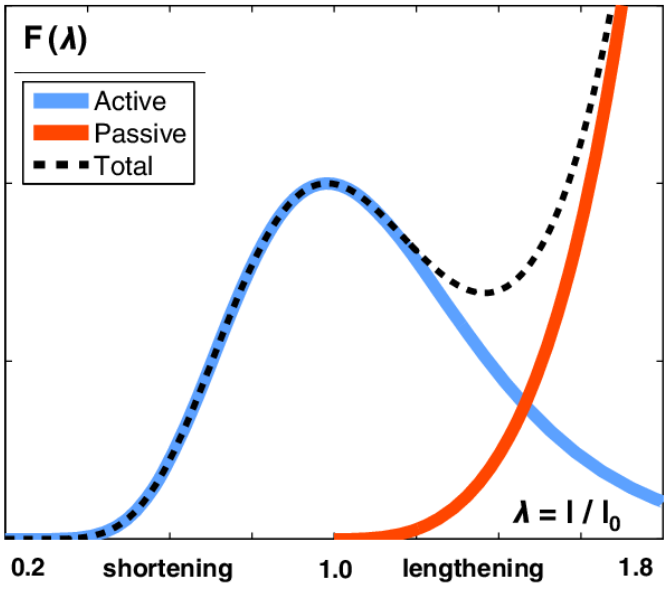


Figura 6.17 - Demostració 2: Força activa i força passiva d'una relaxació muscular - *[Wisdom, K., et al. 2014]*

Com s'explica en el capítol 1.2.2.1., el múscul genera la força activa

amb les proteïnes actina i miosina i la tensió passiva és generada amb la proteïna titina. La força passiva no té un límit teòric de força que pot generar, per tant, pot generar una força major a la força activa màxima. El fet que no tingui límit de força, fa que pugui generar més tensió de la que el múscul i tendó pot aguantar. Per aquesta raó els tendons es trenquen quan el múscul està massa allargat i, per tant, la força passiva és extrema. **[UPMC. 2023]**

Per fer aquest experiment vaig fer servir aquestes especificacions pel model de la simulació. **[Figura 6.18]**

Model:	**Variant:**	**Pes:**	**Simulació:**
Exemple Mínim	Default	50 kg	+Pesos

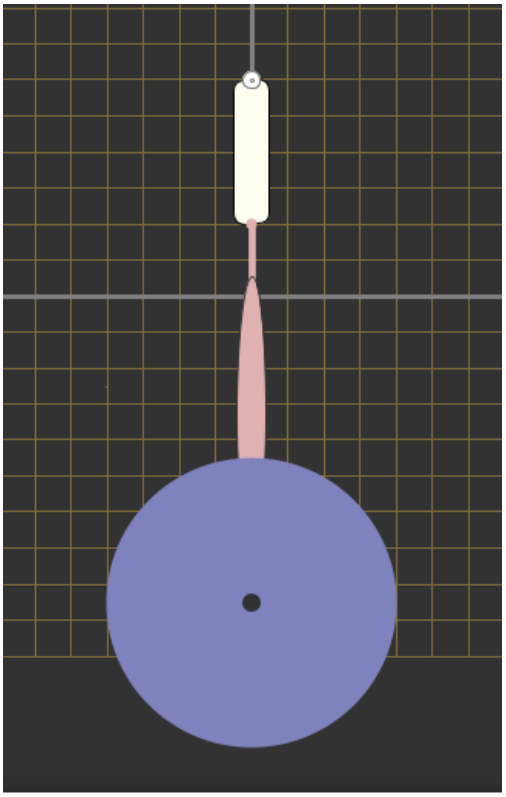


Figura 6.18 - Captura de pantalla de la simulació durant la demostració

Vaig fer servir el model mínim perquè com que el múscul està connectat directament al pes, sense palanques mecàniques, les forces es poden observar més clarament. També vaig utilitzar un pes molt gran perquè així el múscul s'allargarà més i es podrà veure millor la força passiva.

Aquest experiment el vaig realitzar controlant l'activació del múscul. A l'inici l'activació estava a 1, però durant 3 segons la vaig reduir linealment. Això va fer que la força activa es reduís i, per tant, la llargada del múscul hauria d'augmentar.

Quan vaig acabar l'experiment vaig guardar les dades dels gràfics de llargada del múscul i tensió del múscul per poder obtenir una gràfica composta dels dos eixos. Aquesta gràfica té la tensió en l'eix Y i la llargada normalitzada en l'eix X. Això s'ha pogut fer perquè les dues gràfiques de la interfície comparteixen l'eix X de temps.

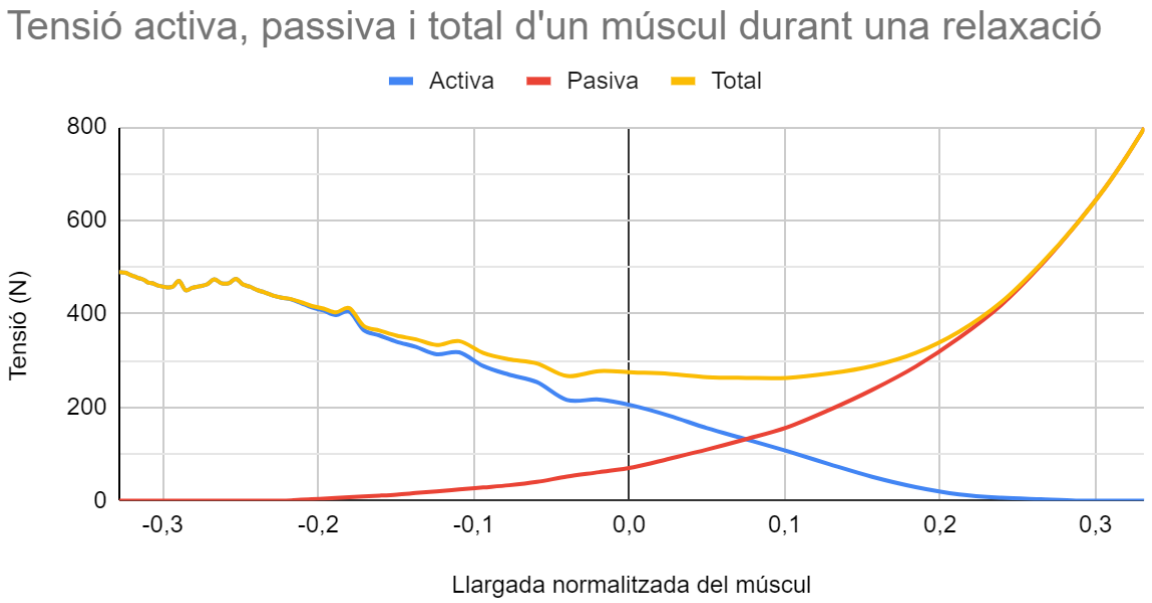


Figura 6.19 - Demostració 2: Gràfica obtinguda a patir del fitxer CSV de la simulació

En la gràfica obtinguda **[Figura 6.19]** es pot veure les similituds amb la figura 6.17. La gràfica obtinguda només mostra la meitat dreta de la figura original, ja que per obtenir la meitat esquerra necessitaria un pes que variï la massa dinàmicament. El gràfic mostra que quan la llargada del múscul augmenta la força activa disminueix i, simultàniament, la força passiva augmenta exponencialment.

6.4.4. Contraccions isomètriques i isotòniques

Com s'ha explicat en el capítol 1.2.1.3., els músculs estriats són capaços de realitzar contraccions isomètriques i isotòniques. Les contraccions isomètriques són contraccions musculars on la llargada del múscul es manté constant encara que la tensió del múscul augmenti o disminueixi. Contràriament, les contraccions isotòniques són contraccions musculars on la tensió generada és constant, però la llargada del múscul varia.

6.4.4.1. Contracció isomètrica

Hi ha dos tipus de contraccions isomètriques:

- Les contraccions **superant** ocorren quan un múscul intenta moure un objecte immòbil, bàsicament quan la força requerida per moure l'objecte supera la força màxima que pot fer el múscul.

- Les altres són les **complaent**. Aquestes contraccions passen quan el múscul està aguantant un pes voluntàriament. El múscul podria moure el pes però només fa força necessària perquè l'objecte es mantingui estàtic. **[LibreTexts. 2023]**

El següent experiment consisteix a produir una contracció isomètrica superant. És a dir, mitjançant un pes elevat, produir una tensió en el múscul sense que aquest es contragui. **[Figura 6.20]**

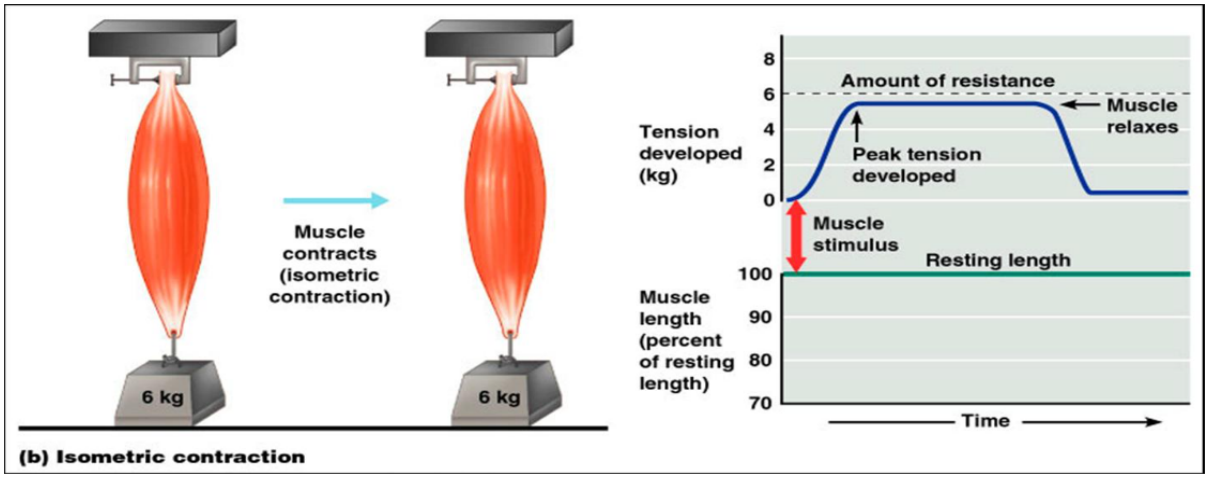


Figura 6.20 - Demostració 3.1: Contracció isomètrica superant - *www.austincc.edu/apreview/PhysText/Muscle*

Per dur a terme l'experiment vaig fer servir aquest model i especificacions. **[Figura 6.21]**

Model:	Braç Superior
Variant:	All Muscles
Pes:	100 kg
Simulació:	+Pesos

Vaig utilitzar un pes de massa de 100 kg perquè els bíceps no tenen suficient força per aixecar aquest pes i, per tant, la llargada del bíceps serà constant.

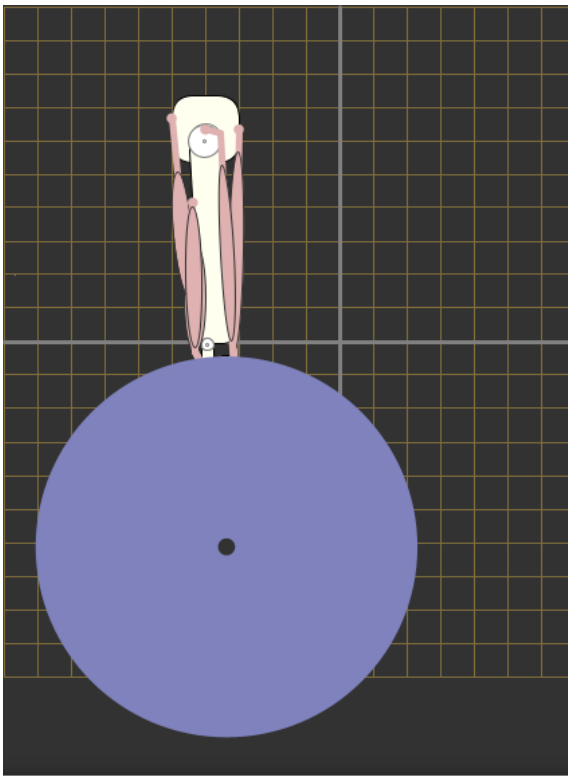


Figura 6.21 - Captura de pantalla de la simulació durant la demostració

Per dur a terme aquest experiment vaig reprogramar la interfície perquè en comptes de controlar l'activació controli la tensió. Vaig augmentar la tensió ràpidament fins a una mica més de 300 N, després vaig deixar la tensió constant durant uns segons i la vaig tornar a baixar. Així vaig poder obtenir una gràfica de la tensió semblant a la de la figura 6.20.

Quan vaig acabar l'experiment, vaig exportar les dades dels gràfics per poder traçar un gràfic més nítid.

Com es pot veure, el gràfic **[Figura 6.22]** mostra el mateix comportament que en la figura 6.20. Quan la tensió activa ha augmentat a 330 N la llargada del múscul no ha variat, el que significa que la contracció ha sigut isomètrica.

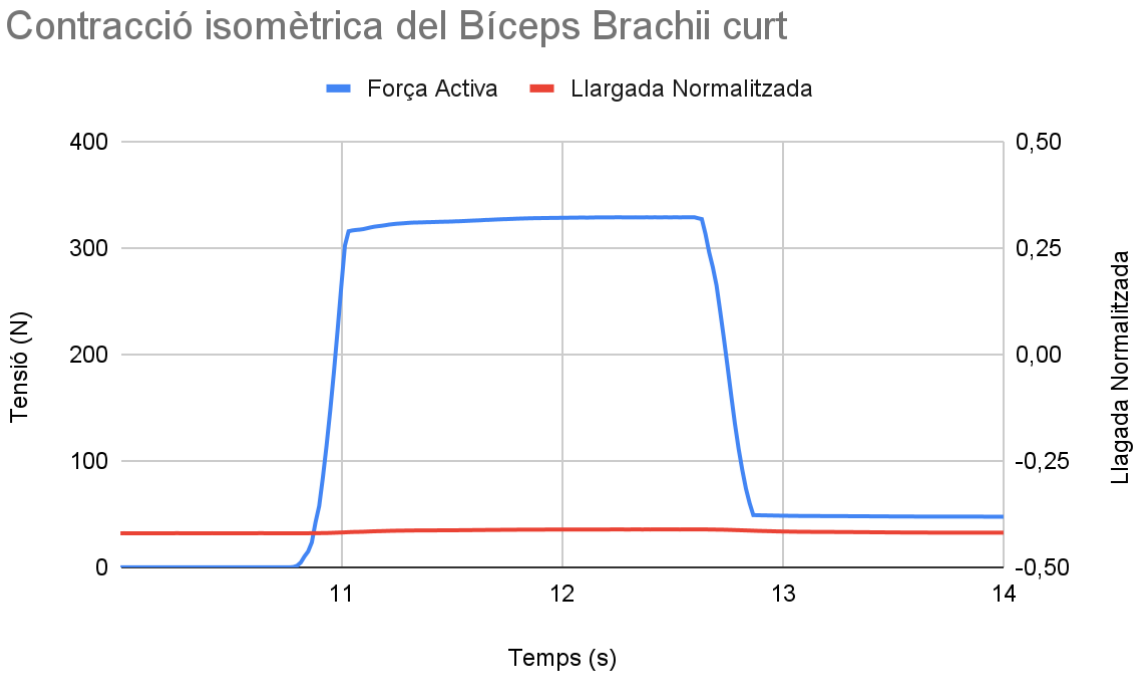


Figura 6.22 - Demostració 3.1: Gràfica obtinguda a patir del fitxer CSV de la simulació

6.4.4.2. Contracció isotònica

Com s'ha explicat abans les contraccions isotòniques es produeixen quan la tensió del múscul és constant però la llargada varia. Aquestes contraccions només poden

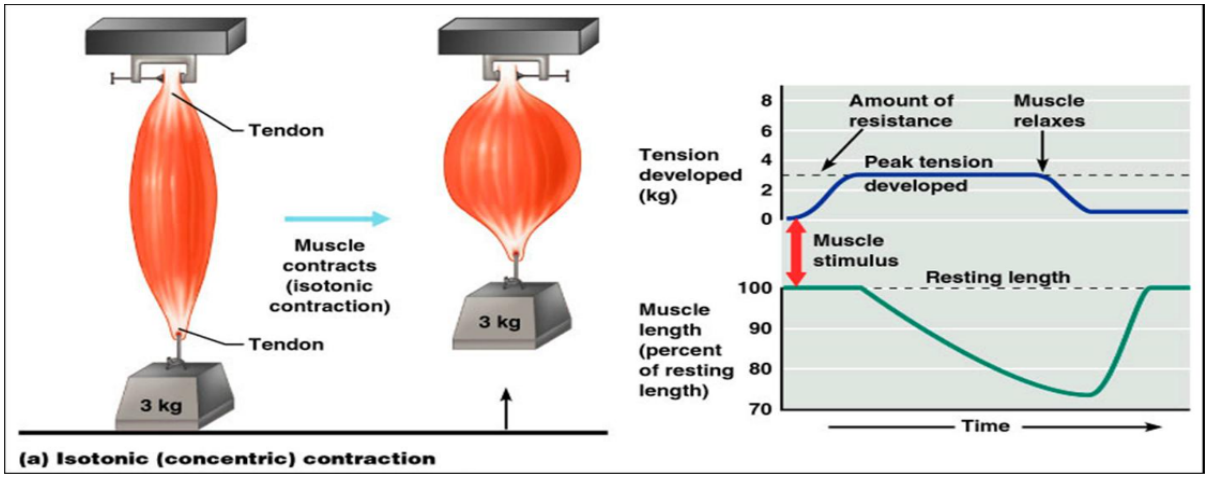


Figura 6.23 - Demostració 3.2: Contracció isotònica concèntrica i acèntrica - *www.austincc.edu/apreview/PhysText/Muscle*

passar quan la força màxima que pot generar un múscul és major que la força que cal per aixecar el pes. Aquestes contraccions poden ser *concèntriques* (el múscul es contrau) o *acèntriques* (el múscul s'allarga) **[LFW. 2023].**

En aquest cas l'experiment consisteix a produir una contracció isotònica. És a dir, causar una tensió muscular constant durant una reducció de la seva llargada. **[Figura 6.23]**

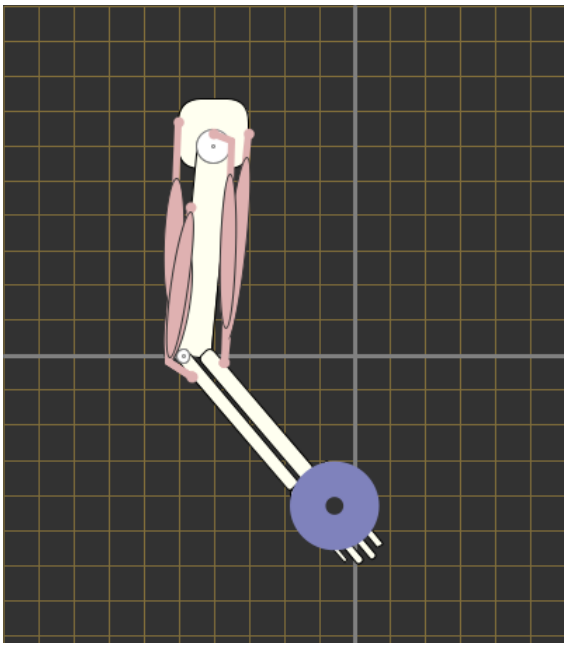


Figura 6.24 - Captura de pantalla de la simulació durant la demostració

Per realitzar l'experiment vaig utilitzar aquest model i especificacions. **[Figura 6.24]**

Model:	Braç Superior
Variant:	All Muscles
Pes:	5 kg
Simulació:	+Pesos

Vaig utilitzar un pes de massa de 5 kg perquè el pes farà que la contracció sigui més lenta i, per tant, es podrà observar millor. Però no vaig posar un pes més gran perquè sinó el múscul no podrà aixecar-lo.

Per realitzar la contracció isotònica també vaig controlar el múscul amb la tensió desitjada i no amb l'activació. Primer vaig augmentar la tensió a uns 350 N i vaig deixar la tensió constant. Després la vaig tornar a baixar, fent que la tensió tingués la mateixa forma que en la figura 6.23.

Com es pot veure, el gràfic **[Figura 6.25]** és molt semblant a la figura 6.23. Durant el temps 2,2 s i 2,9 s la tensió és constant però la llargada del múscul disminueix, això és una contracció concèntrica. També, al final del gràfic, la tensió és zero però la llargada del múscul augmenta, mostrant una contracció isotònica acèntrica. La qual també es veu en la figura 6.23.

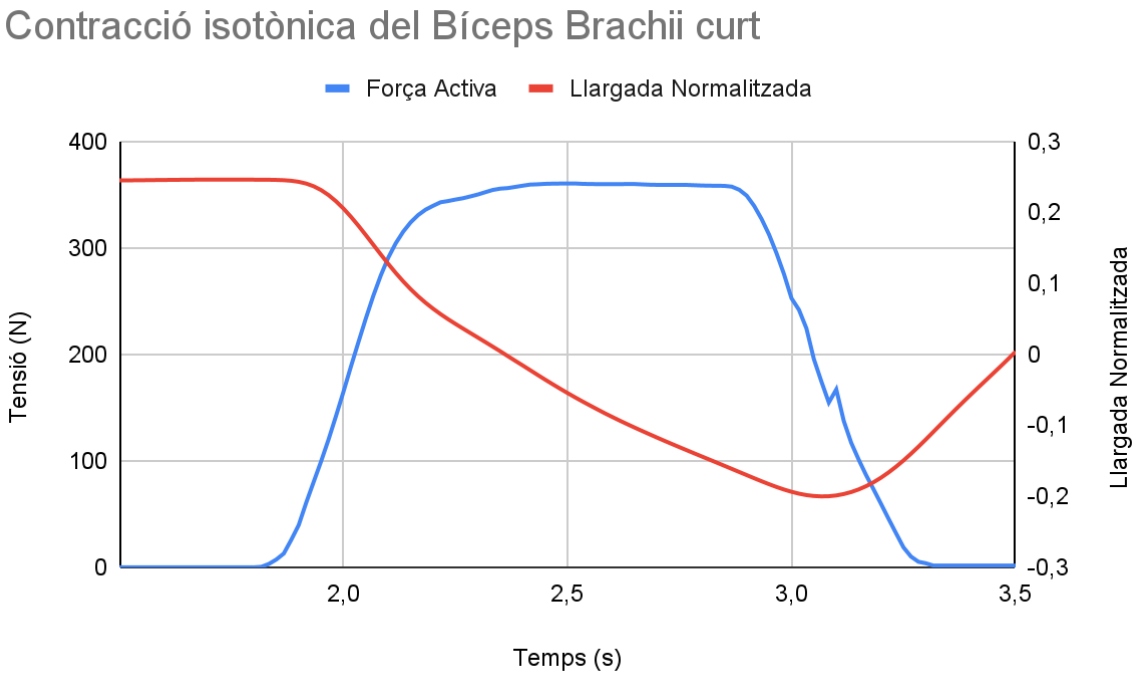


Figura 6.25 - Demostració 3.2: Gràfica obtinguda a partir del fitxer CSV de la simulació

6.4.5. Resposta dinàmica als estímuls

L'última comparació és la resposta dinàmica d'un múscul a estímuls (polsos d'activació) de diferents freqüències. Com s'ha vist en l'apartat 6.4.2. la contracció i relaxació d'un múscul dura bastant més temps que la durada d'un pols d'activació. Però si dones dos polsos seguits a un múscul les contraccions se sumen, obtenint una contracció màxima major que la contracció màxima amb un pols individual. Per un altre costat, si s'envia molts polsos d'activació al múscul seguits la contracció màxima és molt més gran. Si els polsos són suficientment seguits, la contracció s'estabilitzarà i serà constant fins que els polsos s'aturin. **[Figura 6.26]**

Aquest fenomen es coneix com a suma muscular (*muscular summation*), en el que si una segona contracció arriba abans que la primera s'acabi, se sumaran i crearan una contracció més intensa. No hi ha límit de quantes contraccions poden sumar-se, la freqüència de les contraccions determina la magnitud de la contracció total.

