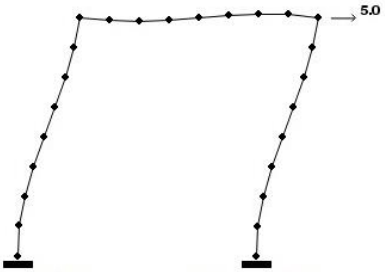
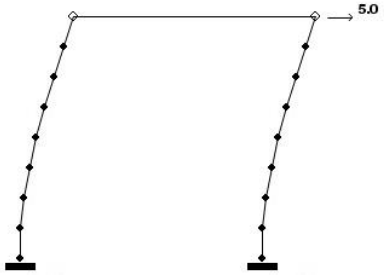
TÉCNICO LISBOA





Simulação de problemas da mecânica estrutural utilizando “physics engines”

João Emanuel Marques de Matos

Dissertação para obtenção do Grau de Mestre em

Engenharia Civil

Orientadores

Professor Doutor Vítor Manuel Azevedo Leitão

Professor Doutor Francisco Afonso Severino Regateiro

Júri

Presidente: Professor Doutor José Joaquim Costa Branco de Oliveira Pedro

Orientador: Professor Doutor Vítor Manuel Azevedo Leitão

Vogal: Professor Doutor Luís Manuel Coelho Guerreiro

Outubro de 2017

Agradecimentos

O desenvolvimento desta dissertação, embora seja um trabalho individual, resultou dos contributos de várias pessoas, cada um à sua maneira, e, por isso, não poderia de deixar de expressar os meus agradecimentos.

Aos meus orientadores, Professor Vítor Leitão e Professor Francisco Regateiro, pela ajuda na resolução de problemas ao longo das várias etapas deste trabalho, pelas suas correções e sugestões, bem como pela sua paciência e disponibilidade para questões que por vezes se mostravam simples e até mesmo triviais.

Ao colega Pedro, autor da dissertação à qual dei continuidade, pela sua disponibilidade e ajuda na compreensão do trabalho anteriormente feito.

À Catarina, colega e amiga, pelas suas incansáveis revisões desta dissertação, pela disponibilização do seu computador, o qual é responsável pela maioria das simulações apresentadas e, principalmente, pela sua amizade. Todo o seu apoio, nestes últimos meses, é responsável por boa parte da minha motivação e persistência neste trabalho.

Aos meus pais e irmão, de quem muito me orgulho e que sempre me apoiaram e fizeram os possíveis e impossíveis para eu atingir os meus objetivos. O seu contributo é de longe o mais importante neste trabalho, pois resulta de uma dedicação sem igual e sem a qual não estaria a escrever estas palavras.

À minha restante família, pelo convívio e apoio, principalmente à Tatiana e Filipa que me acompanharam mais de perto neste percurso.

Aos meus colegas e amigos que de uma forma mais ou menos ativa, contribuíram positivamente ao longo desta ultima fase do curso.

Resumo

Sendo uma das bases para o dimensionamento em Engenharia Civil, a análise estrutural recorre muito a programas computacionais como auxíliares de cálculo. Contudo, tais programas são, maioritariamente, desenvolvidos para profissionais e os seus criadores não estão particularmente interessados em criar produtos mais dedicados aos estudantes. Esta dissertação, foca assim o seu trabalho no desenvolvimento de uma ferramenta computacional didática para o estudo de conceitos de mecânica estrutural.

Neste projeto, conciliando equações da física com conceitos utilizados nos vídeos jogos ("*physics engine*"), foi desenvolvido um simulador gráfico, implementando um Método de Partículas Finitas (MPF), baseado nas leis de Newton e na cinemática. Avaliando apenas o comportamento de massas individuais interligadas entre si (sistema de partículas), este método consegue simular estruturas sem verificar o seu equilíbrio global, uma vez que é verificado localmente nas partículas. Assim, conseguem-se descrever grandes movimentos de partículas ao longo do tempo e oferecer simulações gráficas, em tempo real, de estruturas simples com comportamentos física e geometricamente lineares ou não lineares.

O programa desenvolvido avalia comportamentos física e geometricamente lineares, bem como inicia o estudo de comportamentos fisicamente não lineares, de estruturas reticuladas, permitindo aos utilizadores melhorar o seu entendimento de conceitos lecionados no curso de Engenharia Civil.

Sendo um projeto em progresso, ainda existem melhoramentos possíveis, pois, tanto a precisão de resultados como o desempenho computacional, ainda permitem melhorias. Contudo, este programa já se apresenta como uma ferramenta didática útil no estudo/exposição de conceitos de análise estrutural, da qual tanto alunos como professores podem tirar partido academicamente.

Palavras-chave: ferramenta computacional didática; simulação em tempo real; mecânica estrutural; Método de Partículas Finitas; análise linear; análise não linear.

Abstract

As one of the bases for design in Civil Engineering, structural analysis uses computational programs to support most calculations. However, these programs are, mostly, developed for the professionals and its developers are not, therefore, particularly interested on creating separate products that may be more user-friendly for students. This dissertation focuses its work on the development of a didactic computational tool for the study of structural analysis concepts.

In this project, combining physical equations with video games concepts (physic engine), a graphic simulator was developed, implementing a Finite Particle Method (MPF), based in the laws of Newton and kinematics. By only evaluating the behavior of individual, but interconnected, masses (particle system), this method can simulate structures without checking their global equilibrium, as this is done locally in the particles. It is then possible to describe large particle displacements over time and offer real-time simulations of simple structures with physical and geometrically linear or non-linear behaviors.