Quan la freqüència de les contraccions està al voltant dels 10 Hz o més la contracció serà constant, aquesta contracció és coneguda com a espasmes o, també, contracció tetànica. És el mateix tipus de contracció que passa quan una persona està rebent una descàrrega elèctrica o quan està infectat amb el bacteri *Clostridium Tetani*. Els espasmes fan que els músculs no es relaxin i, per tant, l'individu no pot controlar els músculs. **[Odell. 2016]**

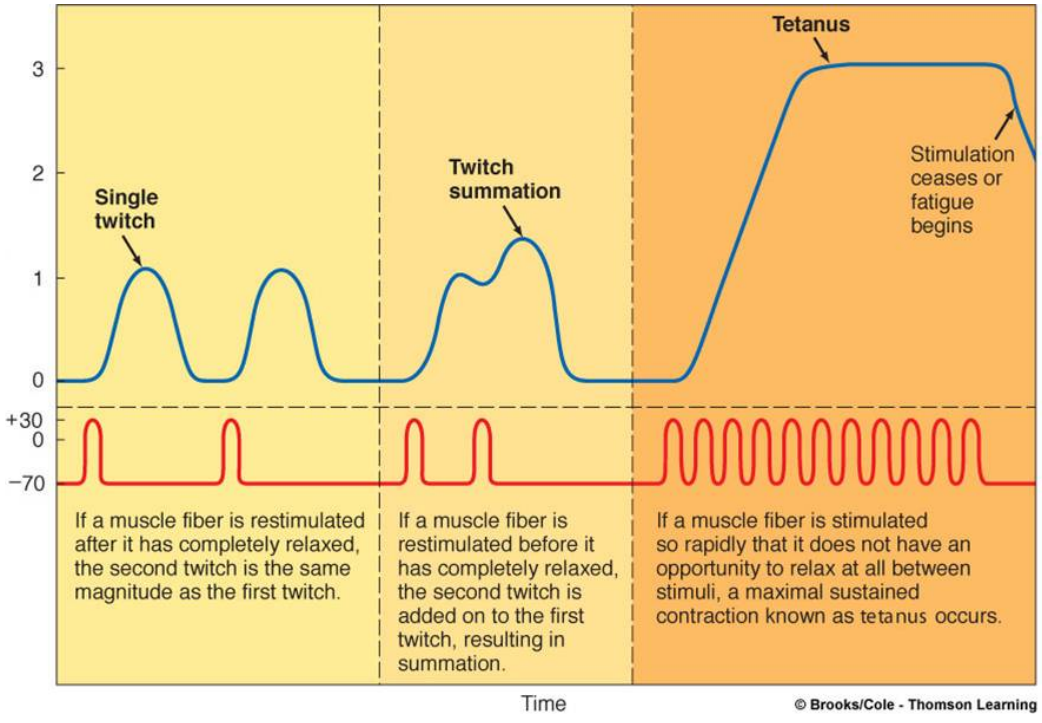


Figura 6.26 - Demostració 4: Resposta dinàmica d'un múscul a estímuls de diferents freqüències - *www.austincc.edu/apreview/PhysText/Muscle*

Per realitzar l'experiment vaig fer servir aquest model i especificacions. **[Figura 6.27]**

Model:	**Variant:**	**Pes:**	**Simulació:**
Braç Superior	All Muscles	5 kg	+Pesos

Vaig fer servir un pes d'una massa de 5 kg perquè és un bon punt entremig perquè la contracció d'un sol pols i la contracció de molts polsos seguits es vegin bé.

Per realitzar aquesta comparació primer vaig activar un pols individual, després vaig activar dos polsos separats per una mica menys de mig segon. Finalment, vaig

activar polsos tan ràpidament com vaig poder, que va resultar ser 9 polsos en una mica més d'un segon (una freqüència de 8 Hz). Quan vaig acabar la demostració vaig exportar les dades per obtenir els gràfics.

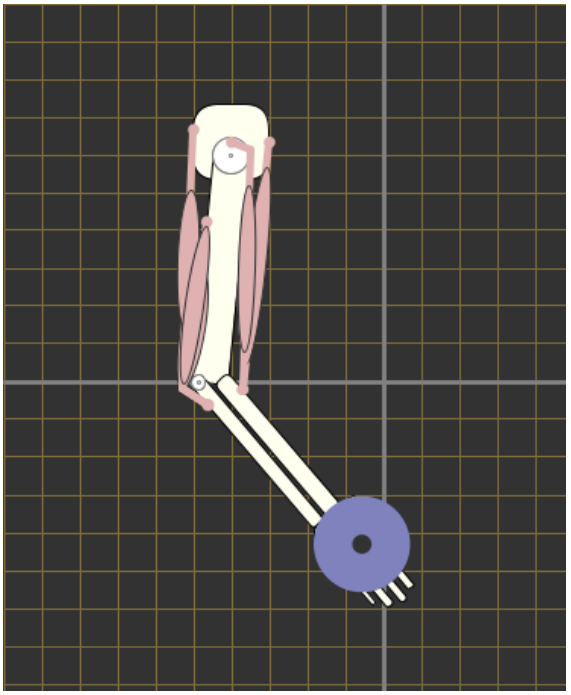


Figura 6.27 - Captura de pantalla de la simulació durant la demostració

Com es pot veure, el múscul de la simulació ha produït uns comportaments **[Figura 6.28]** molt semblants a la figura 6.26. En la primera contracció es veu que passa exactament el mateix que en la demostració de l'apartat 6.4.2. La segona contracció mostra exactament el mateix que la figura 6.26 que els dos polsos creen una contracció més gran que la d'un sol pols. I finalment, en l'última contracció es veu com molts polsos junts creen una sola contracció constant molt més gran que la d'un i dos polsos.

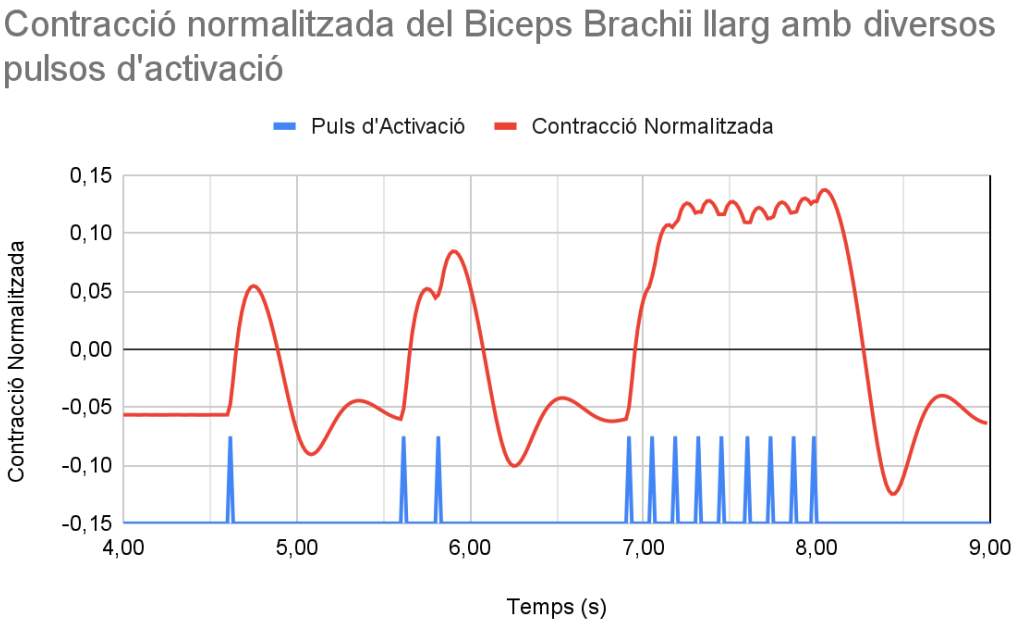


Figura 6.28 - Demostració 4: Gràfica obtinguda a patir del fitxer CSV de la simulació

El gràfic generat és bastant semblant al de la figura 6.26, però l'última contracció no s'estabilitza i oscil·la mínimament. Vaig pensar que això devia ser culpa de la freqüència dels polsos. Per comprovar-ho vaig trobar un estudi **[Cooper & Eccles. 1930]** que recreava aquest mateix experiment empíricament amb músculs d'un gat.

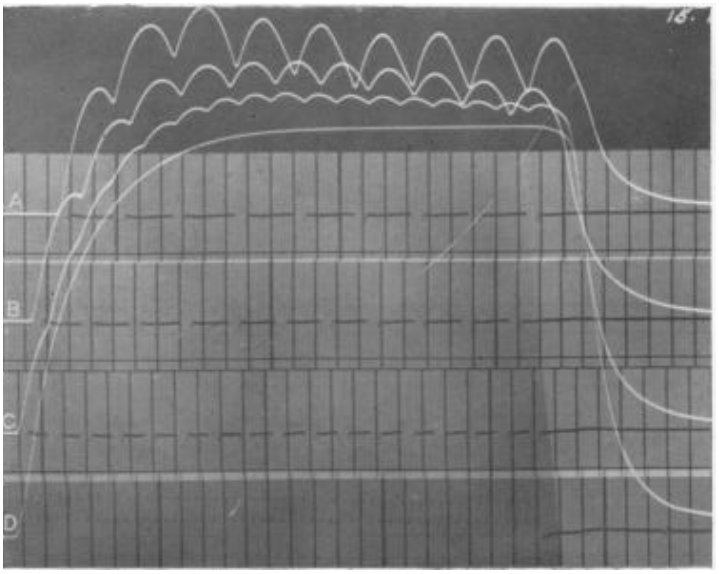


Figura 6.29 - Gràfica empírica que la contracció del múscul real amb diferents freqüències de polsos. (A) 19 Hz, (B) 24 Hz, (C) 35 Hz, (D) 115 Hz - *[Cooper & Eccles. 1930]*

Aquesta gràfica **[Figura 6.29]** mostra com en freqüències baixes la contracció oscil·la però en freqüències altes la contracció s'estabilitza i sembla constant.

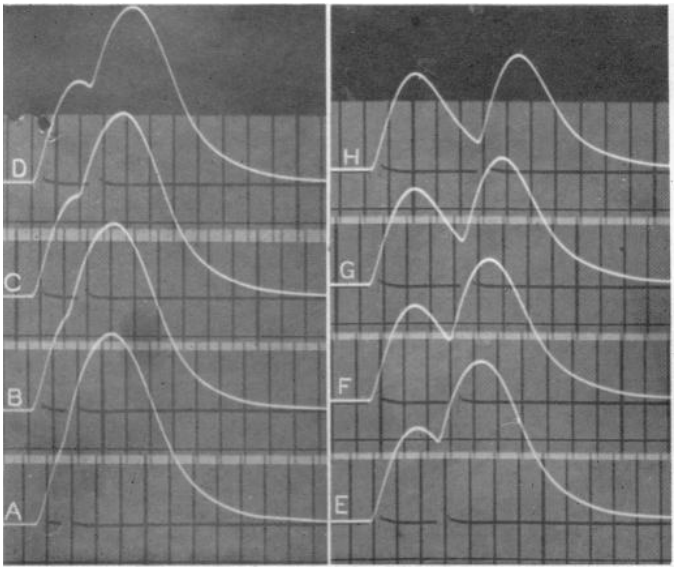


Figura 6.30 - Gràfica empírica que la contracció del múscul real amb dos polsos seguits en diferents freqüències - *[Cooper & Eccles. 1930]*

I la gràfica on s'han activat dos polsos **[Figura 6.30]**, els gràfics són molt més semblants a les de la simulació que els de la figura 6.26. Especialment a l'inici de la contracció on comença bruscament.

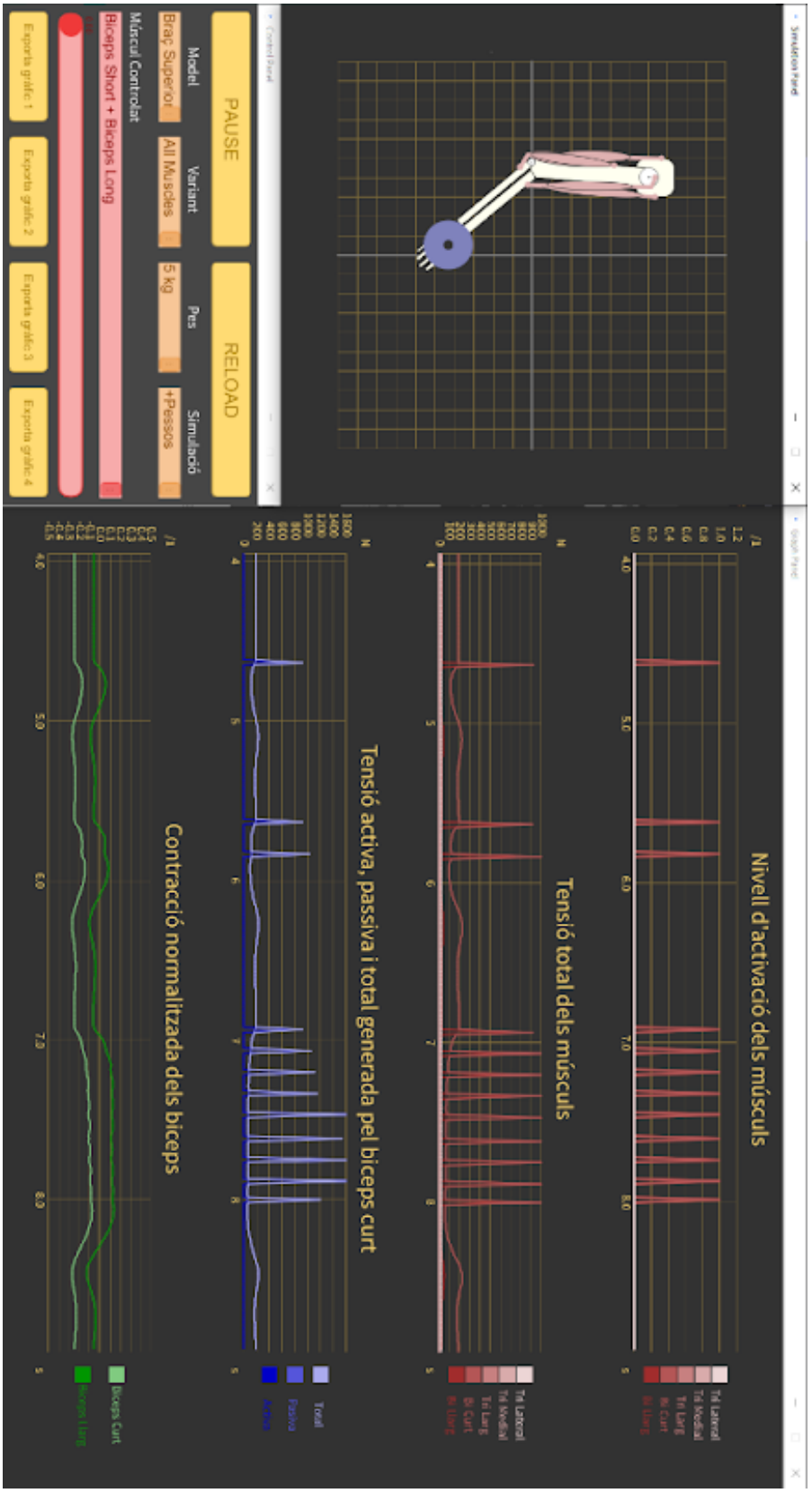


Figura 6.31 - Captura de pantalla de la interfície gràfica durant la demostració 4

Conclusions

I. Introducció

Aquest capítol ofereix una visió general dels resultats i conclusions obtinguts durant la realització d'aquest projecte de recerca.

En el primer apartat es repassa la recerca teòrica i l'acompliment dels primers objectius. En el segon apartat s'expliquen les millores progressives de la implementació pràctica i com, finalment, es van assolir tots els objectius del treball.

En el tercer apartat s'analitzen els resultats proporcionats pels experiments fets a l'eina interactiva. I al darrer apartat es comenten alguns aspectes que seria interessant desenvolupar en futures millores.

II. Recerca teòrica

Inicialment, es va fer recerca sobre com són, biològicament, els elements del sistema musculoesquelètic. El resultat es mostra en el capítol 1 on es descriuen els tipus de músculs, l'estructura de les fibres musculars, la composició dels ossos, els tipus d'articulació i com funciona la interacció entre els músculs i els ossos.

Tot seguit, en el capítol 2, es va aprofundir la recerca bibliogràfica sobre els models musculars. Es va veure que existien models musculars matemàtics en 1D, 2D i 3D, però que la precisió no millorava significativament del model en 2D a 3D. També es va aprendre molt sobre el model de Hill, que explica com a partir de l'activació muscular, la velocitat de contracció i la llargada normalitzada del múscul es pot

determinar les forces activa i passiva que genera un múscul. Aquesta informació va ser fonamental per implementar el model computacional del múscul.

Una vegada establerta la base biològica, es va voler explorar el món de les simulacions físiques computacionals. En el capítol 3 s'expliquen els conceptes adquirits al voltant d'aquest món. S'hi mostra què és un objecte i una força dins una simulació i com es defineixen. També s'hi explica els càlculs interns que ha de fer un motor físic i es mostra com utilitza les mateixes fórmules que la física clàssica. Finalment, també s'hi mostren trucs utilitzats per programadors per optimitzar el rendiment del motor.

Aquesta part de recerca teòrica va cobrir tant l'objectiu inicial d'aprendre sobre els models musculars i sistemes biomecànics com també l'objectiu d'aprendre sobre motors computacionals físics per simular sistemes mecànics.

III. Implementació pràctica

En la implementació pràctica, el primer pas va ser l'obtenció d'un motor computacional físic. En el capítol 4 es mostra el procés de creació d'un motor físic propi. Es va anar avançant fins al punt de calcular interaccions entre objectes, però aquest objectiu només es va aconseguir parcialment. Però com que la creació del motor físic no era un objectiu d'aquest treball de recerca es va decidir utilitzar una llibreria externa que implementava un motor físic més avançat.

En el capítol 5 s'explica com, abans de programar el model muscular, es van haver de crear els elements per modelar un sistema esquelètic. Es van crear les classes pels objectes *Bone* i *Weight*, i a més a més unions de tipus *Hinge* i *Glue* que modelen les articulacions sinovials i fibroses respectivament. Tots els elements necessaris per modelar un esquelet articulat.

En aquest punt es va poder començar a treballar en la classe *Muscle* que aplicaria forces a l'esquelet. A poc a poc, es van anar incorporant els elements del model de Hill; com les fórmules de Hill obtingudes a partir de la interpolació dels valors

empírics. A mesura que era necessari es va anar augmentant la complexitat del model; com la incorporació d'extensions/politges als extrems del múscul per modelar els tendons que presenten un canvi de direcció.

En aquest punt s'havien creat tots els elements necessaris per simular un model musculoesquelètic. A més el model del múscul es contreia segons el model de Hill generant forces equivalents a les d'un múscul biològic. Però per facilitar la utilització d'aquest *framework* es va encapsular en una classe abstracta *MuscularModel* que facilita molt la creació d'altres models musculoesquelètics.

Finalment, com s'explica en el capítol 6, es va crear una eina interactiva que serveix d'àrea de treball on es pot observar, interactuar i analitzar la simulació física. Aquesta eina té tres finestres: una que mostra la simulació, una que permet controlar el model i una tercera que mostra variables internes dels músculs en temps real.

IV. Resultats

A l'inici d'aquest projecte es van definir, a més dels objectius teòrics ja comentats, diferents objectius pràctics.

El primer era programar un model computacional del múscul. Això es va aconseguir amb la classe *Muscle* que utilitza el model de Hill per modelar el funcionament d'un múscul biològic.

El següent objectiu era programar un *framework* que permetés simular sistemes musculoesquelètics. Això es va aconseguir mitjançant la creació d'altres classes (*Bone, Weight, Nail, Hinge* i *Glue*) que modelaven la resta del sistema i d'una classe abstracta *MuscularModel* que encapsulava el *framework*.

El tercer objectiu era programar una eina interactiva que permetés observar en detall la simulació. Aquest objectiu es va complir amb la programació de la demostració. A més de la simulació gràfica, es va incloure un panell de control que dona ordres a la

simulació i un panell gràfic que mostra mitjançant corbes en temps real el que està passant exactament en la simulació.

Finalment, a partir d'aquesta eina interactiva es van dur a terme quatre experiments. En aquests experiments es va observar el comportament intern dels músculs computacionals en diferents situacions. El seu objectiu era comparar els resultats simulats amb els comportaments típics de músculs reals. Els resultats de la comparació van ser força positius, de fet, millors del previst; i es pot concloure que la simulació del model era força realista.

Tot el codi del *framework* desenvolupat i del programa de demostració, poden consultar-se a l'annex C o descarregar-se al GitHub sota la llicència de codi obert, *GNU General Public License v3.0*.

V. Propostes de millora

Com acostuma a succeir en els treballs pràctics, després de finalitzar-lo s'han après moltes lliçons. Hi ha moltes coses que si es tornés a començar es farien d'una manera diferent. I a més a més, en cas d'haver tingut més temps, hi ha àrees del projecte que s'haurien ampliat. Aquest treball no ha estat una excepció, i hi ha diversos aspectes en què seria interessant aprofundir i que podrien ser futures línies de millora.

Una d'aquestes millores seria la incorporació de l'angle de pennació al model muscular. Com s'explica al capítol 2.2.2. l'angle de les fibres musculars és un factor que pot variar la força i la velocitat de contracció del múscul, com un pseudo sistema de marxes. Seria interessant experimentar el seu efecte en la contracció d'un múscul simulat.

Una altra, seria aprofundir en la complexitat mecànica del model ossi incorporant-hi deformacions elàstiques i plàstiques i, a més, punts de fractura. Aquesta millora permetria simular com forces massa grans poden deformar i fins i tot trencar els

ossos. La incorporació d'aquesta millora també podria ser útil en les simulacions dels tendons.

Finalment, una de les principals dificultats que s'ha trobat en crear els models musculoesquelètics 'anatòmicament realistes' és l'obtenció de mides exactes dels músculs, ja que els estudis científics acostumen a publicar les mides totals dels músculs i tendons sense diferenciar-los. A més, com que els músculs, a diferència dels ossos, són elements de llargada dinàmica, sovint no queda clar si la mida proporcionada és la del múscul en contracció o en relaxació.

Per això, una futura millora seria canviar la manera de definir les mides dels músculs en el constructor. Per exemple, fent que el model calculi la llargada total del múscul i tendons a partir de la distància geomètrica entre els punts d'ancoratge. I, potser, indicant-li percentualment la proporció respecte del total que representen els tendons. En qualsevol cas, seria interessant esquivar la dificultat d'obtenir la llargada exacta dels músculs

Referències bibliogràfiques

[Atiquzzaman, M., & Srimani, PK. 2000]
Parallel computing on clusters of workstations. 2000. (doi: 10.1016/S0167-8191(99)00101-5)

[Azizi, E., et al. 2007]
Variable gearing in pennate muscles. 2007. (doi: 10.1073/pnas.0709212105)

[Benjamin, M., et al. 2006]
Where tendons and ligaments meet bone: attachment sites in relation to exercise and/or mechanical load. 2006. (doi:10.1111/j.1469-7580.2006.00540.x)

[Burr, DB. 2002]
The contribution of the organic matrix to bone's material properties. 2002. (doi: 10.1016/S8756-3282(02)00815-3)

[BYU. 2023]
Skeletal Muscle: Whole Muscle Physiology. 2023. (https://content.byui.edu/file/a236934c-3c60-4fe9-90aa-d343b3e3a640/1/module7/readings/muscle_twitches.html) (visitat: 08/2023)

[Cooper, S., & Eccles, JC. 1930]
The isometric responses of mammalian muscles. 1930. (PMID: 16994110)

[Danese, G., et al. 2007]
An Accelerator for Physics Simulations. 2007. (doi: 10.1109/MCSE.2007.94)

[Delp, S. 2022 (1)]
Biomechanics of Movement: Lecture 5.2. 2022. (https://youtu.be/nweJLpG-Ulk)

[Delp, S. 2022 (2)]
Biomechanics of Movement: Lecture 5.3. 2022. (https://youtu.be/MbrVkJX_b4M)

[Delp, S. 2022 (3)]
Biomechanics of Movement: Lecture 5.4. 2022. (https://youtu.be/-FSTsbr3XVQ)

[Dick, TJM., & Wakeling, JM. 2018]
Geometric models to explore mechanisms of dynamic shape change in skeletal muscle. 2018. (doi: 10.1098/rsos.172371)

[Fathima, N., et al. 2019]
Assessment of BMD and Statistical Analysis for Osteoporosis Detection. 2019. (doi:10.13005/bpj/1822)

[Giroux, CL., et al. 2008]
Stature Estimation Based on Dimensions of the Bony Pelvis and Proximal Femur. 2008. (doi:10.1111/j.1556-4029.2007.00598.x)

[Henkel, J. 2013]
Bone Regeneration Based on Tissue Engineering Conceptions. 2013. (doi:10.4248/BR201303002)

[Hill, AV. 1938]
The heat of shortening and the dynamic constants of muscle. 1938. (doi: 10.1098/rspb.1938.0050)

[Hong, E., et al. 2021]
Morphological symmetry of the radius and ulna. 2021. (doi:10.1371/journal.pone.0258232)

[Hrdlicka, A. 1898]
Study of the Normal Tibia. 1898. (www.jstor.org/stable/658462)

[Huxley, AF. 1974]
Muscle Contraction. 1974. (doi:10.1113/jphysiol.1974.sp010740)

[IBM. 2012]
Build a simple 2d physics engine for JavaScript games. 2012. (https://developer.ibm.com/tutorials/wa-build2dphysicsengine) (visitat: 03/2023)

[IBM. 2023]
What is supercomputing?. 2023. (www.ibm.com/topics/supercomputing) (visitat: 03/2023)

[Ide, Y., et al. 2015]
Anatomical examination of the fibula: digital imaging study for osseointegrated implant installation. 2015. (doi:10.1186/s40463-015-0055-9)

[Joseph, J., et al. 2017]
Tecnologia Industrial. 2017. (ISBN:978-84-486-1134-7)

[Karimi, E. 2019]
Correlation of Anthropometric Measurements of Proximal Tibia in Iranian Knees with Size of Current Tibial Implants. 2019. (PMID:31448311)

[Kellermayer, MS., et al. 1997]
Folding-Unfolding Transition in Single Titin Molecules Characterized with Laser Tweezers. 1997. (doi:10.1126/science.276.5315.1112)

[kg-m3. 2020]
Iron density. 2020. (https://kg-m3.com/material/iron) (visitat: 09/2023)

[Khan, MA., et al. 2020]
Determination of Gender from Various Measurements of the Humerus. 2020. (doi:10.7759/cureus.6598)

[Lemaire, KK., et al. 2016]
Comparison of the validity of Hill and Huxley muscle-tendon complex models using experimental data. 2016. (doi: 10.1242/jeb.128280)

[LFW. 2023]
Types of Muscle Contractions: Isotonic and Isometric. 2023. (https://courses.lumenlearning.com/suny-fitness/chapter/types-of-muscle-contractions-isotonic-and-isometric/) (visitat: 08/2023)

[LibreTexts. 2023]
Types of Muscle Contractions: Isotonic and Isometric. 2023. (https://med.libretexts.org/Bookshelves/Anatomy_and_Physiology/Anatomy_and_Physiology_(Boundless)/9%3A_Muscular_System/9.3%3A_Control_of_Muscle_Tension/9.3E%3A_Types_of_Muscle_Contractions%3A_Isotonic_and_Isometric) (visitat: 08/2023)

[Liqun, L. 2018]
Why Is the Human Brain So Efficient?. 2018. (https://.nautil.us/why-is-the-human-brain-so-efficient-237042) (visitat: 03/2023)

[Lorenz, T., & Campello, M. 2012]
Basic Biomechanics of the Musculoskeletal System: Biomechanics of Skeletal Muscle. 2012. (ISBN: 978-1-60913-335-1)

[Lubek, JE. 2017]
Common Free Vascularized Flaps: The Fibula. 2017. (doi:10.1016/B978-0-7020-6056-4.00037-X)

[Milner, GR. 2011]
Humeral and Femoral Head Diameters in Recent White American Skeletons. 2011. (doi:10.1111/j.1556-4029.2011.01953.x)

[Mortensen, J., & Merryweather, A. 2018]
Using OpenSim to Investigate the Effect of Active Muscles and Compliant Flooring on Head Injury Risk. 2018. (doi:10.1007/978-3-319-96089-0_81)

[Normani, F. 2010]
The Physics of Billiards. 2010. (www.real-world-physics-problems.com/physics-of-billiards.html) (visitat: 03/2023)

[Odell, M. 2016]
Electric Shocks and Electrocution, Clinical Effects and Pathology. 2016. (doi:10.1016/B978-0-12-800034-2.00166-X)

[OpenOregon. 2023]
Ultimate Strength of the Human Femur. 2023. (https://openoregon.pressbooks.pub/bodyphysics/chapter/stress-and-strain-on-the-body/) (visitat: 08/2023)

[Özkaya, N., & Leger, D. 2012]
Basic Biomechanics of the Musculoskeletal System: Introduction to Biomechanics: Basic Terminology and Concepts. 2012. (ISBN: 978-1-60913-335-1)

[Podgórski, M., et al. 2019]
'Superior biceps aponeurosis'-Morphological characteristics of the origin of the short head of the biceps brachii muscle. 2019. (doi:10.1016/j.aanat.2019.01.014)

[Ross, SA., et al. 2018]
A modelling approach for exploring muscle dynamics during cyclic contractions. 2018. (doi:10.1371/journal.pcbi.1006123)

[Rüegg, JC. 1987]
Excitation-Contraction Coupling in Fast- and Slow-Twitch Muscle Fibers. 1987. (doi:10.1055/s-2008-1025686)

[Schweitzer, R., et al. 2010]
Connecting muscles to tendons: tendons and musculoskeletal development in flies and vertebrates. 2010. (doi: 10.1242/dev.047498)

[Shiffman, D. 2012]
The Nature of Code: Simulating Natural Systems with Processing. 2012. (ISBN: 9780985930806)

[Shrestha, R, et al. 2019]
Methods Of Estimation Of Time Since Death. 2019. (PMID: 31751057)

[SLH. 2007]
Muscle Performance. 2007. (www.sciencelearn.org.nz/resources/1916-muscle-performance) (visitat: 06/2023)

[Souto, N. 2016]
Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects. 2016. (https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects) (visitat: 03/2023)

[Stromer, M.H. 1998]
The cytoskeleton in skeletal, cardiac and smooth muscle cells. 1998. (doi: 10.14670/HH-13.283)

[Terzidis, I., et al. 2012]
Gender and Side-to-Side Differences of Femoral Condyles Morphology: Osteometric Data from 360 Caucasian Dried Femori. 2012. (doi: 10.1155/2012/679658)

[TMA. 2023]
Soleus. 2023. (www.teachmeanatomy.info/encyclopaedia/s/soleus/) (visitat: 08/2023)

[UH. 2018 (1)]
Fibrous Joints. 2018. (https://pressbooks-dev.oer.hawaii.edu/anatomyandphysiology/chapter/fibrous-joints/) (visitat: 05/2023)

[UH. 2018 (2)]
Cartilaginous Joints. 2018. (https://pressbooks-dev.oer.hawaii.edu/anatomyandphysiology/chapter/cartilaginous-joints/) (visitat: 05/2023)

[UH. 2018 (3)]
Synovial Joints. 2018. (https://pressbooks-dev.oer.hawaii.edu/anatomyandphysiology/chapter/synovial-joints/) (visitat: 05/2023)

[UPMC. 2023]
Tendon and Ligament Tears, Ruptures, and Injuries. 2023. (www.upmc.com/services/orthopaedics/conditions-treatments/tendon-tears-ruptures) (visitat: 08/2023)

[Vilimek, M. 2007]
Musculotendon forces derived by different muscle models. 2007. (PMID: 18421942)

[Wik, TS. 2012]
Experimental evaluation of new concepts in hip arthroplasty. 2012. (doi: 10.3109/17453674.2012.678804)

[Wisdom, K., et al. 2014]
Use it or lose it: multiscale skeletal muscle adaptation to mechanical stimuli. 2014. (doi:10.1007/s10237-014-0607-3)

Annexos

Annex A: Codi motor físic

Motor físic propi v0.4:

```
ArrayList<Circle> circles;
//Definir variables globals de la simulació
float GLOBAL_GRAVITY             = 9.81;
float GLOBAL_FRICTION            = 0.05;
float GLOBAL_BOUNDARY_ELESTICITY = 0.9;
float SPACE_WIDTH                = 2.44;// meter
float SPACE_HEIGHT               = 1.22;// meter
float SPACE_SCALE                = 1.0; // pixels/meter
float TIME_SCALE                 = 0.5; // simulationSecond/irlSecond

//Crear la finestra on és dibuixara la simulació
void settings() {
  float scaleWidth=(displayWidth-50)/SPACE_WIDTH;
  float scaleHeight=(displayHeight-100)/SPACE_HEIGHT;

  if (scaleWidth < scaleHeight) {
    SPACE_SCALE = scaleWidth;
  } else {
    SPACE_SCALE = scaleHeight;
  }

  size(int(SPACE_WIDTH*SPACE_SCALE), int(SPACE_HEIGHT*SPACE_SCALE));
}

void setup() {
  frameRate(30);

  circles = new ArrayList<Circle>();

  //Crear 20 cercles amb mides i color aleatori + afegir una força en una
  //direcció aleatoria
  for (int i = 0; i < 20; i++) {
    Circle newCircle = new Circle(SPACE_WIDTH/2, SPACE_HEIGHT/2, random(0.01,
                                  0.1), color(random(255), random(255),
                                  random(255))); //metric
    newCircle.setMass(0.16*random(0.5, 1.5));

    newCircle.addForce(random(2, 70), random(200,340));

    newCircle.setRestitution(random(0.5, 1));
    circles.add(newCircle);
  }
```

```
}

void draw() {
  //Dibuixar el fons
  background(#F4F5C8);
  noFill();
  stroke(0);
  rect(0, 0, SPACE_WIDTH*SPACE_SCALE, SPACE_HEIGHT*SPACE_SCALE);

  //Actualitzar les propietats de cada cercle
  for (int i = 0; i<circles.size(); i++) {
    //Afeguir la força de la gravetat al cercle
    circles.get(i).addForce(circles.get(i).getMass()*GLOBAL_GRAVITY, 90);
    //Afeguir la froça de fregament al cercle

circles.get(i).addForce(circles.get(i).getVelocityMagnitude()*GLOBAL_FRICTION,
                        circles.get(i).getVelocityAngle()+180);
    //Actualitzar la nova posició del cercle
    circles.get(i).update(TIME_SCALE/frameRate);
    //Esborar les forçes que s'han aplicat al cercle
    circles.get(i).resetTotalForces();
  }

  //Comprobar que cap cercle s'ha sortit de la finestra
  checkBoundaries();

  //Dibuixar tots els cercles
  for (int i = 0; i<circles.size(); i++) {
    circles.get(i).render();
  }
}

//Comprobar que cap cercle s'ha sortit de la finestra
void checkBoundaries() {
  for (int i = 0; i<circles.size(); i++) {
    Circle c = circles.get(i);
    //Comprobar que no s'ha sortit de la part soperior de la finestra
    if (c.getBoxTop()<0) {
      c.setPosY(c.getRadius());
      c.setVelocityAngle(0-c.getVelocityAngle());
      c.setVelocityMagnitude(c.getVelocityMagnitude()*min(c.getRestitution(),
                             GLOBAL_BOUNDARY_ELESTICITY));
    }
    //Comprobar que no s'ha sortit de la part inferior de la finestra
    if (c.getBoxBottom()>SPACE_HEIGHT) {
      c.setPosY(SPACE_HEIGHT-c.getRadius());
      c.setVelocityAngle(0-c.getVelocityAngle());
      c.setVelocityMagnitude(c.getVelocityMagnitude()*min(c.getRestitution(),
```

```
                            GLOBAL_BOUNDARY_ELESTICITY));
    }
    //Comprobar que no s'ha sortit de la dreta de la finestra
    if (c.getBoxRight()>SPACE_WIDTH) {
      c.setPosX(SPACE_WIDTH-c.getRadius());
      c.setVelocityAngle(180-c.getVelocityAngle());
      c.setVelocityMagnitude(c.getVelocityMagnitude()*min(c.getRestitution(),
                            GLOBAL_BOUNDARY_ELESTICITY));
    }
    //Comprobar que no s'ha sortit de l'esquerra de la finestra
    if (c.getBoxLeft()<0) {
      c.setPosX(c.getRadius());
      c.setVelocityAngle(180-c.getVelocityAngle());
      c.setVelocityMagnitude(c.getVelocityMagnitude()*min(c.getRestitution(),
                            GLOBAL_BOUNDARY_ELESTICITY));
    }
  }
}

class Circle {
  //Geometrical Properties
  float posX;    //m
  float posY;    //m
  float radius;  //m
  color colour;

  //Dynamical Properties
  float velAngle;      //degrees
  float velMagnitude;  //m/s
  float accAngle;      //degrees
  float accMagnitude;  //m/s^2

  //Physical Properties
  float restitution;    //%/100
  float mass;                     //kg
  float density;                  //kg/m2
  float totalForcesAngle;         //degrees
  float totalForcesMagnitude = 0;  //N=kg*m/s^2

  //constructor
  Circle(float x, float y, float rad, color col){
    posX=x;
    posY=y;
    radius=rad;
    colour=col;
  }

  //Velocity Functions
```

```
void setCartesianVelocity(float vX, float vY){
  velMagnitude=getMagnitudeFromCartesian(vX, vY);
  velAngle=getAngleFromCartesian(vX, vY);
}

void setPolarVelocity(float mag, float fita){
  velMagnitude=mag;
  velAngle=fita;
}

//Acceleration Functions
void setCartesianAcceleration(float aX, float aY){
  accMagnitude=getMagnitudeFromCartesian(aX, aY);
  accAngle=getAngleFromCartesian(aX, aY);
}

void setPolarAcceleration(float mag, float fita){
  accMagnitude=mag;
  accAngle=fita;
}

//Forces Functions
void setTotalForces(float mag, float fita){
  totalForcesAngle = fita;
  totalForcesMagnitude = mag;
}

void resetTotalForces(){
  totalForcesAngle=0;
  totalForcesMagnitude=0;
}

//Afeguir una força al total de forçes
void addForce(float mag, float fita){
  //Convertir el total de forçes de polar a cartesià
  float x1 = totalForcesMagnitude*cos(radians(totalForcesAngle));
  float y1 = totalForcesMagnitude*sin(radians(totalForcesAngle));

  //Convertir al força que s'ha d'afeguir de polar a cartesià
  float x2 = mag*cos(radians(fita));
  float y2 = mag*sin(radians(fita));

  //Sumar les forçes cartesianes
  float newX = x1 + x2;
  float newY = y1 + y2;

  //Convertir el nou total de forçes de cartesianes a polars
  totalForcesMagnitude = getMagnitudeFromCartesian(newX, newY);
```

```
  totalForcesAngle = getAngleFromCartesian(newX, newY);
}

//Dibuixar el cercle
void render(){
  noStroke();
  fill(colour);
  ellipseMode(RADIUS);
  ellipse(posX*SPACE_SCALE, posY*SPACE_SCALE, radius*SPACE_SCALE,
          radius*SPACE_SCALE);
}

void update(float timeIncrement){
  //Calcular l'acceleració del cercle
  accMagnitude = totalForcesMagnitude / mass;
  accAngle     = totalForcesAngle;

  float accX = accMagnitude*cos(radians(accAngle));
  float accY = accMagnitude*sin(radians(accAngle));
  float velX = velMagnitude*cos(radians(velAngle));
  float velY = velMagnitude*sin(radians(velAngle));

  //Calcular la velocitat nova del cercle
  velX += accX*timeIncrement;
  velY += accY*timeIncrement;

  //Guardar la velocitat nova pel següent frame
  setCartesianVelocity(velX, velY);

  //Calcular la posició nova del cercle
  posX += velX*timeIncrement;
  posY += velY*timeIncrement;
}

//========================================== Geometry
float getPosX(){
  return(posX);
}
void setPosX(float pX){
  posX=pX;
}

float getPosY(){
  return(posY);
}
void setPosY(float pY){
  posY=pY;
}
```

```
void move(float mag, float fita)
{
  println("X="+posX);
  println("Y="+posY);
  posX += cos(fita) * mag;
  posY += sin(fita) * mag;
  println("X="+posX);
  println("Y="+posY+"\n");
}

float getRadius(){
  return(radius);
}

float getArea(){
  return(PI*sq(radius));
}

float getPerimeter(){
  return(2*PI*radius);
}

float getBoxTop(){
  return(posY-radius);
}

float getBoxBottom(){
  return(posY+radius);
}

float getBoxRight(){
  return(posX+radius);
}

float getBoxLeft(){
  return(posX-radius);
}

//================================== Dynamics
float getVelocityAngle(){
  return velAngle;
}

void setVelocityAngle(float fita){
  velAngle=fita;
}
```

```
float getVelocityMagnitude(){
  return velMagnitude;
}

void setVelocityMagnitude(float mag){
  velMagnitude=mag;
}

float getAccelerationAngleX(){
  return(accAngle);
}

void setAccelerationAngle(float fita){
  accAngle=fita;
}

float getAccelerationMagnitude(){
  return(accMagnitude);
}

void setAccelerationMagnitude(float mag){
  accMagnitude=mag;
}

float getMomentum(){
  return(mass * velMagnitude);
}

void setMomentum(float momentum){
  velMagnitude=(momentum/mass);
}

//================================== Physics
float getRestitution(){
  return(restitution);
}
void setRestitution(float e){
  restitution=e;
}

//================================== Newtonian
float getMass(){
  return(mass);
}

void setMass(float m){
  mass=m;
  density=m/getArea();
```

```
}

float getDensity(){
  return(density);
}

void setDensity(float d){
  density=d;
  mass=getArea()*d;
}

float getTotalForcesMagnitude(){
  return(totalForcesMagnitude);
}

void setTotalForcesMagnitude(float mag){
  totalForcesMagnitude=mag;
}

float getTotalForcesAngle(){
  return(totalForcesAngle);
}

void setTotalForcesAngle(float fita){
  totalForcesAngle=fita;
}

//============================================ Auxiliar Functions
float getAngleFromCartesian(float x, float y){
  float fita;
  if (x != 0){
    fita = degrees(atan(y/x));
    if (x<0){
      fita += 180;
    }
    return (fita);
  }
  else {
    if (y < 0){
      fita=-90;
    }
    else if (y > 0){
      fita=90;
    }
    else{
     fita = 0;
    }
    return (fita);
```

```
    }
  }

  float getMagnitudeFromCartesian(float x, float y){
    return(sqrt(sq(x)+sq(y)));
  }

  float getAngleFromShapes(Circle a,Circle b)
  {
    return(getAngleFromCartesian(b.getPosX()-a.getPosX(),
                                 b.getPosY()-a.getPosY()));
  }

  float getMagnitudeFromShapes(Circle a,Circle b)
  {
    return(getMagnitudeFromCartesian(b.getPosX()-a.getPosX(),
                                     b.getPosY()-a.getPosY()));
  }
}
```

Annex B: Funcions de Hill

Annex B.1: Funció Llargada - Força Activa

Llargada N	Força Activa
0,3422	0,0031
0,3815	0,0031
0,4364	0,0031
0,4859	0,0173
0,5165	0,0418
0,5424	0,0724
0,5667	0,1131
0,5848	0,1498
0,6052	0,2008
0,6224	0,2518
0,6327	0,2864
0,6452	0,3313
0,6601	0,3884
0,6766	0,4536
0,6931	0,5148
0,7057	0,5657
0,7214	0,6228
0,7386	0,6779
0,7543	0,7288
0,7834	0,8033
0,8077	0,8552
0,8281	0,8899
0,8626	0,9348
0,8948	0,9653
0,9388	0,9888
0,9780	0,9990
1,0110	0,9980

Llargada N	Força Activa
1,0283	0,9796
1,0432	0,9582
1,0612	0,9286
1,0793	0,8960
1,0989	0,8593
1,1217	0,8145
1,1405	0,7757
1,1766	0,7023
1,2002	0,6555
1,2190	0,6167
1,2527	0,5474
1,2841	0,4862
1,3077	0,4414
1,3359	0,3884
1,3776	0,3129
1,4050	0,2661
1,4317	0,2232
1,4560	0,1865
1,4765	0,1580
1,5047	0,1213
1,5322	0,0866
1,5557	0,0642
1,5808	0,0408
1,6107	0,0204
1,6405	0,0071
1,6656	0,0031
1,7033	0,0020

Annex B.2: Funció Velocitat - Força Activa

Velocitat N	Força Activa
-1,0025	0,0055
-0,8984	0,0208
-0,8240	0,0374
-0,7546	0,0568
-0,6952	0,0748
-0,6233	0,1025
-0,5589	0,1302
-0,4944	0,1620
-0,4325	0,1981
-0,3680	0,2424
-0,3185	0,2825
-0,2714	0,3283
-0,2268	0,3767
-0,1884	0,4266
-0,1314	0,5180
-0,0917	0,6066
-0,0620	0,6870
-0,0397	0,7673
-0,0260	0,8366
-0,0099	0,9197
-0,0012	0,9972

Velocitat N	Força Activa
0,0062	1,0859
0,0112	1,1773
0,0136	1,2465
0,0273	1,3158
0,0570	1,3850
0,0979	1,4349
0,1400	1,4681
0,1995	1,5014
0,2689	1,5277
0,3309	1,5457
0,3854	1,5568
0,4424	1,5679
0,5068	1,5748
0,5638	1,5817
0,6406	1,5873
0,7051	1,5928
0,7695	1,5942
0,8463	1,5970
0,9108	1,5997
0,9901	1,5997
1,0818	1,5983

Annex B.3: Funció Llargada - Força Passiva

Llargada N	Força Passiva
0,4106	-0,0010
0,4718	-0,0010
0,5408	-0,0010
0,6569	-0,0010
0,7792	0,0000
0,8475	0,0041
0,8906	0,0081
0,9337	0,0122
0,9643	0,0183
0,9996	0,0234
1,0302	0,0316
1,0631	0,0407
1,1039	0,0540
1,1400	0,0713
1,1761	0,0916
1,2075	0,1141
1,2420	0,1446
1,2718	0,1792
1,3016	0,2200
1,3220	0,2536
1,3424	0,2912

Llargada N	Força Passiva
1,3596	0,3289
1,3776	0,3697
1,3957	0,4145
1,4137	0,4613
1,4333	0,5143
1,4514	0,5652
1,4686	0,6141
1,4851	0,6609
1,5063	0,7200
1,5220	0,7648
1,5447	0,8299
1,5643	0,8809
1,5863	0,9369
1,6020	0,9807
1,6247	1,0377
1,6490	1,0988
1,6663	1,1395
1,6843	1,1843
1,7008	1,2230
1,7188	1,2637