This program evaluates physical and geometrically linear behaviors, as well as initiate the study of physically non-linear behaviors, in reticulated structures. The users can improve their understanding of concepts learned in various subjects of the Civil Engineering course.

Being a project in progress, there are still possible improvements, as well as necessary ones, because both the accuracy of results and the computational performance are still open to improvement. However, this program already presents results that could be a useful tool in the study/exposition of structural analysis concepts, with which, both, students and professors could benefit, academically.

Key-words: computational didatic tool; real time simulation; structural mechanics; Finite Particle Method; linear analysis; non-linear analysis.

Índice

Índice de Quadros

Índice de Figuras

Símbolos e Abreviaturas

Símbolos

F – Força

m – Massa

a – Aceleração

σ – Tensão

E – Módulo de elasticidade

ε – Extensão

$\vec{X}$ – Vetor de posição

t – Instante de tempo

Δt – Incremento no tempo

j – Momento de inércia de massa

λ – Densidade linear

l – Comprimento

i – Raio de giração

θ – Rotação

$\Delta\theta$ – Incremento de rotação

P – Forças exteriores das partículas

p – Forças exteriores das barras

Q_z – Momentos exteriores das partículas

q_z – Momentos exteriores das barras

f – Força interior das partículas

m_z – Momento interior das partículas

$\vec{u}$ – Vetor de deformação

$d\vec{u}$ – Vetor de deslocamento relativo

u – Deslocamento relativo longitudinal

v – Deslocamento relativo transversal

$\overrightarrow{du}^r$ – Vetor de deslocamento relativo de corpo rígido

$\overrightarrow{du}^d$ – Vetor de deslocamento relativo sem a parcela de corpo rígido

$\underline{R}$ – Matriz de rotação do incremento de rotação (no intervalo de tempo em análise)

$\underline{Q}$ – Matriz de rotação do ângulo inicial (no intervalo de tempo em análise)

$\underline{Q}''$ – Matriz de rotação do ângulo final (no intervalo de tempo em análise)

$\underline{I}$ – Matriz de identidade

$\overrightarrow{d_e}$ – Vetor de deformações independentes

V – Volume

U_e – Trabalho das forças exteriores

$\overrightarrow{f_e}$ – Vetor de esforços interiores independentes

$\overrightarrow{f_e}^{int}$ – Vetor de esforços interiores

A – Área da seção transversal

I – Inércia da seção transversal

$\underline{T}$ – Matriz de transformação de coordenadas

$\underline{M}$ – Matriz de Massa

$\underline{C}$ – Matriz de amortecimento

$\underline{K}$ – Matriz de Rigidez

c – Coeficiente de amortecimento

M_{pl} – Momento resistente

W_{pl} – Módulo de flexão plástico

k – Coeficiente relativo às condições de apoio de uma viga

f_y – Tensão de cedência

ζ – Coeficiente de amortecimento

p – Frequência própria

p_d – Frequência própria amortecida

T – Período

f – Frequência

w – Frequência da ação exterior

β_1 – Fator de amplificação dinâmica

ϕ – Desfasamento entre a ação e a resposta da estrutura

Abreviaturas

GUI – Interface gráfica de utilizador (*Graphical User Interface*)

MPF – Método de Partículas Finitas

MEF – Método dos Elementos Finitos

$VFIFE$ – *Vector Form Intrinsic Finite Element*

CPU – Unidade de processamento central (*Central Processing Unit)*

GPU – Unidade de processamento gráfico (*Graphic Processing Unit)*

1 Introdução

1.1 Motivação

Designa-se por Mecânica Estrutural o conjunto de temas que tratam da análise das distribuições de esforços e de deformações em estruturas sujeitas às ações ditas mecânicas, isto é, aquelas que consistem na aplicação de forças, na imposição de deslocamentos e ainda as associadas a estados de deformação/tensão internos.

De uma forma simplista pode dizer-se que a Mecânica Estrutural é a base sobre a qual assenta a Análise Estrutural no sentido em que esta dedica particular atenção ao comportamento dos materiais estruturais e ao tipo e características dos elementos estruturais no contexto do projeto de estruturas.

Sendo a base do projeto de estruturas, é essencial ter bons fundamentos nesta área. Sem estes não há possibilidade de entender o comportamento das estruturas, nem é possível proceder ao seu dimensionamento.

É assim importante que este estudo seja bem feito, bem consolidado, sem o qual não será possível abordar o resto da formação na área da Mecânica Estrutural, nomeadamente o cálculo de esforços, de tensões e de deformações nas mais diversas estruturas, isostáticas e hiperstáticas.

Assim, logo nas fases iniciais de diversos cursos de engenharia e em particular no de Engenharia Civil, são introduzidos conceitos chave da Mecânica Estrutural, como a Estática, que vão permitir a determinação de reações e de esforços em estruturas isostáticas.

O ensino destas matérias não tem tido, entre nós e desde há largos anos, evolução significativa. Baseia-se no método tradicional, expositivo, à base de livros e apontamentos escritos e onde o foco é direcionado para o aprofundamento dos conhecimentos teóricos, conceitos, definições e métodos. A aplicação dos conhecimentos teóricos aos diversos tipos de estruturas tem sido também feita, normalmente, da forma tradicional, isto é, manualmente e sem recurso a ferramentas computacionais.