Annex C: Codi *Framework*

Annex C.1: Classe MuscularModel

```
// ============================================
// CLASS MUSCULAR MODEL (from the Muscular Model Framework)
// by Castanyer (Updated: 15/8/2023)
// For Research Project
// ============================================
// This is the main class for the Muscular Model Framework
// Any specific muscle-esqueletal model is a subclass of this one.
// It handle all the shared functions (control, update and display)
// except the definitions for the bones, joints, muscles and weights.
// ============================================
//Box2D libraries
import shiffman.box2d.*;
import org.jbox2d.common.*;
import org.jbox2d.dynamics.joints.*;
import org.jbox2d.collision.shapes.*;
import org.jbox2d.collision.shapes.Shape;
import org.jbox2d.common.*;
import org.jbox2d.dynamics.*;
import org.jbox2d.dynamics.contacts.*;

Box2DProcessing box2d;

abstract class MuscularModel
{
  color COLOR_GRID = color(129, 110, 59);

  static final int SIMULATION_LEVEL_NONE = 0;
  static final int SIMULATION_LEVEL_BONES = 1;
  static final int SIMULATION_LEVEL_JOINTS = 2;
  static final int SIMULATION_LEVEL_MUSCLES = 3;
  static final int SIMULATION_LEVEL_WEIGHTS = 4;

  boolean DEBUG_MODE = false;

  ArrayList<Nail> allNails = new ArrayList<Nail>();
  ArrayList<Bone> allBones = new ArrayList<Bone>();
  ArrayList<Muscle> allMuscles = new ArrayList<Muscle>();
  ArrayList<Hinge> allHinges = new ArrayList<Hinge>();
  ArrayList<Glue> allGlues = new ArrayList<Glue>();
  ArrayList<Weight> allWeights = new ArrayList<Weight>();
  ArrayList<int[]> allMuscleGroups = new ArrayList<int[]>();
  ArrayList<String> variantNames = new ArrayList<String>();
```

```
ArrayList<String> weightNames = new ArrayList<String>();

int currentSimulationLevel = SIMULATION_LEVEL_WEIGHTS;
int currentModelVariant = 0;
boolean isSimulationRunning = false;
float gridDivisionsNumber = 10;
float gridDivisionsLength = 0.1;
float displayScale = 100;
float defaultDisplayScale = 100;
int FRAME_RATE = 60;


// CONSTRUCTOR
MuscularModel(PApplet p, float x, float y, float scale, int fps)
{
  this(p, x, y, scale, fps, SIMULATION_LEVEL_WEIGHTS, 0, 0);
}
MuscularModel(PApplet p, float x, float y, float scale, int fps, int level)
{
  this(p, x, y, scale, fps, level, 0, 0);
}
MuscularModel(PApplet p, float x, float y, float scale, int fps, int level,
              int variantIndex, int weightIndex)
{
  box2d.setGravity(0,-9.8);

  println("MuscularModel( "+x+", "+y+", "+scale+", "+level+", "
          +variantIndex+")");
  this.currentSimulationLevel = level;
  this.currentModelVariant = variantIndex;
  this.defaultDisplayScale = scale;
  this.displayScale = defaultDisplayScale;
  this.FRAME_RATE = fps;

  variantNames.clear();
  weightNames.clear();
  this.defineMain();

  if (currentSimulationLevel>=SIMULATION_LEVEL_BONES)
  {
    this.defineBones(x, y, variantIndex);
  }

  if (currentSimulationLevel>=SIMULATION_LEVEL_JOINTS)
  {
    this.defineJoints(x, y, variantIndex);
  }
```

```
  if (currentSimulationLevel>=SIMULATION_LEVEL_MUSCLES)
  {
    this.defineMuscles(x, y, variantIndex);

    if (this.allMuscleGroups.size()==0)
    {
      this.initializeWithSingleMuscleGroups();
    }
  }

  if (currentSimulationLevel>=SIMULATION_LEVEL_WEIGHTS)
  {
    this.defineWeights(x, y, variantIndex, weightIndex);
  }
}

void setGridProperties(float divisionLength, int numberOfQuadrantDivisions)
{
    this.gridDivisionsNumber = numberOfQuadrantDivisions;
    this.gridDivisionsLength = divisionLength;
}

// DEFINITIONS OF ITEMS: ABSTRACT METHODS (To be defined by the derived class)
abstract void defineMain();
abstract void defineBones(float x,float y, int variantIndex);
abstract void defineJoints(float x,float y, int variantIndex);
abstract void defineMuscles(float x,float y, int variantIndex);
abstract void defineWeights(float x,float y, int variantIndex, int
                            weightIndex);

// MODEL COMMANDS ==================================================
void playSimulation()
{
  isSimulationRunning=true;
}

void stopSimulation()
{
  isSimulationRunning=false;
}

void togglePauseSimulation()
{
  isSimulationRunning = !isSimulationRunning;
}

void setGravity(Vec2 gravity)
{
```

```
  box2d.setGravity(gravity.x, gravity.y);
}

void setMuscleActivation(int muscleIndex, float activationLevel)
{
  if (muscleIndex>=0 && muscleIndex<allMuscles.size())
  {
    allMuscles.get(muscleIndex).setActivation(activationLevel);
  }
}
void setMuscleActivation(String muscleName, float activationLevel)
{
  Muscle muscle = getMuscleByName(muscleName);
  if (muscle!=null) { muscle.setActivation(activationLevel); }
}

void setMuscleGroupActivation(int groupIndex, float activationLevel)
{
  if (groupIndex>=0 && groupIndex<allMuscleGroups.size())
  {
    int[] muscleIndexes = model.allMuscleGroups.get(groupIndex);
    for (int i = 0; i < muscleIndexes.length; i++)
    {
      model.setMuscleActivation(muscleIndexes[i], activationLevel);
    }
  }
}

float getMuscleGroupActivation(int groupIndex)
{
  if (groupIndex>=0 && groupIndex<allMuscleGroups.size())
  {
    int[] muscleIndexes = model.allMuscleGroups.get(groupIndex);
    float totalActivation = 0.0;
    for (int i = 0; i < muscleIndexes.length; i++)
    {
      totalActivation = totalActivation +
                        model.allMuscles.get(muscleIndexes[i]).activation;
    }
    return (totalActivation/float(muscleIndexes.length));
  }
  return(0.0);
}

// MUSCLE TENSION ======================================================
void setMuscleTension(int muscleIndex, float tension)
{
  if (muscleIndex>=0 && muscleIndex<allMuscles.size())
```

```
  {
    allMuscles.get(muscleIndex).setTension(tension);
  }
}

void setMuscleTension(String muscleName, float tension)
{
  Muscle muscle = getMuscleByName(muscleName);
  if (muscle!=null) { muscle.setTension(tension); }
}

void setMuscleGroupTension(int groupIndex, float tension)
{
  if (groupIndex>=0 && groupIndex<allMuscleGroups.size())
  {
    int[] muscleIndexes = model.allMuscleGroups.get(groupIndex);
    for (int i = 0; i < muscleIndexes.length; i++)
    {
      model.setMuscleTension(muscleIndexes[i], tension);
    }
  }
}

float getMuscleGroupTension(int groupIndex)
{
  if (groupIndex>=0 && groupIndex<allMuscleGroups.size())
  {
    int[] muscleIndexes = model.allMuscleGroups.get(groupIndex);
    float totalTension = 0.0;
    for (int i = 0; i < muscleIndexes.length; i++)
    {
      totalTension = totalTension +
                     model.allMuscles.get(muscleIndexes[i]).tensionGoal;
    }
    return (totalTension/float(muscleIndexes.length));
  }
  return(0.0);
}

// MUSCLE GROUP FUNCTIONS ================================================
int createMuscleGroup(String[] muscleNames )
{
  if (allMuscleGroups.size()==0)
  {
    initializeWithSingleMuscleGroups();
  }

  IntList muscleIndexes = new IntList();
```

```
  //Get the index for each muscle in the list
  for (int i=0; i<muscleNames.length; i++)
  {
    int muscleIndex = getMuscleIndexByName(muscleNames[i]);
    if (muscleIndex!=-1)
    {
      muscleIndexes.append(muscleIndex);
    }
    else if (DEBUG_MODE)
    {
      println("Warning: in muscle set definition: unknown name '"
              +muscleNames[i]+"'");
    }
  }
  allMuscleGroups.add(0,muscleIndexes.toArray());
  return(allMuscleGroups.size()-1);
}

int initializeWithSingleMuscleGroups()
{
  allMuscleGroups.clear();
  for (int i=0; i<allMuscles.size(); i++)
  {
    allMuscleGroups.add(new int[] {i});
  }
  return(allMuscleGroups.size()-1);
}

String getMuscleGroupName(int groupIndex)
{
  String muscleName = "";
  String groupName = "";
  int[] muscleGroupIndexes = allMuscleGroups.get(groupIndex);

  for (int muscleIndex=0; muscleIndex < muscleGroupIndexes.length;
       muscleIndex++)
  {
    muscleName = allMuscles.get(muscleGroupIndexes[muscleIndex]).name;
    groupName = groupName + " + " + muscleName;
  }

  if (groupName.substring(0,3).equals(" + "))
  {
    groupName = groupName.substring(3);
  }

  return(groupName);
```

```
}


// ELEMENTS GETTERS BY NAME ================================================
Nail getNailByName(String name)
{
  for (int i = 0; i<allNails.size(); i++)
  {
    String candidateName = allNails.get(i).name;
    if (name==candidateName) { return (allNails.get(i)); }
  }
  return(null);
}

Bone getBoneByName(String name)
{
  for (int i = 0; i<allBones.size(); i++)
  {
    String candidateName = allBones.get(i).name;
    if (name==candidateName) { return (allBones.get(i)); }
  }
  return(null);
}

Muscle getMuscleByName(String name)
{
  return(getMuscleByName(name, false));
}

Muscle getMuscleByName(String name, boolean mute)
{
  for (int i = 0; i<allMuscles.size(); i++)
  {
    String candidateName = allMuscles.get(i).name;
    if (name==candidateName) { return (allMuscles.get(i)); }
  }
  if (!mute) {println("ERROR: getMuscleByName(): Unknown muscle '"+name+"'");}
  return(null);
}

int getMuscleIndexByName(String name)
{
  for (int i = 0; i<allMuscles.size(); i++)
  {
    String candidateName = allMuscles.get(i).name;
    if (name==candidateName) { return (i); }
  }
  return(-1);
```

```
}

Hinge getHingeByName(String name)
{
  for (int i = 0; i<allHinges.size(); i++)
  {
    String candidateName = allHinges.get(i).name;
    if (name==candidateName) { return (allHinges.get(i)); }
  }
  return(null);
}

Glue getGlueByName(String name)
{
  for (int i = 0; i<allGlues.size(); i++)
  {
    String candidateName = allBones.get(i).name;
    if (name==candidateName) { return (allGlues.get(i)); }
  }
  return(null);
}

Weight getWeightByName(String name)
{
  for (int i = 0; i<allWeights.size(); i++)
  {
    String candidateName = allWeights.get(i).name;
    if (name==candidateName) { return (allWeights.get(i)); }
  }
  return(null);
}
// STEP METHOD TO SIMULATE MUSCLE
void step()
{
  // Update World Box2d (Bones, Joints & Weights)
  int subSampling = 10;
  float timeIncrement = 1.0/(FRAME_RATE*subSampling);

  for (int i=0; i<subSampling; i++)
  {
    box2d.step(timeIncrement, 8, 3);

    // Update each muscle object
    for (int j = 0; j<allMuscles.size(); j++)
    {
      Muscle muscle = allMuscles.get(j);
      muscle.step(timeIncrement);
    }
```

```
  }
}

// DISPLAY METHOD TO DISPLAY ALL THE MODEL ELEMENTS
void drawGrid()
{
  float gridLength = gridDivisionsNumber*gridDivisionsLength;

  Vec2 pixCenter = new Vec2(box2d.coordWorldToPixels(0, 0));
  Vec2 pixWest = new Vec2(box2d.coordWorldToPixels(-gridLength*displayScale,
                          0));
  Vec2 pixEast = new Vec2(box2d.coordWorldToPixels(+gridLength*displayScale,
                          00));
  Vec2 pixNorth = new Vec2(box2d.coordWorldToPixels(0, + gridLength *
                           displayScale));
  Vec2 pixSouth = new Vec2(box2d.coordWorldToPixels(0, -
                           gridLength*displayScale));

  stroke(COLOR_GRID);
  strokeWeight(1);

  for (float x=-gridLength; x<+gridLength+gridDivisionsLength;
       x=x+gridDivisionsLength)
  {
    Vec2 vStart = new Vec2(box2d.coordWorldToPixels(x * displayScale, +
                           gridLength * displayScale));
    Vec2 vEnd = new Vec2(box2d.coordWorldToPixels(x*displayScale, - gridLength
                         * displayScale));
    Vec2 hStart = new Vec2(box2d.coordWorldToPixels(-gridLength*displayScale,
                           x * displayScale));
    Vec2 hEnd = new Vec2(box2d.coordWorldToPixels(+gridLength*displayScale, x
                         * displayScale));

    line(vStart.x, vStart.y, vEnd.x, vEnd.y);
    line(hStart.x, hStart.y, hEnd.x, hEnd.y);
  }

  stroke(128);
  strokeWeight(3);
  line(pixWest.x, pixCenter.y, pixEast.x, pixCenter.y);
  line(pixCenter.x, pixNorth.y, pixCenter.x, pixSouth.y);

  strokeWeight(1);
}

void incrementDisplayScale(float increment)
{
  displayScale += increment;
```

```
  if (displayScale < 1){
    displayScale = 1;
  }
}

void resetDisplayScale()
{
  displayScale = defaultDisplayScale;
}

void display()
{
  // Axis & Grid
  if (gridDivisionsNumber>0) { drawGrid(); }

  if (currentSimulationLevel>=SIMULATION_LEVEL_BONES)
  {
    // Display Bones
    for (int i = 0; i<allBones.size(); i++)
    {
      Bone bone = allBones.get(i);
      if (bone!=null)    { bone.display(displayScale); }
    }
  }
  if (currentSimulationLevel>=SIMULATION_LEVEL_JOINTS)
  {
    // Display Nails
    for (int i = 0; i<allNails.size(); i++)
    {
      Nail nail = allNails.get(i);
      if (nail!=null)    { nail.display(displayScale); }
    }

    // Display Hinges
    for (int i = 0; i<allHinges.size(); i++)
    {
      Hinge hinge = allHinges.get(i);
      if (hinge!=null)    { hinge.display(displayScale); }
    }

    // Display Glues
    for (int i = 0; i<allGlues.size(); i++)
    {
      Glue glue = allGlues.get(i);
      if (glue!=null)    { glue.display(displayScale); }
    }
  }
```

```
    if (currentSimulationLevel>=SIMULATION_LEVEL_MUSCLES)
    {
      // Display Muscles
      for (int i = 0; i<allMuscles.size(); i++)
      {
        Muscle muscle = allMuscles.get(i);
        if (muscle!=null)  muscle.display(displayScale);
      }
    }

    if (currentSimulationLevel>=SIMULATION_LEVEL_WEIGHTS)
    {
      // Display Weights
      for (int i = 0; i<allWeights.size(); i++)
      {
        Weight weight = allWeights.get(i);
        if (weight!=null) { weight.display(displayScale); }
      }
    }
  }
}
```

Annex C.2: Classe Nail

```
// ==========================================
// CLASS NAIL (from the Muscular Model Framework)
// by Castanyer (Updated: 11/7/2023)
// For Research Project
// ==========================================

class Nail
{
  color NAIL_COLOR = color(255, 255, 255);

  Body body;

  float xPos;
  float yPos;

  String name;

  Nail(float xPos_, float yPos_)
  {
    this(new String("Unknown Nail Joint #")+random(100,999), xPos_, yPos_);
  }
  Nail(String name, float xPos_, float yPos_)
  {
    this.name = name;

    xPos = xPos_;
    yPos = yPos_;

    // Create Body(Fixture(Shape))
    // Create Shape
    PolygonShape shape = new PolygonShape();
      shape.setAsBox(box2d.scalarPixelsToWorld(1/100),
                     box2d.scalarPixelsToWorld(1/100));

    // Create fixture with shape
    FixtureDef fixDef = new FixtureDef();
      fixDef.shape = shape;
      fixDef.friction = 0.9;
      fixDef.restitution = 0.5;

    // Create BodyDef
    BodyDef bodyDef = new BodyDef();
      bodyDef.type = BodyType.STATIC;
      bodyDef.position.set(xPos, yPos);

    // Attach Shape to Body through Fixture
```

```
    body = box2d.createBody(bodyDef);
      body.createFixture(fixDef);
  }
  void display(float displayScale)
  {
    Vec2 pos = body.getWorldCenter();
    Vec2 pix = new Vec2(box2d.coordWorldToPixels(pos.x*displayScale,
                        pos.y*displayScale));
    float a = body.getAngle();

    fill(NAIL_COLOR);
    stroke(0);
    ellipse(pix.x, pix.y, 5, 5);
  }
}
```

Annex C.3: Classe Bone

```
// ==========================================
// CLASS BONE (from the Muscular Model Framework)
// by Castanyer (Updated: 23/7/2023)
// For Research Project
// ==========================================
// A bone uses a Box2D body to model a phisical bone.
// It is defined by its location (x,y) and it's measurements (length and width).
// It's displayed as a white rectangl with the measurements set during the
   creation.
// ==========================================

class Bone
{
  color BONE_COLOR = color(255, 255, 240);
  float BONE_FRICTION = 8;
  float BONE_ELASTICITY = 0.5;
  float BONE_DENSITY = 1900;

  // Bone Properties
  public Body body;

  float posX;
  float posY;
  float sizeLength;
  float sizeWidth;
  float sizeDepth;

  String name;

  Bone(float posX_, float posY_, float length_, float width_)
  {
    this(new String("Unknown Bone #")+random(100,999), posX_, posY_, length_,
         width_, width_, false);
  }

  Bone(float posX_, float posY_, float length_, float width_, float depth_)
  {
    this(new String("Unknown Bone #")+random(100,999), posX_, posY_, length_,
         width_, depth_, false);
  }

  Bone(String name, float posX_, float posY_, float length_, float width_)
  {
    this(name, posX_, posY_, length_, width_, width_, false);
  }
```

```
Bone(String name, float posX_, float posY_, float length_, float width_, float
     depth_)
{
  this(name, posX_, posY_, length_, width_, depth_, false);
}
Bone(String name, float posX_, float posY_, float length_, float width_, float
     depth_, boolean isStatic)
{
  this.name = name;

  posX = posX_;
  posY = posY_;
  sizeLength = length_;
  sizeWidth = width_;
  sizeDepth = depth_;

  // Create BodyDef
  BodyDef bodyDef = new BodyDef();
    bodyDef.type = isStatic ? BodyType.STATIC : BodyType.DYNAMIC;
    bodyDef.position.set(posX, posY);
    bodyDef.linearDamping = BONE_FRICTION;
    bodyDef.angularDamping = BONE_FRICTION;
    bodyDef.bullet = true;
  body = box2d.createBody(bodyDef);

  // Create Shape
  PolygonShape shape = new PolygonShape();
    shape.setAsBox( sizeLength/2, sizeWidth/2);

 // Create fixture with shape
  FixtureDef fixDef = new FixtureDef();
    fixDef.shape = shape;
    fixDef.density = BONE_DENSITY*sizeDepth*2;  //2d density kg/m2
    fixDef.friction = BONE_FRICTION;
    fixDef.restitution = BONE_ELASTICITY;
    fixDef.isSensor = true;

  // Attach Shape to Body through Fixture
  body.createFixture(fixDef);

  println("Bone "+name+" Created: Weight = "+body.m_mass+" kg");

}

void display(float displayScale)
{
  Vec2 posCenter = body.getWorldCenter();
  Vec2 pix = new Vec2(box2d.coordWorldToPixels(posCenter.x*displayScale,
```

```
                        posCenter.y*displayScale));
  float a = body.getAngle();
  float pixLength = box2d.scalarWorldToPixels(sizeLength)*displayScale;
  float pixWidth = box2d.scalarWorldToPixels(sizeWidth)*displayScale;
  float pixCornerRadius = min(pixWidth, pixLength)/3;
  pushMatrix();
  translate(pix.x, pix.y);
  rotate(-a);

  fill(BONE_COLOR);
  stroke(0);
  rectMode(CENTER);
  rect(0, 0, pixLength, pixWidth, pixCornerRadius);

  popMatrix();
}


void printDetails(String name)
{
  println("Bone Details of '"+name+"'");
  int fixtureCounter=0;

  for (Fixture f = body.getFixtureList(); f!=null ; f = f.getNext())
  {
    fixtureCounter++;
    println("- Fixture #"+fixtureCounter+":");

    ShapeType shapeType = f.getType();
    if ( shapeType == ShapeType.POLYGON )
    {
      PolygonShape polygonShape = (PolygonShape)f.getShape();
      int vertexCount = polygonShape.getVertexCount();
      println("  - Type: POLYGON ("+vertexCount+" vertices)");

      // Relative Corners
      float relSumX = 0;
      float relSumY = 0;
      print("  - Relative Vertices: ");
      for (int v=0; v<vertexCount; v++)
      {
        Vec2 vertex = polygonShape.getVertex(v);
        print("("+(vertex.x)+","+(vertex.y)+") ");
        relSumX = relSumX+vertex.x;
        relSumY = relSumY+vertex.y;
      }
      println();
      println("  - Relative Average Center: ("+(relSumX/vertexCount)+",
```

```
                "+(relSumY/vertexCount)+")");

        // Absolute Corners
        float absSumX = 0;
        float absSumY = 0;
        print("  - Absolute Vertices: ");
        for (int v=0; v<vertexCount; v++)
        {
          Vec2 vertex = polygonShape.getVertex(v);
          print("("+(posX+vertex.x)+","+(posY+vertex.y)+") ");
          absSumX = absSumX+posX+vertex.x;
          absSumY = absSumY+posY+vertex.y;
        }
        println();
        println("  - Absolute Average Center: ("+(absSumX/vertexCount)+",
                "+(absSumY/vertexCount)+")");

      }
    }
    println();
  }
}
```

Annex C.4: Classe Weight

```
// ==========================================
// CLASS WEIGHT (from the Muscular Model Framework)
// by Castanyer (Updated: 08/7/2023)
// For Research Project
// ==========================================
// A weight uses a Box2D body to model a phisical metal disc.
// It is defined by its location (x,y) and it's measurements (length and width).
// It's displayed as a blue circle with the measurements set from its mass.
// ==========================================

class Weight
{
  color WEIGHT_COLOR = color(#7F82BC);
  float WEIGHT_FRICTION = 8;
  float WEIGHT_ELASTICITY = 0.5;
  float WEIGHT_DENSITY = 7874;

  // Bone Properties
  public Body body;

  float posX;
  float posY;
  float sizeRadius;
  float sizeDepth = 0.10;
  float mass;

  String name;

  Weight(float posX, float posY, float mass)
  {
    this(new String("Unknown Weight #")+random(100,999), posX, posY, mass);
  }
  Weight(String name, float posX, float posY, float mass)
  {
    this.name = name;

    this.posX = posX;
    this.posY = posY;
    this.mass = mass;
    this.sizeRadius = sqrt(mass/(WEIGHT_DENSITY * sizeDepth * PI));

    if (this.mass>0)
    {
      // Create BodyDef
      BodyDef bodyDef = new BodyDef();
        bodyDef.type = BodyType.DYNAMIC;
```

```
      bodyDef.position.set(posX, posY);
      bodyDef.linearDamping = WEIGHT_FRICTION;
      bodyDef.angularDamping = WEIGHT_FRICTION;
    body = box2d.createBody(bodyDef);

    // Create Shape
    CircleShape shape = new CircleShape();
      shape.m_radius = this.sizeRadius;

   // Create fixture with shape
    FixtureDef fixDef = new FixtureDef();
      fixDef.shape = shape;
      fixDef.density = WEIGHT_DENSITY*sizeDepth;
      fixDef.friction = WEIGHT_FRICTION;
      fixDef.restitution = WEIGHT_ELASTICITY;
      fixDef.isSensor = true;

    // Attach Shape to Body through Fixture
    body.createFixture(fixDef);
  }
  println("Weight "+name+" Created: Weight = "+body.m_mass+" kg");

}

void display(float displayScale)
{
  Vec2 posCenter = body.getWorldCenter();
  Vec2 pix = new Vec2(box2d.coordWorldToPixels(posCenter.x*displayScale,
                      posCenter.y*displayScale));

  float pixDiameter = box2d.scalarWorldToPixels(this.sizeRadius*2) *
                      displayScale;
  fill(WEIGHT_COLOR);
  noStroke();
  ellipse(pix.x, pix.y, pixDiameter, pixDiameter);
  fill(50);
  ellipse(pix.x, pix.y, box2d.scalarWorldToPixels(0.0254)*displayScale,
          box2d.scalarWorldToPixels(0.0254)*displayScale);
}

void printDetails(String name)
{
  println("Bone Details of '"+name+"'");
  int fixtureCounter=0;

  for (Fixture f = body.getFixtureList(); f!=null ; f = f.getNext())
  {
    fixtureCounter++;
```

```
      println("- Fixture #"+fixtureCounter+":");

      ShapeType shapeType = f.getType();
      if ( shapeType == ShapeType.POLYGON )
      {
        PolygonShape polygonShape = (PolygonShape)f.getShape();
        int vertexCount = polygonShape.getVertexCount();
        println("  - Type: POLYGON ("+vertexCount+" vertices)");

        // Relative Corners
        float relSumX = 0;
        float relSumY = 0;
        print("  - Relative Vertices: ");
        for (int v=0; v<vertexCount; v++)
        {
          Vec2 vertex = polygonShape.getVertex(v);
          print("("+(vertex.x)+","+(vertex.y)+") ");
          relSumX = relSumX+vertex.x;
          relSumY = relSumY+vertex.y;
        }
        println("  - Relative Average Center: ("+(relSumX/vertexCount)+", " +
                (relSumY/vertexCount)+")");

        // Absolute Corners
        float absSumX = 0;
        float absSumY = 0;
        print("  - Absolute Vertices: ");
        for (int v=0; v<vertexCount; v++)
        {
          Vec2 vertex = polygonShape.getVertex(v);
          print("("+(posX+vertex.x)+","+(posY+vertex.y)+") ");
          absSumX = absSumX+posX+vertex.x;
          absSumY = absSumY+posY+vertex.y;
        }
        println("  - Absolute Average Center: ("+(absSumX/vertexCount)+", " +
                (absSumY/vertexCount)+")");
      }
    }
  }
}
```

Annex C.5: Classe Glue

```
// ============================================
// CLASS GLUE (from the Muscular Model Framework)
// by Castanyer (Updated: 3/7/2023)
// For Research Project
// ============================================
// A Glue uses a Box2D WeldJoint to glue two two bodies together, usually bones.
// It joins two bodies by their center points displaced some offset (default
value 0,0).
// And it's displayed as a red line between their centers (?).
// ============================================

class Glue
{
  color GLUE_COLOR = color(255, 0, 0);

  // Glue Properties
  WeldJoint weldJoint;
  String name;

  Glue(Body bodyA, Body bodyB)
  {
    this(new String("Unknown Glue Joint #"), bodyA, bodyB);
  }
  Glue(String name, Body bodyA, Body bodyB)
  {
    this(name, bodyA, bodyB, new Vec2(0,0));
  }

  Glue(Body bodyA, Body bodyB, Vec2 relativePosition)
  {
    this(new String("Unknown Glue Joint #"), bodyA, bodyB, relativePosition);
  }
  Glue(String name, Body bodyA, Body bodyB, Vec2 relativePosition)
  {
    this.name = name;

    WeldJointDef glueDef = new WeldJointDef();

    glueDef.initialize(bodyA, bodyB, relativePosition);

    weldJoint = (WeldJoint) box2d.world.createJoint(glueDef);
  }

  void display(float displayScale) {
      Vec2 posA = new Vec2(0,0);
      Vec2 posB = new Vec2(0,0);
```

```
      weldJoint.getAnchorA(posA);
      weldJoint.getAnchorB(posB);
      Vec2 pixA = new Vec2(box2d.coordWorldToPixels(posA.x*displayScale,
                           posA.y*displayScale));
      Vec2 pixB = new Vec2(box2d.coordWorldToPixels(posB.x*displayScale,
                           posB.y*displayScale));

      fill(GLUE_COLOR);
      stroke(128);
      line(pixA.x, pixA.y, pixB.x, pixB.y);
  }
}
```