Embora existam, atualmente, ferramentas computacionais com elevadas capacidades de cálculo e análise de estruturas, a sua utilização como complemento à componente teórica da formação na área da Mecânica Estrutural tem sido mais frequente em estágios mais avançados dos cursos de Engenharia Civil, nomeadamente nas disciplinas que tratam do dimensionamento e do projeto de estruturas.

As ferramentas utilizadas são, nesta altura, os mesmos programas utilizados no contexto profissional. Nestes, o foco é a rapidez, a precisão, a possibilidade de tratar grandes problemas, a interação com programas de desenho, entre outras características mais direcionadas à utilização profissional. Está assim fora do âmbito destes programas apresentar os resultados de forma didática, isto é, em que os cálculos possam ser relacionados com matérias que estejam ainda a ser aprendidas, eventualmente, pelo utilizador.

Nas fases iniciais de estudo das matérias de Mecânica Estrutural faz mais sentido utilizar programas que tenham em conta preocupações didáticas, nomeadamente através de simulações gráficas que complementem os cálculos que estejam a ser realizados e que permitam um acompanhamento dos fenómenos físicos ao longo do tempo. A visualização do movimento, dos deslocamentos e rotações que a estrutura vai sofrendo ao longo do tempo e os respetivos efeitos nos esforços é algo que enriquece a formação nesta área.

Uma área de forte crescimento nas últimas décadas, e onde existe grande foco no desenvolvimento das componentes gráfica e interativa (na visualização e no acompanhamento dos fenómenos físicos ao longo do tempo), é a área dos vídeo jogos, onde existe grande exigência na simulação dos movimentos dos objetos, desde personagens, veículos, armas, entre outros. Para aumentar o realismo desses movimentos, para que pareçam genuínos, é necessário aplicar com algum rigor as leis da mecânica. E por isso estes jogos incluem módulos, os chamados "*physics engines*", cuja única função é aplicar devidamente as leis da mecânica aos objetos.

O ensino de Mecânica Estrutural pode beneficiar largamente do trabalho que tem vindo a ser desenvolvido na área dos vídeo jogos através do desenvolvimento de ferramentas computacionais que integrem também um "*physics engine*" e módulos de visualização semelhantes aos usados nos jogos.

Foi nessa base que se deu início, numa anterior dissertação (Lopes 2015), ao desenvolvimento de um programa computacional interativo e baseado nas técnicas dos vídeo jogos que pudesse contribuir para complementar o ensino das matérias de Mecânica Estrutural. Esse trabalho incidiu, essencialmente, sobre a análise de treliças 2D e permite ao utilizador simular e visualizar o comportamento de estruturas em tempo real.

O objetivo principal dos trabalhos desenvolvidos no âmbito desta dissertação é a extensão do trabalho anterior sobre treliças a pórticos, isto é, permitir a possibilidade de análise de estruturas com componentes estruturais à flexão.

1.2 Dos jogos à engenharia

Ao longo dos anos, devido ao aumento de poder de execução dos processadores, placas gráficas e outros componentes dos computadores, os programas de cálculo computacionais (*software*) estão cada vez mais sofisticados, o que tem permitindo uma maior proximidade entre o que se simula (no mundo virtual) e a realidade.

Tanto na engenharia como na indústria dos jogos existe um crescente trabalho neste campo, com algumas especificidades, uma vez que cada uma destas áreas tem como foco principal diferentes aspetos da simulação. A engenharia tenta obter melhores aproximações no que diz respeito à simulação dos fenómenos físicos, ou seja, tenta reproduzir virtualmente os resultados obtidos em casos reais. Na indústria dos jogos, embora haja preocupação em reproduzir adequadamente os movimentos dos objetos, o foco é na qualidade gráfica e interatividade. São estes aspetos que interessa valorizar por permitirem oferecer uma maior proximidade entre o utilizador e o programa.

Com base em ambas as perspetivas é possível aproveitar o melhor dos dois mundos e criar programas com excelente realismo, tanto físico como gráfico. Na área da engenharia civil em particular, é interessante, como já referido, ter um programa que ofereça realismo físico na análise de estruturas, disponibilize graficamente o processo ao longo do tempo e permita ao utilizador ter uma postura interativa antes, durante e após a simulação.

Para construir um programa que simule em tempo real fenómenos físicos, e onde o utilizador possa acompanhar visualmente tal desenvolvimento, são necessários módulos ("motores") que resolvam as equações físicas, atualizem ao longo do tempo os valores associados aos movimentos, avaliem o estado do programa em todos os instantes e que mostrem ao utilizador, graficamente, o que está a acontecer.

Relativamente à resolução das equações físicas, são usados "motores de física" (*Physics Engines*). Estes usam ciclos de cálculo com recurso a avanços temporais entre iterações de forma a simular movimentos reais. A precisão dos resultados varia com a relação entre as equações físicas e o incremento de tempo, pelo que quanto mais precisas as equações e menor o incremento de tempo, maior a proximidade com a realidade (Lopes 2015).

Durante a execução do programa existem alterações de estado ou ambiente (eventos) que precisam de ser reconhecidos, guardados e posteriormente aplicados, pois irão introduzir alterações no seguimento da simulação. Estes eventos podem ser informação/comandos introduzidos pelo utilizador ou alterações que surgem como consequência da simulação (Gregory 2009). A parte do programa destinada a esta função é chamada de função de eventos (*Event Handler*). Neste trabalho, os eventos dependem dos modos do programa e funcionalidades a eles associadas.

Sendo um programa onde existe uma forte interação com o utilizador, é necessária uma interface gráfica de utilizador (GUI - *Graphical User Interface*), onde os resultados gerados e funcionalidades do programa são exibidos. Para tal, usa-se um "motor gráfico" (*Rendering Engine).* Neste programa, a informação é reunida e organizada em dois tipos de janelas: uma respetiva aos elementos físicos e seu comportamento em tempo real e outra que contém informação das propriedades de tais elementos (Lopes 2015).

Embora o objetivo principal dos jogos não seja o realismo físico, existe o interesse de reproduzir movimentos de objetos e suas interações ao longo do tempo sob a ação de forças. Para tal, recorre-se maioritariamente à dinâmica de corpos rígidos e colisões de corpos, pois, na realidade, embora os objetos possam ter diferentes comportamentos ao interagir, dependendo da sua flexibilidade, deformabilidade e outras propriedades intrínsecas, podem ser representados por corpos rígidos. Na indústria dos jogos, independentemente da complexidade gráfica que os objetos pareçam ter, para tratar o seu comportamento físico, e de modo a simplificar o processo de cálculo, tais elementos gráficos são reduzidos a formas simples isoladas, ou interligadas, de modo a representar movimento relativo entre partes de um elemento gráfico (Gregory 2009).

Com base neste tipo de abordagem é possível transportar para a engenharia tais conceitos e replicar estruturas com base em nós (partículas) ligados entre si, conjunto esse conhecido por sistema de

partículas. Recorrendo a massas discretas e elementos de ligação (os quais serão referidos, neste trabalho, como partículas e barras, respetivamente), têm vindo a ser desenvolvidos métodos baseados em partículas para análise de estruturas. As barras não têm massa e assume-se que têm um comportamento como molas elásticas, cuja única função é transmitir, entre massas, a influência que estas têm umas nas outras em função dos respetivos movimentos

Em termos físicos, o movimento de cada uma das partículas pode ser avaliado usando a lei fundamental da Dinâmica, a segunda lei de Newton, equação (1), que relaciona a resultante, numa dada direção, das forças aplicadas, F, numa dada partícula com a sua massa, m, e a respetiva aceleração, a em movimentos retilíneos.

$$F = m\,a \tag{1}$$

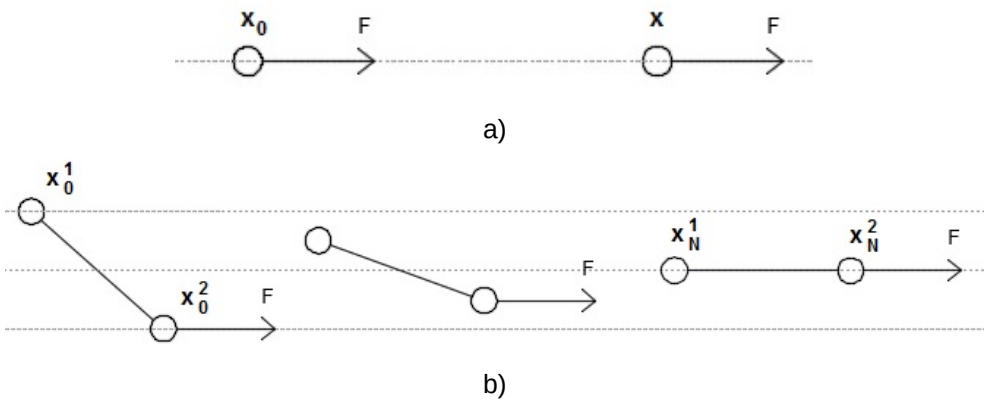


Figura 1: Movimento de partículas. a) Isoladas. b) Ligadas por uma barra.

Num sistema de partículas (um par, por simplicidade) ligadas entre si por um elemento de ligação (uma barra/mola), se as partículas tiverem deslocamentos distintos, o elemento de ligação vai ter que sofrer alteração da sua dimensão inicial, isto é, a barra/mola vai sofrer uma deformação.

A existência dessa deformação, ε, conduz a tensões, σ, em função do módulo de elasticidade E, isto é, tem-se a seguinte relação constitutiva:

$$\sigma = E\,\varepsilon \tag{2}$$

Estas tensões, depois de integradas na seção transversal da barra, vão resultar em forças interiores que, por sua vez, influenciarão o movimento das partículas. Mas esta é apenas uma breve introdução ao método, uma descrição em detalhe será feita no capítulo 3.

Relativamente à simulação dos comportamentos acima descritos, já existem vários “motores de física” no mercado, sendo que alguns oferecem ainda a possibilidade de edição de código para as necessidades do programa. Alguns exemplos são o ODE – *Open Dynamics Engine, Bullet*, *PhysX*, e *Havok* (Gregory 2009).