Annex C.6: Classe Hinge

```
// ============================================
// CLASS HINGE (from the Muscular Model Framework)
// by Castanyer (Updated: 23/7/2023)
// For Research Project
// ============================================
// A Hinge uses a Box2D RevoluteJoint to model a esqueletal joint between two
bodies, usually bones.
// It joins two bodies by their anchor points (default value 0,0), it has a
rotation range defined by a minimum and maximum angle (default 0,360).
// And it's displayed as a white circle of a specific radius set during the
creation.
// ============================================

class Hinge
{
  color HINGE_COLOR = color(255, 255, 255);

  // Hinge Properties
  public RevoluteJoint revoluteJoint;
  float radius;
  float minDegrees;
  float maxDegrees;

  String name;

  Hinge(Body bodyA, Body bodyB, float radius)
  {
    this(new String("Unknown Hinge Joint #")+random(100,999), bodyA, bodyB,
         radius);
  }
  Hinge(String name, Body bodyA, Body bodyB, float radius)
  {
    this(name, bodyA, bodyB, new Vec2(0,0), new Vec2(0,0), radius, 0, 0);
  }

  Hinge(Body bodyA, Body bodyB, float radius, float minDegrees, float
        maxDegrees)
  {
    this(new String("Unknown Hinge #")+random(100,999), bodyA, bodyB, radius,
         minDegrees, maxDegrees);
  }
  Hinge(String name, Body bodyA, Body bodyB, float radius, float minDegrees,
        float maxDegrees)
  {
    this(name, bodyA, bodyB, new Vec2(0,0), new Vec2(0,0), radius, minDegrees,
         maxDegrees);
```

```
}

Hinge(Body bodyA, Body bodyB, Vec2 anchorA, Vec2 anchorB, float radius, float
      minDegrees, float maxDegrees)
{
  this(new String("Unknown Hinge #")+random(100,999), bodyA, bodyB, anchorA,
       anchorB, radius, minDegrees, maxDegrees);
}
Hinge(String name, Body bodyA, Body bodyB, Vec2 anchorA, Vec2 anchorB, float
      radius, float minDegrees, float maxDegrees)
{
  this.name = name;

  this.radius = radius;
  if (maxDegrees>minDegrees)
  {
    this.maxDegrees = maxDegrees;
    this.minDegrees = minDegrees;
  }
  else
  {
    this.maxDegrees = minDegrees;
    this.minDegrees = maxDegrees;
  }


  RevoluteJointDef hingeDef = new RevoluteJointDef();
  hingeDef.bodyA = bodyA;
  hingeDef.bodyB = bodyB;
  hingeDef.localAnchorA = anchorA;
  hingeDef.localAnchorB = anchorB;

  if (minDegrees!=maxDegrees)
  {
    hingeDef.referenceAngle = 0;
    hingeDef.enableLimit = true;
    hingeDef.upperAngle = maxDegrees * PI/180;
    hingeDef.lowerAngle = minDegrees * PI/180;
  }

  revoluteJoint = (RevoluteJoint) box2d.world.createJoint(hingeDef);
}

void display(float displayScale) {

    Vec2 posA = new Vec2(0,0);
    Vec2 posB = new Vec2(0,0);
    revoluteJoint.getAnchorA(posA);
```

```
      revoluteJoint.getAnchorB(posB);
      Vec2 pixA = new Vec2(box2d.coordWorldToPixels(posA.x*displayScale,
                           posA.y*displayScale));
      Vec2 pixB = new Vec2(box2d.coordWorldToPixels(posB.x*displayScale,
                           posB.y*displayScale));

      float pixRadius = box2d.scalarWorldToPixels(radius)*displayScale;

      fill(HINGE_COLOR);
      stroke(128);
      ellipse(pixA.x, pixA.y, pixRadius, pixRadius);
      ellipse(pixA.x, pixA.y, 2, 2);
      ellipse(pixB.x, pixB.y, pixRadius, pixRadius);
      ellipse(pixB.x, pixB.y, 2, 2);
  }
}
```

Annex C.7: Classe Muscle

```
// ==========================================
// CLASS MUSCLE (from the Muscular Model Framework)
// by Castanyer (Updated: 23/7/2023)
// For Research Project
// ==========================================
// A muscle uses a Box2D body to model a phisical muscle based on Hill's Model
   functions.
// It is defined by its measurements (length and width) and their position
   location (x,y).
// It's attached through the tendons to two anchor points and, optionally,
   through two additional pulleys.
// It's displayed as a pink elipse with the measurements set during the creation
   and two lines as tendons/pulleys.
// It's color shade shows its activation level.
// ==========================================

class Muscle
{
  // https://www.color-hex.com/color-palette/36945
  color MUSCLE_COLOR_LOW = color(223,177,177);
  color MUSCLE_COLOR_HIGH = color(161,44,44);
  color TENDON_COLOR = color(223,177,177);

  static final float MUSCLE_MIN_LENGTH = 0.5; // Min Normalized Length
  static final float MUSCLE_SHORT_LENGTH = 0.75; // Short Normalized Length
  static final float MUSCLE_REST_LENGTH = 1.0; // Rest Normalized Length
  static final float MUSCLE_LONG_LENGTH = 1.25; // Long Normalized Length
  static final float MUSCLE_MAX_LENGTH = 1.5; // Max Normalized Length

  float MUSCLE_ZERO_CONTRACTION = 0.00001;
  float MUSCLE_MAX_VELOCITY = 0.25;
  float MUSCLE_SPECIFIC_TENSION = 1500000;
  float MUSCLE_FRICTION = 8;
  float MUSCLE_ELASTICITY = 0.5;
  float MUSCLE_DENSITY = 1055;
  float TENDON_WIDTH = 0.01;

  // Muscle Properties
  public Body body;

  // Tendon references: Bodies (A,B) and their offsets
  public Body tendonBodyA, tendonBodyB;
  public Vec2 tendonOffsetA, tendonOffsetB;
  Vec2 posTendonAnchorA, posTendonAnchorB;
  float anchorDistance;
```

```
// Pulley references: Bodies (A,B) and their offsets
public Body pulleyBodyA, pulleyBodyB;
public Vec2 pulleyOffsetA, pulleyOffsetB;
Vec2 posPulleyAnchorA, posPulleyAnchorB;

float posX;
float posY;
float angle;
float anglePulleyA;
float anglePulleyB;

float currentWidth;
float currentLength;
float lastLength;
float minLength;
float maxLength;

float restLength;
float restWidth;
float restDrawArea;
float restCrossSection;
float restVolume;

float tendonsLength;
float muscleSymmetry = 0.5;

float contraction = 0.0;
float normalizedContraction = 0.0;
float contractionVelocity = 0.0;
float normalizedVelocity = 0.0;

float activation = 0.0;
float forceMax = 0.0;
float forceActive = 0.0;
float forcePasive = 0.0;
float forceTotal = 0.0;
float forceNow = 0.0;
float tensionGoal = -1.0;

String name;

Muscle(float restLength, float restWidth, float startNormalizedLength, Body
       tendonBodyA, Body tendonBodyB)
{
  this(new String("Unknown Muscle #")+random(100,999), restLength, restWidth,
       startNormalizedLength, tendonBodyA, tendonBodyB, new Vec2(0,0), new
       Vec2(0,0));
}
```

```
Muscle(float restLength, float restWidth, float startNormalizedLength, Body
       tendonBodyA, Body tendonBodyB, Vec2 tendonOffsetA, Vec2 tendonOffsetB)
{
  this(new String("Unknown Muscle #")+random(100,999), restLength, restWidth,
       startNormalizedLength, tendonBodyA, tendonBodyB, tendonOffsetA,
       tendonOffsetB);
}
Muscle(String name, float restLength, float restWidth, float
       startNormalizedLength, Body tendonBodyA, Body tendonBodyB)
{
  this(name, restLength, restWidth, startNormalizedLength, tendonBodyA,
       tendonBodyB, new Vec2(0,0), new Vec2(0,0));
}
Muscle(String name, float restLength, float restWidth, float
       startNormalizedLength, Body tendonBodyA, Body tendonBodyB, Vec2
       tendonOffsetA, Vec2 tendonOffsetB)
{
  this.name = name;

  // Initialize dimensions
  this.restLength = restLength;
  this.lastLength = this.restLength;
  this.restWidth = restWidth;
  this.restDrawArea = restLength * restWidth;
  this.restCrossSection = ((restWidth * restWidth * PI) / 4); //π·(d^2)/4
  this.restVolume = restLength * restCrossSection;

  this.currentWidth = restWidth;
  this.currentLength = restLength;
  this.minLength = restLength * MUSCLE_MIN_LENGTH;
  this.maxLength = restLength * MUSCLE_MAX_LENGTH;

  // Calculate Max Force for this muscle
  this.forceMax = this.restCrossSection * MUSCLE_SPECIFIC_TENSION;
  println("MaxForce: "+forceMax+" N");

  // Initialize position, angle and tendons
  this.tendonBodyA = tendonBodyA;
  this.tendonBodyB = tendonBodyB;
  this.tendonOffsetA = tendonOffsetA;
  this.tendonOffsetB = tendonOffsetB;

  // Set muscle angle aligned with the line defined by attachment points
  PVector lineAB = updateMusclePositionBetweenAnchorPoints();

  // Set tendons length
  tendonsLength = lineAB.mag()-(restLength*startNormalizedLength);
  if (tendonsLength<0) tendonsLength = 0;
```

```
  println("Biceps Tendons: "+tendonsLength);
}

void setPulleys(Body pulleyBodyA, Body pulleyBodyB, Vec2 pulleyOffsetA, Vec2
                pulleyOffsetB)
{
  // Initialize pulley values
  this.pulleyBodyA = pulleyBodyA;
  this.pulleyBodyB = pulleyBodyB;
  this.pulleyOffsetA = pulleyOffsetA;
  this.pulleyOffsetB = pulleyOffsetB;
}

void setMuscleSymmetry(float muscleSymmetry)
{
  if(muscleSymmetry > 1.0 ){muscleSymmetry = 1.0;}
  if(muscleSymmetry < 0.0 ){muscleSymmetry = 0.0;}
  this.muscleSymmetry = muscleSymmetry;
}


void setActivation(float activation)
{
  if (activation>1.0)       activation = 1.0;
  else if (activation<0.0)  activation = 0.0;

  this.activation = activation;
  this.tensionGoal = -1;
}

void setTension(float tension)
{
  if (tension > this.forceMax) { tension = forceMax; }
  else if (tension < 0.0) { tension = 0; }

  this.tensionGoal = tension;
}

void setCurrentLength(float currentLength)
{
  if (currentLength<this.restLength*0.2)
  {
    currentLength=this.restLength*0.2;
  }

  this.currentLength = currentLength;
  // Displayed Area of the muscle is constant
  currentWidth = restDrawArea/currentLength;
```

```
  }

  PVector updateMusclePositionBetweenAnchorPoints()
  {
    updateTendonPositions();
    updatePulleyPositions();

    // Update muscle positions as middle point between tendon attachment points
    this.posX = (this.posTendonAnchorA.x * (1-muscleSymmetry)) +
                (this.posTendonAnchorB.x * muscleSymmetry);
    this.posY = (this.posTendonAnchorA.y * (1-muscleSymmetry)) +
                (this.posTendonAnchorB.y * muscleSymmetry);

// Update muscle angle aligned with the line defined by tendon attachment points
    PVector lineMuscleAB = new PVector(this.posTendonAnchorB.x -
                           this.posTendonAnchorA.x, this.posTendonAnchorB.y -
                           this.posTendonAnchorA.y);
    this.angle = lineMuscleAB.heading();

    return (lineMuscleAB);
  }

  void updateTendonPositions()
  {
    // Update property position anchor tendon A
    if (tendonBodyA!=null)
    {
      float tendonBodyAAngle = tendonBodyA.getAngle();
      Vec2 posTendonBodyA = tendonBodyA.getWorldCenter();
      Vec2 prePosTendonAnchorA = posTendonBodyA.add(tendonOffsetA);
      this.posTendonAnchorA = rotatePoint(prePosTendonAnchorA, posTendonBodyA,
                                          tendonBodyAAngle);
    }

    // Update property position anchor tendon B
    if (tendonBodyA!=null)
    {
      float tendonBodyBAngle = tendonBodyB.getAngle();
      Vec2 posTendonBodyB = tendonBodyB.getWorldCenter();
      Vec2 prePosTendonAnchorB = posTendonBodyB.add(tendonOffsetB);
      this.posTendonAnchorB = rotatePoint(prePosTendonAnchorB, posTendonBodyB,
                                          tendonBodyBAngle);
    }
  }

  void updatePulleyPositions()
  {
    // Update property position anchor pulley A
```

```
  if (pulleyBodyA!=null)
  {
    float pulleyBodyAAngle = pulleyBodyA.getAngle();
    Vec2 posPulleyBodyA = pulleyBodyA.getWorldCenter();
    Vec2 prePosPulleyAnchorA = posPulleyBodyA.add(pulleyOffsetA);
    this.posPulleyAnchorA = rotatePoint(prePosPulleyAnchorA, posPulleyBodyA,
                                        pulleyBodyAAngle);

    PVector linePulleyA = new PVector(this.posTendonAnchorA.x -
                          this.posPulleyAnchorA.x, this.posTendonAnchorA.y -
                          this.posPulleyAnchorA.y);
    this.anglePulleyA = linePulleyA.heading();
  }

  // Update property position anchor pulley B
  if (pulleyBodyB!=null)
  {
    float pulleyBodyBAngle = pulleyBodyB.getAngle();
    Vec2 posPulleyBodyB = pulleyBodyB.getWorldCenter();
    Vec2 prePosPulleyAnchorB = posPulleyBodyB.add(pulleyOffsetB);
    this.posPulleyAnchorB = rotatePoint(prePosPulleyAnchorB, posPulleyBodyB,
                                        pulleyBodyBAngle);

    PVector linePulleyB = new PVector(this.posTendonAnchorB.x -
                          this.posPulleyAnchorB.x, this.posTendonAnchorB.y -
                          this.posPulleyAnchorB.y);
    this.anglePulleyB = linePulleyB.heading();
  }
}

void step(float timeIncrement)
{
  // Set muscle position and angle
  PVector lineAB = updateMusclePositionBetweenAnchorPoints();

  // Calculate muscle length
  // Distance between anchor points (through a LowPas filter)
  anchorDistance += (lineAB.mag() - anchorDistance)*0.2;

  float muscleLength = anchorDistance - tendonsLength;
  setCurrentLength(muscleLength);
  float normalizedSizeLength = this.currentLength / this.restLength;

  // Calculate contraction
  this.contraction = restLength - currentLength;
  this.normalizedContraction = contraction/restLength;

  // Calculate contraction velocity based on last step length
```

```
float lengthVariation = currentLength - lastLength;
if (lengthVariation<MUSCLE_ZERO_CONTRACTION) { lengthVariation = 0.0; }

this.contractionVelocity = lengthVariation / timeIncrement;
this.normalizedVelocity = contractionVelocity / MUSCLE_MAX_VELOCITY;

// Tension Controller (Proportional)
if(this.tensionGoal > 0)
{
  float error = (this.tensionGoal - this.forceActive) / this.forceMax;
  float correction = error * 0.08; // Kp = 0.08
  this.activation += correction;
  println(" activation + " + correction + "=" + this.activation);
  if (this.activation > 1) { this.activation = 1; }
  if (this.activation < 0) { this.activation = 0; }
}

// Calculate ForceActive = Fmax * (Activation *
   Fa(1-(currentLength/restLength)) * F(speed/maxSpeed) )
this.forceActive = this.forceMax * this.activation *
                   getActiveForceFromLength(normalizedSizeLength) *
                   getForceFromVelocity(normalizedVelocity);

// Calculate ForcePasive = Fmax * Fp(length)
this.forcePasive = this.forceMax *
                   getPassiveForceFromLength(normalizedSizeLength);

// Calculate ForceTotal = forceActive + forcePasive
this.forceTotal = forceActive + forcePasive;

// Calculate ForceNow = delayed ForceTotal
//this.forceNow += (this.forceTotal-this.forceNow)*0.5;
this.forceNow = this.forceTotal;


// Apply Force A to tendon A
Vec2 forceA = rotatePoint(new Vec2(forceNow, 0), angle);
tendonBodyA.applyForce(forceA, posTendonAnchorA);

// If pulley exists apply action and reaction forces
if (pulleyBodyA!=null)
{
  // Force Applied to TendonPulley
  Vec2 actionForceA = rotatePoint(new Vec2(forceNow, 0), anglePulleyA);
  pulleyBodyA.applyForce(actionForceA, posPulleyAnchorA);

  // Reverse Force Applied to TendonAnchor
  Vec2 reactionForceA = rotatePoint(new Vec2(forceNow, 0), anglePulleyA+PI);
```

```
    tendonBodyA.applyForce(reactionForceA, posTendonAnchorA);
  }

  // Apply Force B to tendon B or (if exists) to PulleyB
  // Force Applied to TendonAnchor
  Vec2 forceB = rotatePoint(new Vec2(forceNow, 0), angle+PI);
  tendonBodyB.applyForce(forceB, posTendonAnchorB);

  if (pulleyBodyB!=null)
  {
    // Force Applied to TendonPulley
    Vec2 actionForceB = rotatePoint(new Vec2(forceNow, 0), anglePulleyB);
    pulleyBodyB.applyForce(actionForceB, posPulleyAnchorB);

    // Reverse Force Applied to TendonAnchor
    Vec2 reactionForceB = rotatePoint(new Vec2(forceNow, 0), anglePulleyB+PI);
    tendonBodyB.applyForce(reactionForceB, posTendonAnchorB);
  }

  // Save current length for next step cycle
  lastLength = currentLength;
}

void display(float displayScale)
{
    updateMusclePositionBetweenAnchorPoints();

    // Displaying Pulley A
    if (pulleyBodyA!=null && tendonBodyA!=null)
    {
      Vec2 pixTendonA = new Vec2(box2d.coordWorldToPixels(posTendonAnchorA.x *
                        displayScale, posTendonAnchorA.y*displayScale));
      Vec2 pixPulleyA = new Vec2(box2d.coordWorldToPixels(posPulleyAnchorA.x *
                        displayScale, posPulleyAnchorA.y*displayScale));

      // Displaying Pulley
      float pixPulleyWidth = box2d.scalarWorldToPixels(TENDON_WIDTH);
      strokeWeight(pixPulleyWidth*displayScale);
      stroke(TENDON_COLOR);
      line(pixPulleyA.x, pixPulleyA.y, pixTendonA.x, pixTendonA.y);

      // Displayin Pulley Anchor A
      strokeWeight(0);
      fill(TENDON_COLOR);
      ellipse(pixPulleyA.x, pixPulleyA.y, 1.5 * pixPulleyWidth * displayScale,
              1.5 * pixPulleyWidth * displayScale);
    }
```

```
    // Displaying Pulley B
    if (pulleyBodyB!=null && tendonBodyB!=null)
    {
      Vec2 pixTendonB = new Vec2(box2d.coordWorldToPixels(posTendonAnchorB.x *
                        displayScale, posTendonAnchorB.y*displayScale));
      Vec2 pixPulleyB = new Vec2(box2d.coordWorldToPixels(posPulleyAnchorB.x *
                        displayScale, posPulleyAnchorB.y*displayScale));

      // Displaying Pulley
      float pixPulleyWidth = box2d.scalarWorldToPixels(TENDON_WIDTH);
      strokeWeight(pixPulleyWidth*displayScale);
      stroke(TENDON_COLOR);
      line(pixPulleyB.x, pixPulleyB.y, pixTendonB.x, pixTendonB.y);

      // Displayin Pulley Anchor B
      strokeWeight(0);
      fill(TENDON_COLOR);
      ellipse(pixPulleyB.x, pixPulleyB.y, 1.5 * pixPulleyWidth * displayScale,
              1.5 * pixPulleyWidth * displayScale);
    }

    // Displaying Tendons A & B
    if (tendonBodyA!=null && tendonBodyB!=null)
    {
      Vec2 pixA = new Vec2(box2d.coordWorldToPixels(posTendonAnchorA.x *
                      displayScale, posTendonAnchorA.y*displayScale));
      Vec2 pixB = new Vec2(box2d.coordWorldToPixels(posTendonAnchorB.x *
                      displayScale, posTendonAnchorB.y*displayScale));

      // Displaying Tendons
      float pixTendonWidth = box2d.scalarWorldToPixels(TENDON_WIDTH);
      strokeWeight(pixTendonWidth*displayScale);
      stroke(TENDON_COLOR);
      line(pixA.x, pixA.y, pixB.x, pixB.y);

      // Displayin Anchors
      strokeWeight(0);
      fill(TENDON_COLOR);
      if (pulleyBodyA==null) { ellipse(pixA.x, pixA.y, 1.5 * pixTendonWidth *
          displayScale, 1.5 * pixTendonWidth * displayScale); }
      if (pulleyBodyB==null) { ellipse(pixB.x, pixB.y, 1.5 * pixTendonWidth *
          displayScale, 1.5 * pixTendonWidth * displayScale); }
    }

    // Displaying Muscle
    Vec2 pix = box2d.coordWorldToPixels(this.posX*displayScale,
                                        this.posY*displayScale);
```

```
      pushMatrix();
      translate(pix.x, pix.y);
      rotate(-this.angle);
      rectMode(CENTER);
      stroke(50);
      strokeWeight(1);

      fill(lerpColor(MUSCLE_COLOR_LOW, MUSCLE_COLOR_HIGH, this.activation));
      ellipse(0, 0, box2d.scalarWorldToPixels(this.currentLength)*displayScale,
              box2d.scalarWorldToPixels(this.currentWidth)*displayScale);
      popMatrix();
  }

  // Auxiliar Funtion
  Vec2 rotatePoint(Vec2 point, float angle)
  {
    return rotatePoint(point, new Vec2(0,0), angle);
  }

  Vec2 rotatePoint(Vec2 point, Vec2 center, float angle)
  {
    Vec2 newPoint = new Vec2();

    //translate point to origin
    newPoint.x = point.x-center.x;
    newPoint.y = point.y-center.y;

    //rotate point
    Vec2 a = new Vec2();
    a.x = newPoint.x * cos(angle) - newPoint.y * sin(angle);
    a.y = newPoint.x * sin(angle) + newPoint.y * cos(angle);
    newPoint = a;

    //translate newPoint back to old offset
    newPoint.x += center.x;
    newPoint.y += center.y;

    return(newPoint);
  }
}
```

Annex C.8: HillsFuncions.pde

```
// ============================================
// HILLS MODEL DATA (from the Muscular Model Framework)
// by Castanyer (Updated: 3/7/2023)
// For Research Project
// ============================================

ArrayList<Float> HillsDataActiveLengthX = new
ArrayList<Float>(java.util.Arrays.asList(0.3422, 0.3815, 0.4364, 0.4859,
0.5165, 0.5424, 0.5667, 0.5848, 0.6052, 0.6224, 0.6327, 0.6452, 0.6601,
0.6766, 0.6931, 0.7057, 0.7214, 0.7386, 0.7543, 0.7834, 0.8077, 0.8281,
0.8626, 0.8948, 0.9388, 0.9780, 1.0110, 1.0283, 1.0432, 1.0612, 1.0793,
1.0989, 1.1217, 1.1405, 1.1766, 1.2002, 1.2190, 1.2527, 1.2841, 1.3077,
1.3359, 1.3776, 1.4050, 1.4317, 1.4560, 1.4765, 1.5047, 1.5322, 1.5557,
1.5808, 1.6107, 1.6405, 1.6656, 1.7033));

ArrayList<Float> HillsDataActiveLengthY = new
ArrayList<Float>(java.util.Arrays.asList(0.0031, 0.0031, 0.0031, 0.0173,
0.0418, 0.0724, 0.1131, 0.1498, 0.2008, 0.2518, 0.2864, 0.3313, 0.3884,
0.4536, 0.5148, 0.5657, 0.6228, 0.6779, 0.7288, 0.8033, 0.8552, 0.8899,
0.9348, 0.9653, 0.9888, 0.9990, 0.9980, 0.9796, 0.9582, 0.9286, 0.8960,
0.8593, 0.8145, 0.7757, 0.7023, 0.6555, 0.6167, 0.5474, 0.4862, 0.4414,
0.3884, 0.3129, 0.2661, 0.2232, 0.1865, 0.1580, 0.1213, 0.0866, 0.0642,
0.0408, 0.0204, 0.0071, 0.0031, 0.0020));

ArrayList<Float> HillsDataPassiveLengthX = new
ArrayList<Float>(java.util.Arrays.asList(0.4106, 0.4718, 0.5408, 0.6569,
0.7792, 0.8475, 0.8906, 0.9337, 0.9643, 0.9996, 1.0302, 1.0631, 1.1039,
1.1400, 1.1761, 1.2075, 1.2420, 1.2718, 1.3016, 1.3220, 1.3424, 1.3596,
1.3776, 1.3957, 1.4137, 1.4333, 1.4514, 1.4686, 1.4851, 1.5063, 1.5220,
1.5447, 1.5643, 1.5863, 1.6020, 1.6247, 1.6490, 1.6663, 1.6843, 1.7008,
1.7188));

ArrayList<Float> HillsDataPassiveLengthY = new
ArrayList<Float>(java.util.Arrays.asList(0.0000, 0.0000, 0.0000, 0.0000,
0.0000, 0.0041, 0.0081, 0.0122, 0.0183, 0.0234, 0.0316, 0.0407, 0.0540,
0.0713, 0.0916, 0.1141, 0.1446, 0.1792, 0.2200, 0.2536, 0.2912, 0.3289,
0.3697, 0.4145, 0.4613, 0.5143, 0.5652, 0.6141, 0.6609, 0.7200, 0.7648,
0.8299, 0.8809, 0.9369, 0.9807, 1.0377, 1.0988, 1.1395, 1.1843, 1.2230,
1.2637));

ArrayList<Float> HillsDataForceVelocityX = new
ArrayList<Float>(java.util.Arrays.asList(-1.0025, -0.8984, -0.8240,
```

```
-0.7546, -0.6952, -0.6233, -0.5589, -0.4944, -0.4325, -0.3680, -0.3185,
-0.2714, -0.2268, -0.1884, -0.1314, -0.0917, -0.0620, -0.0397, -0.0260,
-0.0099, -0.0012, 0.0062, 0.0112, 0.0136, 0.0273, 0.0570, 0.0979,
0.1400, 0.1995, 0.2689, 0.3309, 0.3854, 0.4424, 0.5068, 0.5638, 0.6406,
0.7051, 0.7695, 0.8463, 0.9108, 0.9901, 1.0818));

ArrayList<Float> HillsDataForceVelocityY = new
ArrayList<Float>(java.util.Arrays.asList(0.0055, 0.0208, 0.0374, 0.0568,
0.0748, 0.1025, 0.1302, 0.1620, 0.1981, 0.2424, 0.2825, 0.3283, 0.3767,
0.4266, 0.5180, 0.6066, 0.6870, 0.7673, 0.8366, 0.9197, 0.9972, 1.0859,
1.1773, 1.2465, 1.3158, 1.3850, 1.4349, 1.4681, 1.5014, 1.5277, 1.5457,
1.5568, 1.5679, 1.5748, 1.5817, 1.5873, 1.5928, 1.5942, 1.5970, 1.5997,
1.5997, 1.5983));


float getActiveForceFromLength(float xValue)
{
  return(interpolateY(xValue, HillsDataActiveLengthX,
HillsDataActiveLengthY));
}

float getPassiveForceFromLength(float xValue)
{
  return(interpolateY(xValue, HillsDataPassiveLengthX,
HillsDataPassiveLengthY));
}

float getForceFromVelocity(float xValue)
{
  return(interpolateY(xValue, HillsDataForceVelocityX,
HillsDataForceVelocityY));
}

float interpolateY(float xValue, ArrayList<Float> xList,
ArrayList<Float> yList)
{
  int topEnd = xList.size()-1;
  int bottomEnd = 0;
  int index = (topEnd+bottomEnd)/2;
  boolean foundIndex = false;
  float yValue;

  //Check Boundaries
  if(xValue < xList.get(0))
```

```
{
  return(yList.get(0));
}
else if(xValue > xList.get(xList.size()-1))
{
  return(yList.get(xList.size()-1));
}

//Find Index
while(!foundIndex)
{
  index = floor((topEnd+bottomEnd)/2);
  if (xValue >= xList.get(index) && xValue < xList.get(index + 1))
  {
  foundIndex = true;
  }

  else
  {
    if(xValue < xList.get(index))
    {
      topEnd = index;
    }

    else
    {
      bottomEnd = index;
    }
  }
}
//obtain interpolation range
float x1 = xList.get(index);
float y1 = yList.get(index);
float x2 = xList.get(index+1);
float y2 = yList.get(index+1);
//if xValue is = to a data point there is no need to interpolate
if(xValue == x1)
{
  yValue = y1;
}
else
{
  //calculate fromula of the straight line --> y = mx+c
  float m = (y2-y1)/(x2-x1);
```

```
    float c = -(m*x1-y1);
    //interpolate y value
    yValue = m * xValue + c;
  }
  return(yValue);
}
```