O dimensionamento e verificação de segurança de estruturas (a análise estrutural) recorre frequentemente ao uso de métodos numéricos especializados como o Método de Elementos Finitos. Fazem-se, normalmente, análises estáticas física e geometricamente lineares bem como análises dinâmicas modais. Este tipo de análises é suficiente para a maioria das situações de dimensionamento de estruturas, no entanto, em determinados casos (como os que envolvem, por exemplo, grandes

deslocamentos, encurvadura, colapso, entre outros (Martini 2001), ou na simulação de comportamentos física ou geometricamente não lineares das estruturas), é necessário recorrer à utilização de ferramentas mais avançadas e, inevitavelmente, mais difíceis de usar.

Na indústria dos jogos têm vindo a ser usadas análises não lineares no domínio do tempo para descrever grandes movimentos de objetos e acompanhar o seu estado em cada instante de jogo. Desta forma, é possível aplicar esta metodologia a programas de engenharia e, juntamente com o "motor gráfico" do programa, acompanhar o movimento da estrutura ao longo da simulação. Surge assim, uma nova forma de analisar comportamentos não lineares de estruturas com base em incrementos de tempo, o que permite não só um acompanhamento em tempo real da simulação, mas também, caso pretendido, interação entre o utilizador e o programa a qualquer instante.

Apesar de escassos, existem já alguns trabalhos baseados em simulações em tempo real para a análise de estruturas. '*Arcade*' (Martini 2001) é o nome de um programa, baseado nos vídeo jogos, que permite a interação e introdução de dados ao longo de uma simulação em tempo real de estruturas simples, lineares ou não lineares. Outro exemplo de programas interativos na análise de estruturas é o '*Push me-Pull me*', desenvolvido pela empresa '*Expedition Engineering*' no projeto '*Expedition Workshed*', e que tem como foco a criação de material didático para o ensino de engenharia (Expedition 2007).

Recorrendo a tais simulações, é possível demonstrar a possibilidade de juntar componentes da indústria dos jogos com métodos de análise de partículas que incorporem "motores de física" (*Physics Engines*) para propósitos didáticos.

Como se viu atrás, os "motores de física" baseiam-se em partículas com massas ligadas ou não por elementos que se comportam como molas. No âmbito da Mecânica Estrutural é apropriado (por ser mais claro) incluir o substantivo "partículas" na designação do método em vez de "motor de física". É esta a razão da designação mais corrente para o método de análise do movimento de partículas atrás referido, o "método das partículas".

Para além das aplicações didáticas, estes métodos têm vindo a provar o seu valor em diversos problemas não lineares. Ao longo dos anos, vários trabalhos foram desenvolvidos por forma a melhor perceber fenómenos relacionados com fragmentação, dinâmica de fluidos, condutividade térmica, entre outros (Asprone, Auricchio et al. 2014). Tais desenvolvimentos, que foram adaptando este tipo de método para corresponder às suas necessidades e também por forma a melhorar os resultados, criaram variantes e complementos, tais como o Método das Partículas Finitas (MPF) e o Método da Relaxação Dinâmica, mais à frente referidos neste trabalho.

Esta dissertação surge então no seguimento de um trabalho prévio, *"Software development for assistance in the learning of structural analysis*", (Lopes 2015), cujo objetivo foi criar de raiz um programa interativo para auxílio na aprendizagem de análise de estruturas que permitisse simular, em tempo real, estruturas articuladas simples 2D e cujo "motor de física" se baseasse num método de análise de partículas.

Neste trabalho, é então dada continuidade a esse programa. Das várias possibilidades de desenvolvimento do trabalho anterior, foi escolhida, por forma a alargar a aplicabilidade do programa, a análise de estruturas simples sujeitas a flexão sob ação de forças constantes e variáveis, bem como o melhoramento da interface gráfica. Este melhoramento é relevante, pois, sendo este um programa interativo e com propósitos didáticos, é de todo importante que o utilizador tenha meios intuitivos de interagir com o programa. Tendo em conta que a dissertação anterior se baseou principalmente no desenvolvimento do programa, outro objetivo deste trabalho é dar ênfase à metodologia de cálculo dos métodos de partículas finitas por forma a dar a entender a sua extensa aplicabilidade.

O objetivo final, em termos computacionais, é então obter um só programa que ofereça ao utilizador a possibilidade de escolha, na análise de estruturas simples, entre as aplicabilidades de ambos os trabalhos, nomeadamente treliças planas, no primeiro trabalho, e pórticos no trabalho aqui descrito. Além da utilização do programa, existe também o objetivo de continuidade deste projeto e com isso possibilitar, a programadores, o desenvolvimento computacional do programa.

1.3 Delineamento da tese

Esta dissertação é composta por quatro capítulos principais pelos quais se divide a apresentação deste trabalho. A sua organização foi pensada por forma a haver um entendimento cada vez mais pormenorizado ao longo da leitura, oferecendo assim uma perceção crescente da abrangência e dimensão deste tema ao mesmo tempo que particulariza no trabalho realizado.

No capítulo um, para além da motivação, é apresentado um enquadramento ao tema desta dissertação de modo a dar a entender de onde surgiu e quais as suas bases.

O capítulo dois apresenta uma descrição do funcionamento do programa computacional e a interface gráfica desenvolvida.

O capítulo três apresenta, detalhadamente, a metodologia de cálculo do método utilizado, sendo, no entanto, apresentada de forma um pouco mais abrangente àquela utilizada no programa, por forma a dar a conhecer as abordagens recentes sobre o método.

O capítulo quatro é referente à validação do modelo, onde são apresentados resultados gerados pelo programa nos casos de estudo deste trabalho.

Para finalizar são apresentadas, no capítulo final, algumas conclusões deste trabalho, desafios e dificuldades superadas, bem como possíveis desenvolvimentos futuros.

2 Programa Computacional

Nesta seção serão apresentadas algumas das principais características do programa computacional. Começando pela linguagem de programação utilizada, segue-se a organização do programa e seu modo de funcionamento. De seguida, são referidos os módulos do programa, onde é descrita a sua implementação e como atuam na organização e tratamento de dados. Por último, são referidos os desenvolvimentos que foram acrescentados ao programa e sua implementação.

Esta secção será dedicada apenas à exposição do conteúdo do programa, sendo algumas das suas funcionalidades de simulação apresentadas no capítulo 4 à medida que os casos de estudo forem sendo apresentados.

Embora o foco deste documento seja o reporte dos desenvolvimentos realizados, serão mantidos alguns conceitos já referidos no trabalho anterior com o mero propósito de mostrar a ligação entre ambos os trabalhos bem como dar uma noção de tais conceitos, uma vez que são importantes para o entendimento deste trabalho.

2.1 Linguagem de programação utilizada

Sendo a continuação de um trabalho pré-existente, mas pretendendo-se construir um só programa onde os utilizadores possam usufruir das suas funcionalidades de forma rápida e interativa, optou-se por manter a linguagem de programação usada nesse trabalho, *Python*. Optou-se por essa linguagem não só por facilitar a compatibilização de ambos os trabalhos, mas também por ser uma linguagem que oferece várias vantagens (essencialmente por ser mais simples) sobre outras linguagens correntes (como, por exemplo, o Java, o C ou o C++).

Para os presentes objetivos, é de todo vantajoso utilizar uma linguagem acessível que permita uma aprendizagem/utilização fácil. O *Python*, sendo uma linguagem direcionada a objetos, permite organizar o código de forma “hierarquizada”. Deste modo, e com o auxílio de classes, além de se conseguir programar de uma forma muito mais compacta, é mais simples compatibilizar código novo com o já existente. Uma outra vantagem é ser uma linguagem onde as variáveis e parâmetros das funções criadas não têm de ser associadas a um determinado tipo (não tem de ser um número real, um número inteiro, um vetor, etc), oferecendo uma maior liberdade ao utilizador/programador para reutilizar segmentos de código (*dynamic typing*).

A combinação destas vantagens com a sintaxe simples do *Python* permite que mesmo um utilizador/programador com pouca experiência consiga ler e perceber de forma quase direta códigos feitos por outros programadores, bem como acrescentar ou desenvolver esse código caso haja necessidade.

Outra das maiores vantagens desta linguagem é a vasta biblioteca de módulos do *Python* já existentes e que possibilita tanto a importação de funcionalidades pré-definidas como a criação de módulos por parte do utilizador/programador com informação que se quer reutilizar. Consegue-se assim compactar o código e evitar a repetição de informação.

Dos vários módulos disponíveis para *Python*, existem alguns associados à criação de interfaces gráficas, tais como *Tkinter, wxWidgets*, entre outros (Rossum 1991). Para decidir sobre qual usar, é necessário definir o nível de interação e qualidade gráfica que se pretende oferecer ao utilizador. Neste trabalho, pela simplicidade dos casos analisados e tipo de programa que se pretende obter nesta fase do projeto, não foram necessários muitos elementos gráficos, sendo possível atingir, com formas geométricas simples, os objetivos propostos. Por isso, e porque já tinha sido usado na dissertação anterior, optou-se pela continuação da utilização do módulo *Pygame*, também ele pertencente à bibioteca do *Python* (Shinners 2000) (Rossum 1991). Tanto o *Python* como o *Pygame* são disponibilizados gratuitamente *online*, sendo a sua instalação bastante direta e simples, o que é mais uma vantagem na utilização desta linguagem.

2.2 Organização do programa

Neste seção será feita uma breve descrição da estrutura do programa e dos seus vários modos de utilização, para o leitor conseguir perceber o fluxo organizacional do programa. Com o objetivo de alargar a aplicabilidade de um programa já funcional, fez sentido aproveitar a estrutura geral do mesmo, tendo sido feitos acréscimos e alterações de acordo com as novas necessidades. Desta forma, a estrutura base do programa não será descrita, podendo, no entanto, ser consultada em (Lopes 2015).

Embora um programa possa ser descrito segundo os seus módulos ("*Engines*"), neste caso será feita uma abordagem com base nos seus modos de funcionamento (*Game modes*), já que pela forma como o programa foi implementado, tal abordagem oferece uma perspetiva mais próxima do seu funcionamento.

Antes de mais, é relevante apresentar o tipo de elementos que existem no programa, sendo estes: partículas (massas), barras (elementos de ligação/molas), grelhas e um ambiente. Os dois primeiros são os responsáveis pela simulação física e sua exibição para o utilizador, as grelhas têm apenas um propósito geométrico no auxílio à criação da estrutura a analisar e o ambiente simula condições exteriores às partículas e barras (Figura 2). Será apresentada na seção 2.4.2 a implementação de tais elementos.