Annex C.9: SampleModel.pde

```
// ==========================================
// CLASS SAMPLE MODEL (subclass of MuscularModel from the Muscular Model
   Framework)
// For Research Project
// ==========================================
// Extends Muscular Model to create an specifi model
// It can contain bones, hinges, muscles and weights
// It allows to create muscle groups and model variants
// ==========================================

class SampleModel extends MuscularModel
{
  // Global objects of the model
  Nail sampleNail;
  Bone sampleBone1, sampleBone2;
  Hinge sampleHinge;
  Muscle sampleMuscle;
  Weight sampleWeight;
  Glue sampleGlue;

  // CONSTRUCTOR ==========================================
  SampleModel(PApplet p, float x, float y, float scale, int fps, int
              simulationLevel)
  {
    super(p, x, y, scale, fps, simulationLevel);
  }
  SampleModel(PApplet p, float x, float y, float scale, int fps,  int
              simulationLevel, int variantIndex, int weightIndex)
  {
    super(p, x, y, scale, fps, simulationLevel, variantIndex, weightIndex);
  }

  void defineMain()
  {
    variantNames.add("Default");

    weightNames.add("0 kg");
    weightNames.add("1 kg");
    weightNames.add("5 kg");
    weightNames.add("10 kg");
    weightNames.add("50 kg");
  }

  void defineBones(float x, float y, int variantIndex)
  {
    sampleBone1 = new Bone("Bone1", x+0, y-0.1, 0.05, 0.2);
```

```
    allBones.add(sampleBone1);

    sampleBone2 = new Bone("Bone2", x+0.2, y-0.6, 0.05, 0.05);
    allBones.add(sampleBone2);
  }

  void defineJoints(float x, float y, int variantIndex)
  {
    // FIXED JOINT
    sampleNail = new Nail("Nail", x, y);
    allNails.add(sampleNail);

    // HINGE JOINT
    sampleHinge = new Hinge("Hinge", sampleNail.body, sampleBone1.body, new
                            Vec2(0,0), new Vec2(0, 0.1), 0.025, 0, 0);
    allHinges.add(sampleHinge);
  }

  void defineMuscles(float x, float y, int variantIndex)
  {
    // MUSCLE
    sampleMuscle = new Muscle("Muscle", 0.3, 0.05, Muscle.MUSCLE_REST_LENGTH,
                              sampleBone1.body, sampleBone2.body,
                              new Vec2(0, -0.1), new Vec2(0, 0));
    allMuscles.add(sampleMuscle);
  }

  void defineWeights(float x, float y, int variantIndex, int weightIndex)
  {
    float[] variantWeights = {0.0, 1.0, 5.0, 10.0, 50.0};

    // WEIGHT
    sampleWeight = new Weight("Weight", x+0.2, y-0.6,
                              variantWeights[weightIndex]);
    allWeights.add(sampleWeight);

    //WEIGHT-RADIUS
    sampleGlue = new Glue(sampleBone2.body, sampleWeight.body);
    allGlues.add(sampleGlue);
  }
} // End of Class
```

Annex D: Codi Demostració

Annex D.1: Demo.pde

```
// ============================================
// DEMO MAIN (for the Muscular Model Framework)
// by Castanyer (Updated: 23/8/2023)
// For Research Project
// ============================================
// Handles the interactive demo for multiple muscular models
// ============================================

//General libraries
import g4p_controls.*;
import java.awt.Font;

int FRAME_RATE = 60;

int currentModel;
int selectedModel;
int simulationLevel;
int modelVariantIndex;
int modelWeightIndex;
int muscleGroupIndex;

float activation;
int pulseLastZeroTime = 0;
boolean activationPulse = false;
boolean activationPulseEnabled = false;

// Window objects (apart from default)
GWindow graphWindow, controlWindow;

// Model object
MuscularModel model;

//Graph objects
LiveGraph graph1, graph2, graph3, graph4;

// Control objects
GButton playPauseButton, reloadButton;
GDropList modelDropList, variantDropList, levelDropList, weightDropList,
muscleGroupDropList;
GSlider activationSlider;
GButton saveGraph1, saveGraph2, saveGraph3, saveGraph4;
```

```
void settings()
{
  //set size of Main Window
  size(displayHeight*3/5, displayHeight*3/5);

  smooth();
}

void setup()
{
  // SetUp Box2D
  box2d = new Box2DProcessing(this);
  box2d.createWorld();

  // Set default currents
  currentModel = 1;
  selectedModel = 1;
  simulationLevel = MuscularModel.SIMULATION_LEVEL_WEIGHTS;
  modelVariantIndex = 0;
  modelWeightIndex = 1;

  // Load current MuscularModel
  reloadSelectedModel();

  // Setup Windows
  setupAllWindows();
  setupGraphsForCurrentModel();
  setupControlWindow();
}


void draw()
{
  background(50);

  if (model.isSimulationRunning==true)
  {
    model.step();

    if (activationPulse)
    {
      pulseLastZeroTime = millis();
      model.setMuscleGroupActivation(muscleGroupIndex, 1);
      activationPulse = false;
    }
    else
    {
      model.setMuscleGroupActivation(muscleGroupIndex,
```

```
                                    activationSlider.getValueF());
      //model.setMuscleGroupTension(muscleGroupIndex,
                                    activationSlider.getValueF()*500);
    }
  }


  model.display();

  addDataPointsToGraphs();

}

// Model Functions ============================

void reloadSelectedModel()
{
  if (selectedModel == 0)
  {
    model = new SampleModel(this, 0.0, 0.3, 75, FRAME_RATE, simulationLevel,
modelVariantIndex, modelWeightIndex);
    model.setGridProperties(0.05, 10);
    model.setGravity(new Vec2(0, -9.8));
  }
  else if (selectedModel == 1)
  {
    model = new ArmModel(this, -0.20, 0.30, 50, FRAME_RATE, simulationLevel,
modelVariantIndex, modelWeightIndex);
    model.setGridProperties(0.05, 10);
    model.setGravity(new Vec2(0, -9.8));
  }
  else if (selectedModel == 2)
  {
    model = new LegModel(this, -0.30, 0.30, 50, FRAME_RATE, simulationLevel,
modelVariantIndex, modelWeightIndex);
    model.setGridProperties(0.05, 10);
    model.setGravity(new Vec2(0, -9.8));
  }

}

// Windows Functions ==========================

void setupAllWindows()
{
    //General setup
  frameRate(FRAME_RATE);
```

```
  // Set Default Window
  surface.setLocation(-7, 0);
  surface.setTitle("Simulation Panel");
  textSize(100); //stop blurry text

  // Create and Set Graph Window
  graphWindow =  GWindow.getWindow(this, "Graph Panel",
                 displayHeight*3/5-5, 0, displayWidth-displayHeight*3/5,
                 displayHeight-70, JAVA2D);
  graphWindow.addDrawHandler(this, "drawGraphPanel");
  graphWindow.textSize(100); //stop blurry text

  // Create and Set Control Window
  int controlWindowHeight = displayHeight*2/5-100;
  int controlWindowWidth = displayHeight*3/5;
  controlWindow =  GWindow.getWindow(this, "Control Panel", -7,
                   displayHeight*3/5+30, controlWindowWidth,
                   controlWindowHeight, JAVA2D);
  controlWindow.addDrawHandler(this, "drawControlPanel");
  controlWindow.textSize(100); //stop blurry text
}

public void drawGraphPanel(PApplet window, GWinData data)
{
  window.background(50);
  if (this.graph1 == null)
  {
    delay(200);
  }
  graph1.display();
  graph2.display();
  graph3.display();
  graph4.display();
}

public void drawControlPanel(PApplet window, GWinData data)
{
  window.background(70);

  // Draw Text Control Panel
  window.textSize(20);
  window.text("Model", 55, 90);
  window.text("Variant", 210, 90);
  window.text("Pes", 390, 90);
  window.text("Simulació", 525, 90);
  window.text("Múscul Controlat", 10, 165);
}
```

```
// Simulation Events ======================================
void keyPressed()
{
  // Triggers an activation pulse
  if (key == ' ' && activationPulseEnabled) {activationPulse = true;
      activationPulseEnabled = false;}
}

void keyReleased()
{
  // Re-enable activation pulse triggering
  if (key == ' ') {activationPulseEnabled = true;}
}

void mouseWheel(MouseEvent event)
{
  // Zooms Simulation content
  float wheelChange = event.getCount();
  model.incrementDisplayScale(-2*wheelChange);
}

void mousePressed() {
  // Reset Simulation Zoom
  if(mouseButton == RIGHT){
    model.resetDisplayScale();
  }
}
```

Annex D.2: DemoControl.pde

```
// ============================================
// DEMO CONTROLS (for the Muscular Model Framework)
// by Castanyer (Updated: 23/8/2023)
// For Research Project
// ============================================
// Handles the GUI functions for the interactive demo
// ============================================

void setupControlWindow()
{
  int controlWindowHeight = displayHeight*2/5-100;
  int controlWindowWidth = displayHeight*3/5;

  int margin = 10;

  playPauseButton = new GButtonPlus(controlWindow, margin, margin,
                        controlWindowWidth/2-2*margin, 50, "PLAY");
  playPauseButton.setFont(new Font("Arial", Font.PLAIN, 24));
  playPauseButton.setLocalColorScheme(7);

  reloadButton = new GButtonPlus(controlWindow,
                margin+controlWindowWidth/2, margin,
                controlWindowWidth/2-2*margin, 50, "RELOAD");
  reloadButton.setFont(new Font("Arial", Font.PLAIN, 24));
  reloadButton.setLocalColorScheme(7);

  // GDropList => GDropListPlus
  modelDropList = new GDropListPlus(controlWindow, margin, 2*margin+80,
                 ((controlWindowWidth/4-2*margin)), 110, 0, 20);
  modelDropList.setFont(new Font("Arial", Font.PLAIN, 20));
  modelDropList.addItem("Exemple Mínim");
  modelDropList.addItem("Braç Superior");
  modelDropList.addItem("Cama Inferior");
  modelDropList.setLocalColorScheme(4);
  modelDropList.setSelected(selectedModel);

  variantDropList = new GDropListPlus(controlWindow,
                   ((controlWindowWidth/4*1))+margin, 2*margin+80,
                   ((controlWindowWidth/4-2*margin)), 310, 10, 20);
  variantDropList.setFont(new Font("Arial", Font.PLAIN, 20));
  variantDropList.addItem("No hi ha variants creades");
  variantDropList.setLocalColorScheme(4);
  updateVariantDropList();

  weightDropList = new GDropListPlus(controlWindow,
                   ((controlWindowWidth/4*2))+margin, 2*margin+80,
```

```
                ((controlWindowWidth/4-2*margin)), 310, 10, 20);
weightDropList.setFont(new Font("Arial", Font.PLAIN, 20));
weightDropList.addItem("No hi ha pesos creats");
updateWeightDropList();
weightDropList.setLocalColorScheme(4);

levelDropList = new GDropListPlus(controlWindow,
                ((controlWindowWidth/4*3))+margin, 2*margin+80,
                ((controlWindowWidth/4-2*margin)), 310, 10, 20);
levelDropList.setFont(new Font("Arial", Font.PLAIN, 20));
levelDropList.addItem("Res");
levelDropList.addItem("Ossos");
levelDropList.addItem("+Articulacions");
levelDropList.addItem("+Músculs");
levelDropList.addItem("+Pesos");
levelDropList.setLocalColorScheme(4);
levelDropList.setSelected(simulationLevel);

muscleGroupDropList = new GDropListPlus(controlWindow, margin,
                      controlWindowHeight/2+margin,
                      controlWindowWidth-2*margin, 140, 4, 20);
muscleGroupDropList.setFont(new Font("Arial", Font.PLAIN, 20));
muscleGroupDropList.addItem("No hi ha músculs creats");
muscleGroupDropList.setLocalColorScheme(0);
updateGroupDropListFromCurrentModel();

activationSlider = new GSliderPlus(controlWindow, margin,
                   controlWindowHeight/2+25,
                   controlWindowWidth-2*margin, 100, 30);
activationSlider.setLimits(0, 1);
activationSlider.setNumberFormat(G4P.DECIMAL, 2);
activationSlider.setLocalColorScheme(0);
activationSlider.setValue(0);
activationSlider.setShowValue(true);
activationSlider.setEasing(2);

saveGraph1 = new GButtonPlus(controlWindow, margin,
             controlWindowHeight-60, controlWindowWidth/4-2*margin,
             50, "Exporta gràfic 1");
saveGraph1.setFont(new Font("Arial", Font.PLAIN, 16));
saveGraph1.setLocalColorScheme(7);

saveGraph2 = new GButtonPlus(controlWindow,
             margin+controlWindowWidth/4, controlWindowHeight-60,
             controlWindowWidth/4-2*margin, 50, "Exporta gràfic 2");
saveGraph2.setFont(new Font("Arial", Font.PLAIN, 16));
saveGraph2.setLocalColorScheme(7);
```

```
  saveGraph3 = new GButtonPlus(controlWindow,
               margin+2*controlWindowWidth/4, controlWindowHeight-60,
               controlWindowWidth/4-2*margin, 50, "Exporta gràfic 3");
  saveGraph3.setFont(new Font("Arial", Font.PLAIN, 16));
  saveGraph3.setLocalColorScheme(7);

  saveGraph4 = new GButtonPlus(controlWindow,
               margin+3*controlWindowWidth/4, controlWindowHeight-60,
               controlWindowWidth/4-2*margin, 50, "Exporta gràfic 4");
  saveGraph4.setFont(new Font("Arial", Font.PLAIN, 16));
  saveGraph4.setLocalColorScheme(7);

  // Customized Color Schemes
  GCScheme.changePaletteColor(0, 11, color(230,0,0));//highlighted ball
  GCScheme.changePaletteColor(0, 14, color(230,0,0));//ball clicked
  GCScheme.changePaletteColor(0, 15, color(230,0,0));//ball moving
  GCScheme.changePaletteColor(7, 14, color(232, 198, 105));//ballClicked
  GCScheme.changePaletteColor(4, 14, color(206, 148, 94));//ball clicked
}

void updateControlForCurrentModel()
{
    playPauseButton.setText("PLAY");

    updateGroupDropListFromCurrentModel();

    activationSlider.setValue(0);
    currentModel = selectedModel;
    activation = 0;
}

// Update Droplist Functions ================================
void updateGroupDropListFromCurrentModel()
{
  // set no-items message
  while (muscleGroupDropList.removeItem(0)){};
  muscleGroupDropList.addItem("No hi ha músculs creats");
  muscleGroupDropList.removeItem(0);

  for(int i = 0; i < model.allMuscleGroups.size(); i++)
  {
    muscleGroupDropList.addItem(model.getMuscleGroupName(i));
  }
  muscleGroupDropList.removeItem(0);
  muscleGroupDropList.setSelected(0);
  muscleGroupIndex = 0;
}
```

```
void updateVariantDropList()
{
  println("updating variant");
  MuscularModel modelDummy;
  if (selectedModel == 0)
  {
    modelDummy = new SampleModel(this, -0.2, 0.15, 100, FRAME_RATE,
simulationLevel, 0, 1);
    println("sample model");
  }
  else if (selectedModel == 1)
  {
    modelDummy = new ArmModel(this, -0.2, 0.15, 100, FRAME_RATE,
simulationLevel, 0, 1);
    println("arm model");
  }
  else if (selectedModel == 2)
  {
    modelDummy = new LegModel(this, -0.30, 0.30, 100, FRAME_RATE,
                  simulationLevel, 0, modelWeightIndex);
    println("leg model");
  }
  else
  {
    modelDummy = null;
  }

  if (modelDummy != null)
  {
    while (variantDropList.removeItem(0)){};
    for(int i = 0; i < modelDummy.variantNames.size(); i++)
    {
      variantDropList.addItem(modelDummy.variantNames.get(i));
    }

    variantDropList.removeItem(0);
    if (modelVariantIndex > modelDummy.variantNames.size()-1)
    {
      modelVariantIndex = modelDummy.variantNames.size()-1;
    }
    variantDropList.setSelected(modelVariantIndex);
  }
  else
  {
    println("Error on selectedModel when updating variant drop list");
  }
```

```
}

void updateWeightDropList()
{
  MuscularModel modelDummy;
  if (selectedModel == 0)
  {
    modelDummy = new SampleModel(this, -0.2, 0.15, 100, FRAME_RATE,
                 simulationLevel, 0, 1);
    println("sample model");
  }
  else if (selectedModel == 1)
  {
    modelDummy = new ArmModel(this, -0.2, 0.15, 100, FRAME_RATE,
                 simulationLevel, 0, 1);
    println("arm model");
  }
  else if (selectedModel == 2)
  {
    modelDummy = new LegModel(this, -0.2, 0.15, 100, FRAME_RATE,
                 simulationLevel, 0, 1);
    println("leg model");
  }
  else
  {
    modelDummy = null;
  }


  if (modelDummy != null)
  {
    while (weightDropList.removeItem(0)){};
    for(int i = 0; i < modelDummy.weightNames.size(); i++)
    {
      weightDropList.addItem(modelDummy.weightNames.get(i));
    }
    weightDropList.removeItem(0);
    weightDropList.setSelected(1);
    modelWeightIndex = 1;
  }
  else
  {
    println("Error on selectedModel when updating weight drop list");
  }
}

// Slider Event ================================================
void handleSliderEvents(GValueControl slider, GEvent event)
```

```
{
  if (slider == activationSlider)
  {
    activationSlider.setEasing(2);
  }
}

// Droplist Events ==========================================
void handleDropListEvents(GDropList list, GEvent event)
  {
    if(list == muscleGroupDropList)
    {
      muscleGroupIndex = list.getSelectedIndex();
      activationSlider.setEasing(1);
      println("Activation group "+muscleGroupIndex+" is " +
              model.getMuscleGroupActivation(muscleGroupIndex));
      activationSlider.setValue(model.getMuscleGroupActivation(
                                muscleGroupIndex));
    }
    else if(list ==modelDropList)
    {
      selectedModel = list.getSelectedIndex();
      updateVariantDropList();
      updateWeightDropList();
    }
    else if(list == levelDropList)
    {
      simulationLevel = list.getSelectedIndex();
    }
    else if(list == variantDropList)
    {
      modelVariantIndex = list.getSelectedIndex();
    }
    else if(list == weightDropList)
    {
      modelWeightIndex = list.getSelectedIndex();
    }
  }

// Button Events ====================================
void handleButtonEvents(GButton button, GEvent event)
{
  //play pause toggle button
  if (button == playPauseButton)
  {
    model.togglePauseSimulation();

    if(model.isSimulationRunning)
```

```
    {
      playPauseButton.setText("PAUSE");
    }
    else
    {
      playPauseButton.setText("PLAY");
    }
  }
  else if (button == reloadButton)
  {
    reloadSelectedModel();
    resetGraphsData();
    updateControlForCurrentModel();
    setupGraphsForCurrentModel();
  }
  else if (button == saveGraph1)
  {
    graph1.export("graph1");
  }
  else if (button == saveGraph2)
  {
    graph2.export("graph2");
  }
  else if (button == saveGraph3)
  {
    graph3.export("graph3");
  }
  else if (button == saveGraph4)
  {
    graph4.export("graph4");
  }
}


// ====================== DropList FontSize Fix
//https://forum.processing.org/one/topic/g4p-droplist-how-to-change-font-and-or-
size-of-text.html
public class GDropListPlus extends GDropList {

  public GDropListPlus(PApplet theApplet, float p0, float p1, float p2, float
                       p3) {
    super(theApplet, p0, p1, p2, p3);
  }

  public GDropListPlus(PApplet theApplet, int p0, int p1, int p2, int p3, int
                       dropListMaxSize, int other) {
    super(theApplet, p0, p1, p2, p3, dropListMaxSize, other);
  }
```

```
  public void setFont(Font font) {
    if (font != null && font != localFont && buffer != null) {
      localFont = font;
      buffer.g2.setFont(localFont);
      bufferInvalid = true;
    }
  }
}

public class GButtonPlus extends GButton {

  public GButtonPlus(PApplet theApplet, int p0, int p1, int p2, int p3,
                     String text) {
    super(theApplet, p0, p1, p2, p3, text);
  }

  public void setFont(Font font) {
    if (font != null && font != localFont && buffer != null) {
      localFont = font;
      buffer.g2.setFont(localFont);
      bufferInvalid = true;
    }
  }
}

public class GSliderPlus extends GSlider {

  public GSliderPlus(PApplet theApplet, int p0, int p1, int p2, int p3,
                     int p4) {
    super(theApplet, p0, p1, p2, p3, p4);
  }

  public void setFont(Font font) {
    if (font != null && font != localFont && buffer != null) {
      localFont = font;
      buffer.g2.setFont(localFont);
      bufferInvalid = true;
    }
  }
}
```

Annex D.3: DemoGraphs.pde

```
// ============================================
// DEMO GRAPHS (for the Muscular Model Framework)
// by Castanyer (Updated: 23/8/2023)
// For Research Project
// ============================================
// Handles the graphs for the interactive demo
// ============================================

// Graph Colors ================================
color COLOR_TEXT=color(255, 219, 115);
color COLOR_GRID=color(129, 110, 59);
color COLOR_ACTIVATION=color(161,44,44);
color COLOR_TENSION=color(161,44,44);
color COLOR_ACTPASTOT=color(0,0,200);
color COLOR_CONTRACTION=color(0,150,0);

// Graphs Functions ============================
void resetGraphsData()
{
    graph1.resetData();
    graph2.resetData();
    graph3.resetData();
    graph4.resetData();
}

void setupGraphsForCurrentModel()
{
  if (currentModel == 0) //Sample Model
  {
    float graphLength = (displayWidth-displayHeight*3/5);
    float graphHeight = ((displayHeight-70)/4);
    graph1 = new LiveGraph(0, 0, graphLength, graphHeight, graphWindow);
    graph1.setAxisProperties(5, 0, 1.2, (1/float(FRAME_RATE)), "/1");
    graph1.setGridProperties(1, 0.2, COLOR_GRID);
    graph1.setGraphProperties(color(50), COLOR_ACTIVATION, 3);
    graph1.setTextProperties(16, 1, "Nivell d'activació dels múscul",
                             30, new String[]{"Múscul"}, COLOR_TEXT);

    graph2 = new LiveGraph(0, graphHeight, graphLength, graphHeight,
                           graphWindow);
    graph2.setAxisProperties(5, 0, 1600, (1/float(FRAME_RATE)), "N");
    graph2.setGridProperties(1, 200, COLOR_GRID);
    graph2.setGraphProperties(color(50), COLOR_TENSION, 2);
    graph2.setTextProperties(16, 0, "Tensió total dels múscul", 30, new
                             String[]{"Múscul"}, COLOR_TEXT);
```

```
  graph3 = new LiveGraph(0, graphHeight*2, graphLength, graphHeight,
                         graphWindow, 3);
  graph3.setAxisProperties(5, 0, 1600, (1/float(FRAME_RATE)), "N");
  graph3.setGridProperties(1, 200, COLOR_GRID);
  graph3.setGraphProperties(color(50), COLOR_ACTPASTOT, 2);
  graph3.setTextProperties(16, 0, "Tensió activa, passiva i total
                           generada pel múscul", 30, new
                           String[]{"Activa", "Pasiva", "Total"},
                           COLOR_TEXT);

  graph4 = new LiveGraph(0, graphHeight*3, graphLength, graphHeight,
                         graphWindow);
  graph4.setAxisProperties(5, -0.5, 0.5, (1/float(FRAME_RATE)), "/1");
  graph4.setGridProperties(1, 0.1, COLOR_GRID);
  graph4.setGraphProperties(color(50), COLOR_CONTRACTION, 2);
  graph4.setTextProperties(16, 1, "Contracció normalitzada dels
                           biceps", 30, new String[]{"Múscul"},
                           COLOR_TEXT);
}
else if (currentModel == 1) //Arm model
{
  float graphLength = (displayWidth-displayHeight*3/5);
  float graphHeight = ((displayHeight-70)/4);
  graph1 = new LiveGraph(0, 0, graphLength, graphHeight, graphWindow,
                         5);
  graph1.setAxisProperties(5, 0, 1.2, (1/float(FRAME_RATE)), "/1");
  graph1.setGridProperties(1, 0.2, COLOR_GRID);
  graph1.setGraphProperties(color(50), COLOR_ACTIVATION, 3);
  graph1.setTextProperties(16, 1, "Nivell d'activació dels músculs",
                           30, new String[]{"Bi Llarg", "Bi Curt",
                           "Tri Larg", "Tri Medial", "Tri Lateral"},
                           COLOR_TEXT);

  graph2 = new LiveGraph(0, graphHeight, graphLength, graphHeight,
                         graphWindow, 5);
  graph2.setAxisProperties(5, 0, 1000, (1/float(FRAME_RATE)), "N");
  graph2.setGridProperties(1, 100, COLOR_GRID);
  graph2.setGraphProperties(color(50), COLOR_TENSION, 2);
  graph2.setTextProperties(16, 0, "Tensió total dels músculs", 30, new
                           String[]{"Bi Llarg", "Bi Curt",  "Tri
                           Larg", "Tri Medial", "Tri Lateral"},
                           COLOR_TEXT);

  graph3 = new LiveGraph(0, graphHeight*2, graphLength, graphHeight,
                         graphWindow, 3);
  graph3.setAxisProperties(5, 0, 1600, (1/float(FRAME_RATE)), "N");
  graph3.setGridProperties(1, 200, COLOR_GRID);
  graph3.setGraphProperties(color(50), COLOR_ACTPASTOT, 2);
```

```
  graph3.setTextProperties(16, 0, "Tensió activa, passiva i total
                           generada pel biceps curt", 30, new
                           String[]{"Activa", "Pasiva", "Total"},
                           COLOR_TEXT);

  graph4 = new LiveGraph(0, graphHeight*3, graphLength, graphHeight,
                         graphWindow, 2);
  graph4.setAxisProperties(5, -0.5, 0.5, (1/float(FRAME_RATE)), "/1");
  graph4.setGridProperties(1, 0.1, COLOR_GRID);
  graph4.setGraphProperties(color(50), COLOR_CONTRACTION, 2);
  graph4.setTextProperties(16, 1, "Contracció normalitzada dels
                           biceps", 30, new String[]{"Biceps Llarg",
                           "Biceps Curt"}, COLOR_TEXT);
}
else if (currentModel == 2) //Leg model
{
  float graphLength = (displayWidth-displayHeight*3/5);
  float graphHeight = ((displayHeight-70)/4);
  graph1 = new LiveGraph(0, 0, graphLength, graphHeight, graphWindow,
                         4);
  graph1.setAxisProperties(5, 0, 1.2, (1/float(FRAME_RATE)), "/1");
  graph1.setGridProperties(1, 0.2, COLOR_GRID);
  graph1.setGraphProperties(color(50), COLOR_ACTIVATION, 3);
  graph1.setTextProperties(16, 1, "Nivell d'activació dels músculs",
                           30, new String[]{"Gas Lateral", "Gas
                           Medial",  "Soli", "Tib Anterior"},
                           COLOR_TEXT);

  graph2 = new LiveGraph(0, graphHeight, graphLength, graphHeight,
                         graphWindow, 4);
  graph2.setAxisProperties(5, 0, 3500, (1/float(FRAME_RATE)), "N");
  graph2.setGridProperties(1, 500, COLOR_GRID);
  graph2.setGraphProperties(color(50), COLOR_TENSION, 2);
  graph2.setTextProperties(16, 0, "Tensió total dels músculs", 30, new
                           String[]{"Gas Lateral", "Gas Medial",
                           "Soli", "Tib Anterior"}, COLOR_TEXT);

  graph3 = new LiveGraph(0, graphHeight*2, graphLength, graphHeight,
                         graphWindow, 3);
  graph3.setAxisProperties(5, 0, 3500, (1/float(FRAME_RATE)), "N");
  graph3.setGridProperties(1, 500, COLOR_GRID);
  graph3.setGraphProperties(color(50), COLOR_ACTPASTOT, 2);
  graph3.setTextProperties(16, 0, "Tensió activa, passiva i total
                           generada pel Gastrocnemi Lateral", 30, new
                           String[]{"Activa", "Pasiva", "Total"},
                           COLOR_TEXT);

  graph4 = new LiveGraph(0, graphHeight*3, graphLength, graphHeight,
```

```
                            graphWindow, 2);
    graph4.setAxisProperties(5, -0.5, 0.5, (1/float(FRAME_RATE)), "/1");
    graph4.setGridProperties(1, 0.1, COLOR_GRID);
    graph4.setGraphProperties(color(50), COLOR_CONTRACTION, 2);
    graph4.setTextProperties(16, 1, "Contracció normalitzada dels
                             biceps", 30, new String[]{"Gas Lateral",
                             "Gas Medial"}, COLOR_TEXT);
  }
}

void addDataPointsToGraphs()
{
  if (model.isSimulationRunning && model.currentSimulationLevel >
      MuscularModel.SIMULATION_LEVEL_JOINTS)
  {
    if (currentModel == 0) //Sample Model
    {
      if (model.getMuscleByName("Muscle", false) != null)
      {
        float data1 = model.getMuscleByName("Muscle").activation;
        graph1.addPoint(data1);

        float data2 = model.getMuscleByName("Muscle").forceNow;
        graph2.addPoint(data2);

        float[] data3 = new float[]{model.getMuscleByName("Muscle").forceActive,
                                    model.getMuscleByName("Muscle").forcePasive,
                                    model.getMuscleByName("Muscle").forceTotal};
        graph3.addPoint(data3);

        float data4 =
          model.getMuscleByName("Muscle").normalizedContraction;
        graph4.addPoint(data4);
      }
    }
    else if (currentModel == 1) //Arm Model
    {
      if (model.getMuscleByName("Biceps Long", false) != null &&
          model.getMuscleByName("Triceps Long", false) != null)
      {
        float[] data1 = new float[] {
           model.getMuscleByName("Biceps Long").activation,
           model.getMuscleByName("Biceps Short").activation,
           model.getMuscleByName("Triceps Long").activation,
           model.getMuscleByName("Triceps Medial").activation,
           model.getMuscleByName("Triceps Lateral").activation};
        graph1.addPoint(data1);
      }
```

```
    if (model.getMuscleByName("Biceps Long", false) != null &&
        model.getMuscleByName("Triceps Long", false) != null)
    {
      float[] data2 = new float[] {
        model.getMuscleByName("Biceps Long").forceTotal,
        model.getMuscleByName("Biceps Short").forceTotal,
        model.getMuscleByName("Triceps Long").forceTotal,
        model.getMuscleByName("Triceps Medial").forceTotal,
        model.getMuscleByName("Triceps Lateral").forceTotal};
      graph2.addPoint(data2);
    }

    if (model.getMuscleByName("Biceps Short", false) != null &&
        model.currentSimulationLevel > 2)
    {
      float[] data3 = new float[] {
        model.getMuscleByName("Biceps Short").forceActive,
        model.getMuscleByName("Biceps Short").forcePasive,
        model.getMuscleByName("Biceps Short").forceTotal};

      graph3.addPoint(data3);

      float[] data4 = new float[]{
        model.getMuscleByName("Biceps Long",
        false).normalizedContraction, model.getMuscleByName("Biceps
        Short", false).normalizedContraction};
      graph4.addPoint(data4);
    }
  }
  else if (currentModel == 2) //Leg Model
  {
    if (model.getMuscleByName("Gastrocnemius Lateral", false) != null
        && model.getMuscleByName("Tibialis Anterior", false) != null)
    {
      float[] data1 = new float[] {
        model.getMuscleByName("Gastrocnemius Lateral").activation,
        model.getMuscleByName("Gastrocnemius Medial").activation,
        model.getMuscleByName("Soleus").activation,
        model.getMuscleByName("Tibialis Anterior").activation};
      graph1.addPoint(data1);
    }

    if (model.getMuscleByName("Gastrocnemius Lateral", false) != null
        && model.getMuscleByName("Tibialis Anterior", false) != null)
    {
      float[] data2 = new float[] {
        model.getMuscleByName("Gastrocnemius Lateral").forceTotal,
```

```
          model.getMuscleByName("Gastrocnemius Medial").forceTotal,
          model.getMuscleByName("Soleus").forceTotal,
          model.getMuscleByName("Tibialis Anterior").forceTotal};
        graph2.addPoint(data2);
      }

      if (model.getMuscleByName("Gastrocnemius Lateral", false) != null
          && model.currentSimulationLevel > 2)
      {
        float[] data3 = new float[] {
          model.getMuscleByName("Gastrocnemius Lateral").forceActive,
          model.getMuscleByName("Gastrocnemius Lateral").forcePasive,
          model.getMuscleByName("Gastrocnemius Lateral").forceTotal
                                    };
        graph3.addPoint(data3);

        float[] data4 = new float[] {
          model.getMuscleByName("Gastrocnemius Lateral").normalizedContraction,
          model.getMuscleByName("Gastrocnemius Lateral").normalizedContraction};
        graph4.addPoint(data4);
      }
    }
  }
}
```

Annex D.4: LiveGraph.pde

```
// ================================================
// CLASS LIVE GRAPH (Chart Object for the Demo GUI)
// by Castanyer (Updated: 28/8/2023)
// For Research Project
// ================================================
// A LiveGraph shows a dinamic chart ploted from a stream of data.
// It allows an undefined length of data through automatic scroll.
// It also allows to plot multiple channels if they share the same Y axis range.
// ================================================

class LiveGraph
{
  PApplet window;

  float PIX_MARGIN = 60;
  FloatList[] dataPoints;

  // Position and Sizes of the LiveGraph
  float xPos;
  float yPos;
  float windowWidth;
  float windowHeight;
  float graphWidth;
  float graphHeight;

  // Axis limit values and parameters
  float tMax;
  float yMax;
  float yMin;
  float timeIncrement;
  int maxPoints;
  String yUnit;

  // Graph line properties
  color backgroundColor = color(255);
  color lineColor;
  float lineThickness;

  float margin = 15;  //??

  // Grid parameters
  float tGridIncrement;
  float yGridIncrement;
  color gridColor;

  // Text Properties
```

```
  float textSize;
  int decimalPlaces;
  String title;
  float titleSize;
  String channelNames[];
  color textColor;

  // Other
  float zeroYPos;
  int numOfExport = 0;

  // Constructors ================================================

  LiveGraph(float xPos, float yPos, float windowWidth, float
            windowHeight, PApplet window)
  {
    this(xPos, yPos, windowWidth, windowHeight, window, 1);
  }

  LiveGraph(float xPos, float yPos, float windowWidth, float
            windowHeight, PApplet window, int numOfChannels)
  {
    this.window = window;

    this.xPos = xPos;
    this.yPos = yPos;
    this.windowWidth = windowWidth;
    this.windowHeight = windowHeight;

    this.graphHeight = this.windowHeight-2*PIX_MARGIN;
    this.graphWidth = this.windowWidth-4*PIX_MARGIN;

    this.channelNames = new String[numOfChannels];
    for (int i=0; i<numOfChannels; i++)
    {
      this.channelNames[i] = "Channel #"+i;
    }

    dataPoints = new FloatList[numOfChannels];
    resetData();
  }

  // Properties Setters: Axis, Graph, Grid, Text
==========================================
  void setAxisProperties(float tMax, float yMin, float yMax, float
                         timeIncrement)
  {
    setAxisProperties(tMax, yMin, yMax, timeIncrement, "");
```

```
}

void setAxisProperties(float tMax, float yMin, float yMax, float
                       timeIncrement, String yUnit)
{
  this.tMax = tMax;
  this.yMax = yMax;
  this.yMin = yMin;
  this.timeIncrement = timeIncrement;
  this.yUnit = yUnit;

  this.maxPoints = int(nf(tMax/timeIncrement , 0, 0));
  this.zeroYPos = map(0, yMax, yMin, margin, windowHeight);
}

void setGraphProperties(color backgroundColor, color lineColor)
{
  setGraphProperties(backgroundColor, lineColor, 1);
}

void setGraphProperties(color backgroundColor, color lineColor, float
                        lineThickness)
{
  this.backgroundColor = backgroundColor;
  this.lineColor = lineColor;
  this.lineThickness = lineThickness;
}

void setGridProperties(float tGridIncrement, float yGridIncrement,
                       color gridColor)
{
  this.tGridIncrement = tGridIncrement;
  this.yGridIncrement = yGridIncrement;
  this.gridColor = gridColor;
}

void setTextProperties(float textSize, int decimalPlaces)
{
  setTextProperties(textSize, decimalPlaces, "", 1, null, color(0));
}

void setTextProperties(float textSize, int decimalPlaces, String
                       title, float titleSize)
{
  setTextProperties(textSize, decimalPlaces, title, titleSize, null,
                    color(0));
}
```

```
void setTextProperties(float textSize, int decimalPlaces, String
                       title, float titleSize, String[] channelNames,
                       color textColor)
{
  this.textSize = textSize;
  this.decimalPlaces = decimalPlaces;
  this.title = title;
  this.titleSize = titleSize;
  this.textColor = textColor;

  if (channelNames!=null)
  {
    this.channelNames = channelNames;
  }
}

// Data functions ======================================================
void addPoint(float num)
{
  addPoint(new float[]{num});
}

void addPoint(float[] nums)
{
  for (int i = 0; i < dataPoints.length; i++)
  {
    dataPoints[i].append(nums[i]);
  }
}

void resetData()
{
  int numOfChannels = dataPoints.length;
  for (int i = 0; i < numOfChannels; i++)
  {
    dataPoints[i] = new FloatList();
  }
}

void export(String tableName)
{
  PrintWriter output;
  output = createWriter(tableName+".csv");
  String header = new String("time (s)");
  for (int i = 0; i < dataPoints.length; i++)
  {
    header = header + ", " + channelNames[i];
  }
```

```
  output.println(header);

  for (int i = 0; i < dataPoints[0].size(); i++)
  {
    String newLine = new String(str(timeIncrement*i));
    for (int j = 0; j < dataPoints.length; j++)
    {
      newLine = newLine + ", " + str(dataPoints[j].get(i));
    }
    output.println(newLine);
  }
  output.flush();
  output.close();
}

// Display functions ================================================
void display()
{
  PVector pix, endPix;
  //PVector val, endVal;
  String text;

  //graph shift
  int iShift = 0;
  float tShift = 0.0;
  if(dataPoints[0].size() >= maxPoints)
  {
    iShift = dataPoints[0].size() - maxPoints;
    tShift = iShift * timeIncrement;
  }

  // Window
  window.fill(backgroundColor);
  window.noStroke();
  window.rectMode(CORNER);
  window.rect(xPos, yPos, windowWidth, windowHeight);

  // Graph area
  pix = unitsToWindow(0, yMax, 0);
  window.fill(gridColor);
  window.noStroke();
  window.rectMode(CORNER);
  window.rect(pix.x, pix.y, graphWidth, graphHeight);
  window.fill(backgroundColor);
  window.rect(pix.x+1, pix.y+1, graphWidth-1, graphHeight-1);

  // Y axis
  window.strokeWeight(2);
```

```
window.stroke(gridColor);
//0 Line
if (yMax>0 && yMin<0)
{
  pix = unitsToWindow(0, 0, 0);
  endPix = unitsToWindow(tMax, 0, 0);
  window.line(pix.x, pix.y, endPix.x, endPix.y);
}

// Y grid
window.strokeWeight(1);
window.stroke(gridColor);
float yRange = yMax - yMin;
float numberOfDivisions = floor(yRange/yGridIncrement);
float zeroOffset = ((yMin / yGridIncrement)-floor(yMin /
                   yGridIncrement)) * yGridIncrement;

for (int i=0; i<=numberOfDivisions+1; i++)
{
  float y = yMin + i*yGridIncrement - zeroOffset;
  float yBottomLineValue;
  if ((y>=yMin) && (y<=yMax))
  {
    pix = unitsToWindow(0, y);
    endPix = unitsToWindow(tMax, y);
    window.fill(gridColor);
    window.line(pix.x, pix.y, endPix.x, endPix.y);


    pix = unitsToWindow(0, y);
    endPix = unitsToWindow(0, 0);
    yBottomLineValue = floor((endPix.y-pix.y)/yGridIncrement);
    window.textAlign(CENTER);
    window.textSize(textSize);
    window.fill(textColor);
    text = nf(y, 0, decimalPlaces);
    window.text(text, xPos+PIX_MARGIN - textWidth(text) *
               (textSize/100) - 5, pix.y+textSize/4);
  }
}
window.text(yUnit, xPos+(PIX_MARGIN)- textWidth(yUnit) *
           (textSize/100) -3, yPos+(2*PIX_MARGIN/3));

// X grid
numberOfDivisions = floor(tMax/tGridIncrement);
float dShift = floor(tShift/tGridIncrement);
float zeroTimeOffset = ((tShift / tGridIncrement)-dShift) *
                       tGridIncrement;
```

```
for (int i=0; i<=numberOfDivisions+1; i++)
{
  float t = i*tGridIncrement - zeroTimeOffset;
  if (t<=tMax && t>=0)
  {
    pix = unitsToWindow(t, yMin);
    endPix = unitsToWindow(t, yMax);
    window.fill(gridColor);
    window.line(pix.x, pix.y, endPix.x, endPix.y);

    window.textAlign(CENTER);
    window.textSize(textSize);
    window.fill(textColor);
    text = nf((i+dShift) * tGridIncrement, 0, decimalPlaces);
    window.text(text, pix.x, yPos + PIX_MARGIN + graphHeight +
                textSize);
  }
}
window.textSize(textSize);
window.text("s", xPos+PIX_MARGIN + graphWidth + textSize/2 +
            textSize, yPos + PIX_MARGIN + graphHeight + textSize);

// Data lines
window.strokeWeight(lineThickness);
for(int i = 0; i < dataPoints.length; i++)
{
  window.stroke(lerpColor(lineColor, color(255),
                float(i)/float(dataPoints.length)));
  for(int j = iShift+1; j < dataPoints[i].size(); j++)
  {
    pix = unitsToWindow(j*timeIncrement, dataPoints[i].get(j),
                        tShift);
    endPix = unitsToWindow((j-1)*timeIncrement,
             dataPoints[i].get(j-1), tShift);
    float yPixelValueMax = yPos+PIX_MARGIN;
    if(pix.y<yPixelValueMax) {pix.y = yPixelValueMax;}
    if(endPix.y<yPixelValueMax) {endPix.y = yPixelValueMax;}
    float yPixelValueMin = yPos+PIX_MARGIN+graphHeight;
    if(pix.y>yPixelValueMin) {pix.y = yPixelValueMin;}
    if(endPix.y>yPixelValueMin) {endPix.y = yPixelValueMin;}
    window.line(pix.x, pix.y, endPix.x, endPix.y);
  }
}

// Title
window.textSize(titleSize);
window.textAlign(CENTER);
```

```
    window.fill(textColor);
    window.text(title, xPos + windowWidth/2, yPos + PIX_MARGIN/2 +
                titleSize/3);

    // Legend
    if (channelNames!=null && channelNames.length>0)
    {
      float incY = graphHeight / (1+channelNames.length);
      float absX = xPos + PIX_MARGIN + graphWidth + PIX_MARGIN/2;
      float absY = yPos + PIX_MARGIN + graphHeight - incY;
      float squareSize = PIX_MARGIN/3;
      if (squareSize>=incY) { squareSize=incY-1; }

      window.noStroke();
      window.rectMode(CENTER);

      for(int i = 0; i < channelNames.length; i++)
      {
        window.fill(lerpColor(lineColor, color(255),
                    float(i)/float(channelNames.length)));
        window.rect(absX, absY-(incY*i), squareSize, squareSize);

        window.textSize(textSize);
        window.text(channelNames[i], absX + squareSize + 2*textSize,
                    absY-(incY*(0.25+i)) + textSize);
      }
    }
  }

  PVector unitsToWindow(float t, float y)
  {
    return unitsToWindow(t, y, 0);
  }
  PVector unitsToWindow(float t, float y, float tShift)
  {
    float pixX = xPos + (PIX_MARGIN) + map(t-tShift, 0, tMax, 0, graphWidth);
    float pixY = yPos + windowHeight - (PIX_MARGIN) - map(y, yMin, yMax, 0,
graphHeight);
    return(new PVector(pixX, pixY));
  }
}
```

Annex D.5: AnatomicConstants.pde

```
// ==========================================
// ANATOMICAL DATA (from the Muscular Model Framework)
// by Castanyer (Updated: 16/8/2023)
// For Research Project
// ==========================================
// Bones Measurements
float SCAPULA_LENGTH = 0.10;
float SCAPULA_WIDTH  = 0.10;
float HUMERUS_LENGTH = 0.304;
float HUMERUS_WIDTH  = 0.048;
float RADIUS_LENGTH = 0.229;
float RADIUS_WIDTH  = 0.024;
float ULNA_LENGTH   = 0.245;
float ULNA_WIDTH    = 0.019;
float HAND_LENGTH   = 0.080;
float HAND_WIDTH    = 0.060;
float HAND_DEPTH    = 0.020;
float FINGER_LENGTH = 0.070;
float FINGER_WIDTH  = 0.010;

// Muscle Measurements
float BICEPS_SHORT_LENGTH = (0.313)/1.5;
float BICEPS_SHORT_WIDTH  = (0.0164)*2;

float BICEPS_LONG_LENGTH = (0.313)/1.5;
float BICEPS_LONG_WIDTH  = (0.0164)*1.5;

float TRICEPS_LONG_LENGTH = (0.313)/Muscle.MUSCLE_MAX_LENGTH;
float TRICEPS_LONG_WIDTH  = (0.0164)*Muscle.MUSCLE_MAX_LENGTH;

float TRICEPS_LATERAL_LENGTH = (0.313)/Muscle.MUSCLE_MAX_LENGTH;
float TRICEPS_LATERAL_WIDTH  = (0.0164)*Muscle.MUSCLE_MAX_LENGTH;

float TRICEPS_MEDIAL_LENGTH = (0.313)/Muscle.MUSCLE_MAX_LENGTH;
float TRICEPS_MEDIAL_WIDTH  = (0.0164)*Muscle.MUSCLE_MAX_LENGTH;

// Hinges Measurements
float SHOULDER_RADIUS = HUMERUS_WIDTH;
float SHOULDER_ANGLE_MAX = +90;
float SHOULDER_ANGLE_MIN = -160;

float ELBOW_RADIUS = ULNA_WIDTH;
float ELBOW_ANGLE_MAX = +150;
float ELBOW_ANGLE_MIN = -10;

//LEG MODEL=============================================================
```

```
//BONES
float PELVIS_LENGTH = 0.17;
float PELVIS_WIDTH = 0.22;

float FEMUR_LENGTH = 0.48;
float FEMUR_WIDTH = 0.0537;

float FIBULA_LENGTH = 0.35;
float FIBULA_WIDTH = 0.0175;

float TIBIA_LENGTH = 0.36;
float TIBIA_WIDTH = 0.0258;

float FOOT_LENGTH = 0.17;
float FOOT_WIDTH = 0.04;

float TOE_LENGTH = 0.04;
float TOE_WIDTH = 0.02;

//JOINT
float HIP_RADIUS = FEMUR_WIDTH;
float HIP_ANGLE_MAX = 999;
float HIP_ANGLE_MIN = -999;

float KNEE_RADIUS = TIBIA_WIDTH;
float KNEE_ANGLE_MAX = 999;
float KNEE_ANGLE_MIN = -999;

float ANKLE_RADIUS = TIBIA_WIDTH;
float ANKLE_ANGLE_MAX = 500;
float ANKLE_ANGLE_MIN = -70;

float SOLE_RADIUS = TOE_WIDTH;
float SOLE_ANGLE_MAX = 999;
float SOLE_ANGLE_MIN = -999;

//MUSCLE
float GAS_LATERAL_LENGTH = 0.2;
float GAS_LATERAL_WIDTH  = 0.05;

float GAS_MEDIAL_LENGTH = 0.2;
float GAS_MEDIAL_WIDTH  = 0.047;

float SOLEUS_LENGTH = 0.25;
float SOLEUS_WIDTH  = 0.03;

float TIBIALIS_LENGTH = 0.24;
float TIBIALIS_WIDTH  = 0.02;
```