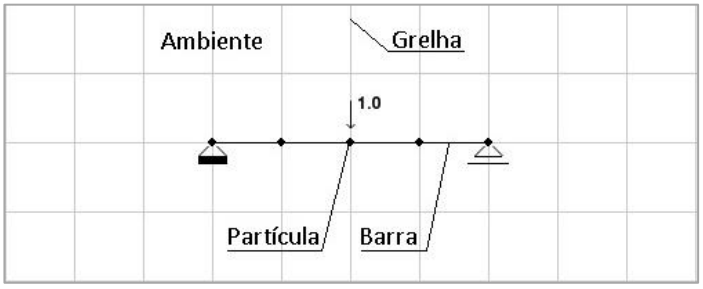


Figura 2: Exemplo representativo dos elementos do programa.

Passando então ao modo de funcionamento do programa, este, durante toda a sua execução, corre dentro de um ciclo principal (*Main Loop*), a partir do qual é verificado constantemente qual dos modos está ativo e é chamada a sua respetiva função. Cada modo representa um estado no programa (máquina de estados), no qual o utilizador navega pelos vários estados de acordo com as suas necessidades de análise.

Dos vários modos, os já existentes no trabalho anterior e que sofreram alterações são: *Simulation* (simulação do fenómeno físico), *Pause* (pausa da simulação), *Properties* (consulta e alteração das propriedades dos elementos), *Save* (armazenamento de ficheiros) e *Load* (carregamento de ficheiros). Nesta dissertação foram criados quatro novos modos referentes à interface gráfica, sendo eles: *Initial Menu* (menu inicial), *Controls* (consulta dos comandos), *Message* (mensagem de alerta ao utilizador) e *About* (informação sobre o programa). Existe um outro modo, *Force Method,* que não sofreu alteração neste trabalho, sendo este apenas aplicável ao modo de análise referente ao trabalho anterior.

Cada um destes modos é implementado na forma de uma função, dentro da qual existe um ciclo que, enquanto o modo não se altere, atualiza e exibe, através do *Event Handler* e *Rendering Engine,* respetivamente, a informação tratada na simulação ou introduzida/alterada pelo utilizador (Figura 3). O modo de simulação tem também incluído o *Physics Engine* no seu ciclo, de forma a gerar e atualizar valores dos elementos associados ao fenómeno físico estudado.

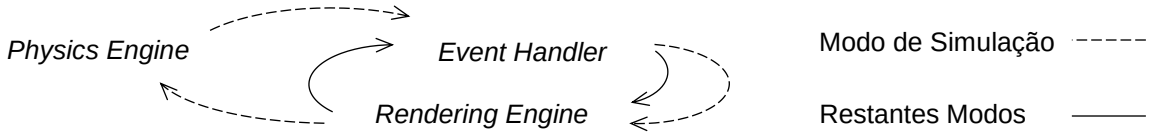


Figura 3: Representação esquemática do ciclo intrínseco a cada modo.

Relativamente à alteração do modo ativo, a transição não pode ser direta entre quaisquer dois modos (Figura 4). Dos vários modos, dois servem de ligação entre as várias transições possíveis, sendo estes os modos '*Initial Menu'* e '*Pause'*. O modo '*Initial Menu'* serve de interface inicial, onde o utilizador tem acesso a informação de como utilizar o programa e funcionalidades disponíveis. O modo '*Pause',* além da sua função principal de parar momentaneamente a simulação, permite a navegação entre praticamente todos os modos, pois qualquer alteração que se queira fazer nos parâmetros da simulação, requer que a simulação esteja em modo pausa (Lopes 2015).

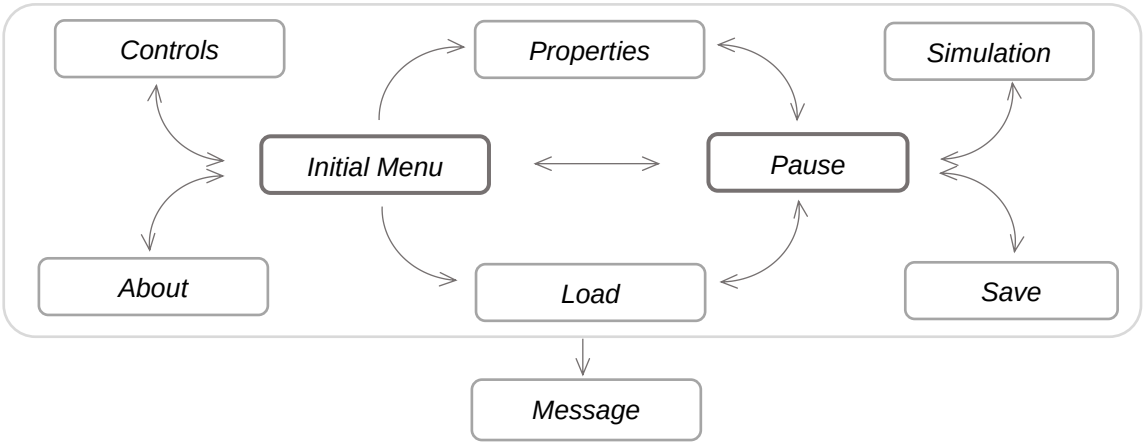


Figura 4: Relação entre modos do programa.