Annex D.6: ArmModel.pde

```
// ============================================
// CLASS ARM MODEL (subclass of MuscularModel from the Muscular Model Framework)
// by Castanyer (Updated: 22/7/2023)
// For Research Project
// ============================================
// Extends Muscular Model to create the model of an arm
// It contains the bones humerus, ulna, radius and a hand
// It contains the muscles biceps short, biceps long and triceps(?)
// ============================================

class ArmModel extends MuscularModel
{
  Nail nail;
  Bone scapula, humerus, ulna, radius;
  Bone hand, fingerIndex, fingerMiddle, fingerRing, fingerPinkie, fingerThumb;
  Glue nailScapula, ulnaRadius, handRadius, handIndex, handMiddle, handRing,
       handPinkie, handThumb, radiusWeight;
  Hinge shoulder, elbow;
  Muscle bicepsShort, bicepsLong;
  Muscle tricepsLong, tricepsLateral, tricepsMedial;
  Weight weight;

  // CONSTRUCTOR ==========================================

  ArmModel(PApplet p, float x, float y, float scale, int fps, int
           simulationLevel)
  {
    super(p, x, y, scale, fps, simulationLevel);
  }
  ArmModel(PApplet p, float x, float y, float scale, int fps, int
           simulationLevel, int variantIndex, int weightIndex)
  {
    super(p, x, y, scale, fps, simulationLevel, variantIndex, weightIndex);
  }

  void defineMain()
  {
    // Defining variants of the models
    variantNames.add("All Muscles");           // Index 0
    variantNames.add("Biceps Short");          // Index 1
    variantNames.add("All Biceps");            // Index 2
    variantNames.add("All Triceps");           // Index 3
    variantNames.add("All Biceps + Triceps");  // Index 4

    weightNames.add("0 kg");              // Index 0
    weightNames.add("1 kg");              // Index 1
```

```
  weightNames.add("5 kg");              // Index 2
  weightNames.add("10 kg");             // Index 3
  weightNames.add("50 kg");             // Index 4
  weightNames.add("100 kg");            // Index 5
}

void defineBones(float x, float y, int variantIndex)
{
  // SCAPULA BONE
  scapula = new Bone("Scapula", x, y+SCAPULA_WIDTH/6, SCAPULA_LENGTH,
                     SCAPULA_WIDTH);
  allBones.add(scapula);

  // HUMERUS BONE
  float offsetHumerusX = HUMERUS_LENGTH/2;
  float offsetHumerusY = 0;
  humerus = new Bone("Humerus", x+offsetHumerusX, y+offsetHumerusY,
                     HUMERUS_LENGTH, HUMERUS_WIDTH);
  allBones.add(humerus);

  // ULNA BONE
  float offsetUlnaX = HUMERUS_LENGTH + ULNA_LENGTH/2;
  float offsetUlnaY = -HUMERUS_WIDTH/2 + ULNA_WIDTH/2;
  ulna = new Bone("Ulna", x+offsetUlnaX, y+offsetUlnaY, ULNA_LENGTH,
                  ULNA_WIDTH);
  allBones.add(ulna);

  // RADIUS BONE
  float offsetRadiusX = HUMERUS_LENGTH + ULNA_LENGTH - RADIUS_LENGTH/2;
  float offsetRadiusY = HUMERUS_WIDTH/2 - RADIUS_WIDTH/2;
  radius = new Bone("Radius", x+offsetRadiusX, y+offsetRadiusY, RADIUS_LENGTH,
                    RADIUS_WIDTH);
  allBones.add(radius);

  // HAND BONES
  float offsetHandX = HUMERUS_LENGTH + ULNA_LENGTH + HAND_LENGTH/2;
  float offsetHandY = 0;
  hand = new Bone("Hand", x+offsetHandX, y+offsetHandY, HAND_LENGTH,
                  HAND_WIDTH, HAND_DEPTH);
  allBones.add(hand);

  float offsetThumbX = HUMERUS_LENGTH + ULNA_LENGTH + (FINGER_LENGTH*0.8)/2;
  float offsetThumbY = HAND_WIDTH/2 + (FINGER_WIDTH*1.5)/2;//+0.002;
  fingerThumb = new Bone("Thumb Finger", x+offsetThumbX, y+offsetThumbY,
                         FINGER_LENGTH*0.8, FINGER_WIDTH*1.5);
  allBones.add(fingerThumb);

  float offsetIndexX = HUMERUS_LENGTH + ULNA_LENGTH + HAND_LENGTH +
```

```
                        (FINGER_LENGTH*0.9)/2;
  float offsetIndexY = HAND_WIDTH/2 - FINGER_WIDTH/2;
  fingerIndex = new Bone("Index Finger", x+offsetIndexX, y+offsetIndexY,
                         (FINGER_LENGTH*0.9), FINGER_WIDTH);
  allBones.add(fingerIndex);

  float offsetMiddleX = HUMERUS_LENGTH + ULNA_LENGTH + HAND_LENGTH +
                        FINGER_LENGTH/2;
  float offsetMiddleY = FINGER_WIDTH/2+0.002;
  fingerMiddle = new Bone("Middle Finger", x+offsetMiddleX, y+offsetMiddleY,
                          FINGER_LENGTH, FINGER_WIDTH);
  allBones.add(fingerMiddle);

  float offsetRingX = HUMERUS_LENGTH + ULNA_LENGTH + HAND_LENGTH +
                      (FINGER_LENGTH*0.9)/2;
  float offsetRingY = - FINGER_WIDTH/2-0.006;
  fingerRing = new Bone("Ring Finger", x+offsetRingX, y+offsetRingY,
                        (FINGER_LENGTH*0.9), FINGER_WIDTH);
  allBones.add(fingerRing);

  float offsetPinkieX = HUMERUS_LENGTH + ULNA_LENGTH + HAND_LENGTH +
                        (FINGER_LENGTH*0.7)/2;
  float offsetPinkieY = -HAND_WIDTH/2 + FINGER_WIDTH/2*0.7;
  fingerPinkie = new Bone("Pinkie Finger", x+offsetPinkieX, y+offsetPinkieY,
                          FINGER_LENGTH*0.7, FINGER_WIDTH*0.7);
  allBones.add(fingerPinkie);
}

void defineJoints(float x, float y, int variantIndex)
{
  // FIXED
  nail = new Nail("Nail", x, y);
  allNails.add(nail);

  // NAIL-SCAPULA GLUE
  nailScapula = new Glue(nail.body, scapula.body);
  allGlues.add(nailScapula);

  // SHOULDER HINGE
  Vec2 shoulderScapulaAnchor = new Vec2(0, -SCAPULA_LENGTH/6);
  Vec2 shoulderHumerusAnchor = new Vec2(-HUMERUS_LENGTH/2, 0);
  shoulder = new Hinge("Shoulder", scapula.body, humerus.body,
                 shoulderScapulaAnchor, shoulderHumerusAnchor,
                 SHOULDER_RADIUS, SHOULDER_ANGLE_MIN, SHOULDER_ANGLE_MAX);
  allHinges.add(shoulder);

  // ELBOW HINGE
  Vec2 elbowHumerusAnchor = new Vec2(+HUMERUS_LENGTH/2, -HUMERUS_WIDTH/2 +
```

```
                                        ULNA_WIDTH/2);
  Vec2 elbowUlnaAnchor = new Vec2(-ULNA_LENGTH/2, 0);
  elbow = new Hinge("Elbow", humerus.body, ulna.body, elbowHumerusAnchor,
          elbowUlnaAnchor, ELBOW_RADIUS, ELBOW_ANGLE_MIN, ELBOW_ANGLE_MAX);
  allHinges.add(elbow);


  // ULNA-RADIUS GLUE
  ulnaRadius = new Glue(ulna.body, radius.body);
  allGlues.add(ulnaRadius);

  //// HAND-RADIUS GLUE
  handRadius = new Glue(hand.body, radius.body);
  allGlues.add(handRadius);

  // HAND-FINGERS GLUE
  handThumb = new Glue(hand.body, fingerThumb.body);
  handIndex = new Glue(hand.body, fingerIndex.body);
  handMiddle = new Glue(hand.body, fingerMiddle.body);
  handRing = new Glue(hand.body, fingerRing.body);
  handPinkie = new Glue(hand.body, fingerPinkie.body);
  allGlues.add(handThumb);
  allGlues.add(handIndex);
  allGlues.add(handMiddle);
  allGlues.add(handRing);
  allGlues.add(handPinkie);
}

void defineMuscles(float x, float y, int variantIndex)
{
  if (variantIndex==0 || variantIndex==1 || variantIndex==2 ||
      variantIndex==4)
  {
    // BICEPS SHORT MUSCLE
    Vec2 bicepsShortScapulaAnchor = new Vec2(SCAPULA_WIDTH/2, 0);
    Vec2 bicepsShortRadiusAnchor = new Vec2(-RADIUS_LENGTH*3/8,
                                            RADIUS_WIDTH/2);
    bicepsShort = new Muscle("Biceps Short", BICEPS_SHORT_LENGTH,
                  BICEPS_SHORT_WIDTH, Muscle.MUSCLE_REST_LENGTH+0.1,
                  scapula.body, radius.body, bicepsShortScapulaAnchor,
                  bicepsShortRadiusAnchor);
    bicepsShort.setMuscleSymmetry(0.6);
    allMuscles.add(bicepsShort);
  }

  if (variantIndex==0 || variantIndex==2 || variantIndex==4)
  {
    // BICEPS LONG MUSCLE
```

```
    Vec2 bicepsLongHumerusAnchor = new Vec2(-HUMERUS_LENGTH/2-0.01,
                                            HUMERUS_WIDTH/2);
    Vec2 bicepsLongRadiusAnchor = new Vec2(-RADIUS_LENGTH*3/8,
                                           RADIUS_WIDTH/2);
    bicepsLong = new Muscle("Biceps Long", BICEPS_LONG_LENGTH,
                 BICEPS_LONG_WIDTH, Muscle.MUSCLE_LONG_LENGTH, humerus.body,
                radius.body, bicepsLongHumerusAnchor, bicepsLongRadiusAnchor);
    Vec2 bicepsScapulaAnchor = new Vec2(0, 0);
    bicepsLong.setPulleys(scapula.body, null, bicepsScapulaAnchor, null );
    bicepsLong.setMuscleSymmetry(0.6);
    allMuscles.add(bicepsLong);
  }

  if (variantIndex==0 || variantIndex==3 || variantIndex==4)
  {
    // TRICEPS MEDIAL MUSCLE
    Vec2 tricepsMedialHumerusAnchorA = new Vec2(0, -HUMERUS_WIDTH/2);
    Vec2 tricepsMedialHumerusAnchorB = new Vec2(+HUMERUS_LENGTH/2+0.01,
                                                -HUMERUS_WIDTH/2-0.01);
    tricepsMedial = new Muscle("Triceps Medial", TRICEPS_MEDIAL_LENGTH,
                    TRICEPS_MEDIAL_WIDTH, Muscle.MUSCLE_REST_LENGTH,
                    humerus.body, humerus.body, tricepsMedialHumerusAnchorA,
                    tricepsMedialHumerusAnchorB);
    Vec2 tricepsMedialUlnaAnchor = new Vec2(-ULNA_LENGTH*3/8, -ULNA_WIDTH/2);
    tricepsMedial.setPulleys(null, ulna.body, null, tricepsMedialUlnaAnchor);
    allMuscles.add(tricepsMedial);

    // TRICEPS LONG MUSCLE
    Vec2 tricepsLongScapulaAnchor = new Vec2(-SCAPULA_LENGTH/2,
                                    SCAPULA_WIDTH/6);
    Vec2 tricepsLongHumerusAnchor = new Vec2(+HUMERUS_LENGTH/2+0.01,
                                    -HUMERUS_WIDTH/2-0.01);
    tricepsLong = new Muscle("Triceps Long", TRICEPS_LONG_LENGTH,
                  TRICEPS_LONG_WIDTH, Muscle.MUSCLE_REST_LENGTH, scapula.body,
                  humerus.body, tricepsLongScapulaAnchor,
                  tricepsLongHumerusAnchor);
    Vec2 tricepsLongUlnaAnchor = new Vec2(-ULNA_LENGTH*3/8, -ULNA_WIDTH/2);
    tricepsLong.setPulleys(null, ulna.body, null, tricepsLongUlnaAnchor);
    allMuscles.add(tricepsLong);

    // TRICEPS LATERAL MUSCLE
    Vec2 tricepsLateralHumerusAnchorA = new Vec2(-HUMERUS_LENGTH/5,
                                        -HUMERUS_WIDTH/2);
    Vec2 tricepsLateralHumerusAnchorB = new Vec2(+HUMERUS_LENGTH/2+0.01,
                                        -HUMERUS_WIDTH/2-0.01);
    tricepsLateral = new Muscle("Triceps Lateral", TRICEPS_LATERAL_LENGTH,
                     TRICEPS_LATERAL_WIDTH, Muscle.MUSCLE_REST_LENGTH,
                     humerus.body, humerus.body, tricepsLateralHumerusAnchorA,
```

```
                        tricepsLateralHumerusAnchorB);
      Vec2 tricepsLateralUlnaAnchor = new Vec2(-ULNA_LENGTH*3/8, -ULNA_WIDTH/2);
      tricepsLateral.setPulleys(null, ulna.body, null,tricepsLateralUlnaAnchor);
      allMuscles.add(tricepsLateral);
    }

    // MUSCLE SETS
    if (variantIndex==0 || variantIndex==3 || variantIndex==4)
      { createMuscleGroup(new String[] {"Triceps Long", "Triceps Medial",
                                        "Triceps Lateral"}); }

    if (variantIndex==0 || variantIndex==2 || variantIndex==4)
      { createMuscleGroup(new String[] {"Biceps Short", "Biceps Long"}); }
  }

  void defineWeights(float x, float y, int variantIndex, int weightIndex)
  {
    float[] variantWeights = {0.0, 1.0, 5.0, 10.0, 50.0, 100.0};
    float weightValue = variantWeights[weightIndex];

    if (weightValue>0)
    {
      // WEIGHT
      float offsetWeightX = HUMERUS_LENGTH + ULNA_LENGTH + RADIUS_LENGTH/4;
      float offsetWeightY = HUMERUS_WIDTH/2 - RADIUS_WIDTH/2;

      weight = new Weight("Weight", x+offsetWeightX, y+offsetWeightY,
                          weightValue);
      allWeights.add(weight);

      //WEIGHT-RADIUS
      radiusWeight = new Glue(radius.body, weight.body);
      allGlues.add(radiusWeight);
    }
  }
} // End of Class
```

Annex D.7: LegModel.pde

```
// ==========================================
// CLASS LEG MODEL (subclass of MuscularModel from the Muscular Model Framework)
// by Castanyer (Updated: 22/7/2023)
// For Research Project
// ==========================================
// Extends Muscular Model to create the model of a leg
// It contains the bones pelvis, femur, fibula, tibia, foot and toes
// It contains the muscles gastrocnemius lateral, gastrocnemius medial, soleus
   and tibialis anterior
// ==========================================

class LegModel extends MuscularModel
{
  Nail nailTop, nailBottom;
  Bone pelvis, femur, fibula, tibia, foot, toe;
  Glue nailTopPelvis, fibulaTibia, nailBottomToe, femurWeight, weightFemur;
  Hinge hip, knee, ankle, sole;
  Muscle gastrocnemiusLateral, gastrocnemiusMedial, soleus, tibialisAnterior;
  Weight weight;

  // CONSTRUCTOR ==========================================

  LegModel(PApplet p, float x, float y, float scale, int fps, int
           simulationLevel)
  {
    super(p, x, y, scale, fps, simulationLevel);
  }
  LegModel(PApplet p, float x, float y, float scale, int fps, int
           simulationLevel, int variantIndex, int weightIndex)
  {
    super(p, x, y, scale, fps, simulationLevel, variantIndex, weightIndex);
  }

  void defineMain()
  {
    // Defining variants of the models
    variantNames.add("All Muscles");                // Index 0
    variantNames.add("All Gastrocnemius");          // Index 1
    variantNames.add("All Gas + Soleus");           // Index 2
    variantNames.add("Tibialis Anterior");          // Index 3
    variantNames.add("Soleus + Tibialis Anterior"); // Index 4

    //weightNames.add("0 kg");              // Index 0
    weightNames.add("10 kg");               // Index 3
    weightNames.add("50 kg");               // Index 4
    weightNames.add("100 kg");              // Index 3
```

```
  weightNames.add("200 kg");              // Index 4
}

void defineBones(float x, float y, int variantIndex)
{
  // PELVIS BONE
  pelvis = new Bone("Pelvis", x, y, PELVIS_LENGTH, PELVIS_WIDTH);
  allBones.add(pelvis);

  // FEMUR BONE
  float offsetFemurX = FEMUR_LENGTH/2;
  float offsetFemurY = -PELVIS_WIDTH/2 + FEMUR_WIDTH/2;
  femur = new Bone("Femur", x+offsetFemurX, y+offsetFemurY, FEMUR_LENGTH,
                   FEMUR_WIDTH);
  allBones.add(femur);

  // FIBULA BONE
  float offsetFibulaX = FEMUR_LENGTH -FIBULA_WIDTH/2 - 0.02;
  float offsetFibulaY = -PELVIS_WIDTH/2 + FEMUR_WIDTH/2 -FIBULA_LENGTH/2;
  fibula = new Bone("Fibula", x+offsetFibulaX, y+offsetFibulaY, FIBULA_WIDTH,
                    FIBULA_LENGTH);
  allBones.add(fibula);

  // TIBIA BONE
  float offsetTibiaX = FEMUR_LENGTH;
  float offsetTibiaY = -PELVIS_WIDTH/2 + FEMUR_WIDTH/2 -TIBIA_LENGTH/2;
  tibia = new Bone("Tibia", x+offsetTibiaX, y+offsetTibiaY, TIBIA_WIDTH,
                   TIBIA_LENGTH);
  allBones.add(tibia);

  // FOOT BONE
  float offsetFootX = FEMUR_LENGTH + FOOT_LENGTH/2 - 0.05;
  float offsetFootY = -PELVIS_WIDTH/2 + FEMUR_WIDTH/2 -TIBIA_LENGTH;
  foot = new Bone("Foot", x+offsetFootX, y+offsetFootY, FOOT_LENGTH,
                  FOOT_WIDTH);
  allBones.add(foot);

  // TOE BONE
  float offsetToeX = FEMUR_LENGTH + FOOT_LENGTH + TOE_LENGTH/2 - 0.05;
  float offsetToeY = -PELVIS_WIDTH/2 + FEMUR_WIDTH/2 -TIBIA_LENGTH
                     -FOOT_WIDTH/2 + TOE_WIDTH/2;
  toe = new Bone("Toe", x+offsetToeX, y+offsetToeY, TOE_LENGTH, TOE_WIDTH,
                 TOE_WIDTH, true);
  allBones.add(toe);
}

void defineJoints(float x, float y, int variantIndex)
{
```

```
// FIXED
nailTop = new Nail("Nail Top", x, y);
allNails.add(nailTop);

nailBottom = new Nail("Nail Bottom", x + FEMUR_LENGTH + FOOT_LENGTH +
TOE_LENGTH/2 - 0.05, y -PELVIS_WIDTH/2 + FEMUR_WIDTH/2 -TIBIA_LENGTH -
FOOT_WIDTH/2 + TOE_WIDTH/2);
allNails.add(nailBottom);

// NAIL TOP-PELVIS GLUE
nailTopPelvis = new Glue(nailTop.body, pelvis.body);
allGlues.add(nailTopPelvis);

// HIP HINGE
Vec2 hipPelvisAnchor = new Vec2(0, -PELVIS_WIDTH/2 + FEMUR_WIDTH/2);
Vec2 hipFemurAnchor = new Vec2(-FEMUR_LENGTH/2, 0);
hip = new Hinge("Hip", pelvis.body, femur.body, hipPelvisAnchor,
                hipFemurAnchor, HIP_RADIUS, HIP_ANGLE_MIN, HIP_ANGLE_MAX);
allHinges.add(hip);

// KNEE HINGE
Vec2 kneeFemurAnchor = new Vec2(FEMUR_LENGTH/2, 0);
Vec2 kneeTibiaAnchor = new Vec2(0, TIBIA_LENGTH/2);
knee = new Hinge("Knee", femur.body, tibia.body, kneeFemurAnchor,
       kneeTibiaAnchor, ELBOW_RADIUS, ELBOW_ANGLE_MIN, ELBOW_ANGLE_MAX);
allHinges.add(knee);

// FIBULA - TIBIA GLUE
fibulaTibia = new Glue(fibula.body, tibia.body);
allGlues.add(fibulaTibia);

// ANKLE HINGE
Vec2 ankleTibiaAnchor = new Vec2(0, -TIBIA_LENGTH/2);
Vec2 ankleFootAnchor = new Vec2(-FOOT_LENGTH/2 + 0.05, 0);
ankle = new Hinge("Ankle", tibia.body, foot.body, ankleTibiaAnchor,
        ankleFootAnchor, ANKLE_RADIUS, ANKLE_ANGLE_MIN, ANKLE_ANGLE_MAX);
allHinges.add(ankle);

// SOLE HINGE
Vec2 soleFootAnchor = new Vec2(FOOT_LENGTH/2, -FOOT_WIDTH/2 + TOE_WIDTH/2);
Vec2 soleToeAnchor = new Vec2(-TOE_LENGTH/2, 0);
sole = new Hinge("Sole", foot.body, toe.body, soleFootAnchor, soleToeAnchor,
                 ELBOW_RADIUS, ELBOW_ANGLE_MIN, ELBOW_ANGLE_MAX);
allHinges.add(sole);

// TOE - NAIL BOTTOM
nailBottomToe = new Glue(nailBottom.body, toe.body);
allGlues.add(nailBottomToe);
```

```
}

void defineMuscles(float x, float y, int variantIndex)
{
  if (variantIndex==0 || variantIndex==2 || variantIndex==4)
  {
    // SOLEUS MUSCLE
    Vec2 soleusFibulaAnchor = new Vec2(0, FIBULA_LENGTH/2);
    Vec2 soleusFootAnchor = new Vec2(-FOOT_LENGTH/2, -FOOT_WIDTH/2);
    soleus = new Muscle("Soleus", SOLEUS_LENGTH, SOLEUS_WIDTH,
                        Muscle.MUSCLE_REST_LENGTH, fibula.body, foot.body,
                        soleusFibulaAnchor, soleusFootAnchor);
    allMuscles.add(soleus);
  }

  if (variantIndex==0 || variantIndex==1 || variantIndex==2)
  {
    // GASTROCNEMIUS LATERAL MUSCLE
    Vec2 gasLateralFemurAnchor = new Vec2(FEMUR_LENGTH/2 - FIBULA_WIDTH -
                                          TIBIA_WIDTH, -FEMUR_WIDTH/2);
    Vec2 gasLateralFootAnchor = new Vec2(-FOOT_LENGTH/2, -FOOT_WIDTH/2);
    gastrocnemiusLateral = new Muscle("Gastrocnemius Lateral",
                           GAS_LATERAL_LENGTH, GAS_LATERAL_WIDTH,
                           Muscle.MUSCLE_REST_LENGTH, femur.body, foot.body,
                           gasLateralFemurAnchor, gasLateralFootAnchor);
    gastrocnemiusLateral.setMuscleSymmetry(0.25);
    allMuscles.add(gastrocnemiusLateral);

    // GASTROCNEMIUS MEDIAL MUSCLE
    Vec2 gasMedialFemurAnchor = new Vec2(FEMUR_LENGTH/2 - TIBIALIS_WIDTH,
                                         -FEMUR_WIDTH/2);
    Vec2 gasMedialFootAnchor = new Vec2(-FOOT_LENGTH/2, -FOOT_WIDTH/2);
    gastrocnemiusMedial = new Muscle("Gastrocnemius Medial",
                          GAS_MEDIAL_LENGTH, GAS_MEDIAL_WIDTH,
                          Muscle.MUSCLE_REST_LENGTH, femur.body, foot.body,
                          gasMedialFemurAnchor, gasMedialFootAnchor);
    gastrocnemiusMedial.setMuscleSymmetry(0.25);
    allMuscles.add(gastrocnemiusMedial);
  }

  if (variantIndex==0 || variantIndex==3 || variantIndex==4)
  {
    // TIBIALIS ANTERIOR MUSCLE
    Vec2 tibAnteriorFibulaAnchor = new Vec2(0, FIBULA_LENGTH/2);
    Vec2 tibAnteriorFootAnchor = new Vec2(0, FOOT_WIDTH/2);
    tibialisAnterior = new Muscle("Tibialis Anterior", TIBIALIS_LENGTH,
                       TIBIALIS_WIDTH, Muscle.MUSCLE_SHORT_LENGTH,
                       fibula.body, foot.body, tibAnteriorFibulaAnchor,
```

```
                        tibAnteriorFootAnchor);
      allMuscles.add(tibialisAnterior);
    }

    // MUSCLE SETS
    if (variantIndex==0 || variantIndex==1 || variantIndex==2)
      { createMuscleGroup(new String[] {"Gastrocnemius Lateral", "Gastrocnemius
                                        Medial"}); }

    if (variantIndex==0 || variantIndex==2)
      { createMuscleGroup(new String[] {"Gastrocnemius Lateral", "Gastrocnemius
                                        Medial", "Soleus"}); }
  }

  void defineWeights(float x, float y, int variantIndex, int weightIndex)
  {
    float[] variantWeights = {10, 50, 100, 200};
    float weightValue = variantWeights[weightIndex];

    if (weightValue>0)
    {
      // WEIGHT
      float offsetWeightX = FEMUR_LENGTH;
      float offsetWeightY = -PELVIS_WIDTH/2 + FEMUR_WIDTH/2;

      weight = new Weight("Weight", x+offsetWeightX, y+offsetWeightY,
                          weightValue);
      allWeights.add(weight);

      //WEIGHT-FEMUR
      weightFemur = new Glue(femur.body, weight.body);
      allGlues.add(weightFemur);
    }
  }
} // End of Class
```