Uma particularidade do programa é que durante uma simulação, independentemente da navegação que seja feita entre os vários modos, não é perdida nenhuma informação referente à estrutura já criada e/ou analisada, ou seja, qualquer percurso feito com base na Figura 4 (à exceção de sair do programa,

passando por *Message*), congela a informação introduzida e/ou alterada no programa até retomar a simulação. Assim, é possível ao utilizador parar a simulação pelas mais variadas razões, desde a alteração de propriedades, consulta de comandos, entre outras. Contudo, é necessário referir que o programa não tem nenhum mecanismo que guarde informação automaticamente. Para tal, o utilizador precisa de recorrer ao modo '*Save'* para guardar uma ou várias versões da simulação*,* caso contrário perderá toda a informação assim que fechar o programa.

2.3 Módulos do programa

Tal como referido no subcapítulo 1.2, são necessários um "motor de física" (*Physics Engine)*, um "motor gráfico" (*Rendering Engine)* e uma função de eventos (*Event Handler)* para criar um programa com capacidade de tratamento de dados ao mesmo tempo que interage com o utilizador em tempo real. Neste programa, cada um desses módulos é implementado segundo uma ou várias funções.

Relativamente ao "motor físico"*,* utilizado somente no modo simulação*,* este é implementado pela função '*updateElementsPhysics',* a qual vai tratar a informação de cada partícula e barra em todos os instantes da simulação, através das devidas equações (capítulo 3), avançar para a iteração seguinte, atualizar a informação necessária e repetir o processo até se obter uma solução satisfatória ou sair do modo de simulação. Todo este processo, dentro da função acima referida, envolve várias outras funções associadas a cada tipo de elemento. Uma descrição mais detalhada daquilo que cada função leva a cabo pode ser encontrada em anexo (capítulo 0).

A função de eventos é implementada sob a forma de uma só função, '*checkEvents'*, e esta já está presente em todos os modos do programa, pois sendo responsável pelo reconhecimento de qualquer alteração de informação, introduzida pelo utilizador ou alterada pela simulação, tem de estar ativa em todos os instantes do programa. Esta função está incorporada no ciclo intrínseco de cada modo, tal como apresentado anteriormente (Figura 3).

Finalmente, o módulo responsável pela componente gráfica é implementado com recurso a várias funções, dependendo do modo em execução. A função comum a todos os modos e que termina o processo de atualização gráfica é a função '*update',* a qual recorre diretamente ao *Pygame* e desenha os elementos na janela do programa. No entanto, é necessário tratar e enviar a informação referente a alterações, durante cada instante, para a função '*update',* e para tal, dependendo do modo ativo, existem várias funções a atuar. No modo de simulação, como existe movimento e este é exibido na janela do programa, é necessário receber informação referente às alterações do ciclo físico (função '*updateElementsGraphics'*) e tratar essa informação de modo a ter os parâmetros organizados de acordo com o necessário na função '*update'* (através das funções '*propertiesToReadableVector'* e '*sendElementsToWindow'*). Nos restantes modos, por não conduzirem a alterações do que se representa na janela do programa, o processo é bastante mais simples envolvendo apenas a função '*update'.*

O ritmo das atualizações dos módulos (eventos, físico e gráfico) não é o mesmo. No modo *Simulation*, onde atuam os três módulos, em cada ciclo de simulação o "motor de física" é atualizado 20 vezes, enquanto a atualização pela função de eventos e "motor gráfico" acontece apenas uma vez por forma

a tornar o processo mais rápido. Esta diferença no ritmo de atualizações tem consequências sobre a visualização das simulações. Se o número de atualizações do "motor de física" é muito grande face ao do "motor gráfico", pode haver perda visível de fluidez de movimento (Lopes 2015).

2.4 Implementação

Embora já tenha sido explicado neste capítulo o funcionamento e organização dos modos do programa, é importante referir a estruturação do código.

Neste subcapítulo, será então apresentada a estrutura do código, a implementação dos elementos do programa e interface gráfica desenvolvida nesta dissertação. Não sendo o objetivo desta dissertação fazer tal descrição em detalhe (tendo em conta que boa parte já foi objeto de descrição detalhada em (Lopes 2015)), serão apresentadas as questões mais relevantes de modo a oferecer ao leitor uma perspetiva geral. Para um maior detalhe, aconselha-se a consulta do anexo (capítulo 0) onde está disposto o código do programa.

2.4.1 Estruturação de código

Havendo o objetivo de criar um programa que possibilitasse a análise do mais variado tipo de estruturas, já fazia parte dos objetivos do trabalho anterior (Lopes 2015) a possível e desejável continuação e desenvolvimento desse trabalho em trabalhos posteriores. Desta forma, esta dissertação além dos objetivos propostos associados ao seu próprio tema, teve também o objetivo de conseguir num só programa computacional a análise quer de treliças planas, quer de pórticos planos, bem como deixar ainda a possibilidade de futuros desenvolvimentos com aplicação a simulações mais complexas.

Para juntar a análise de estruturas articuladas (dissertação anterior) e a análise de elementos sujeitos a flexão (dissertação atual), a primeira fase foi pensar em como compatibilizar o trabalho já feito com o que iria ser desenvolvido. Sendo utilizado o mesmo método de análise em ambas as dissertações, foi de todo vantajoso tirar proveito de informação já existente, visto que os elementos do programa iriam ter características em comum. Assim, a compatibilização foi feita de forma a que o trabalho desta dissertação abrangesse o da anterior, pensando nas treliças como um caso particular de estruturas porticadas. Para melhor entender tal relação, é apresentado num esquema a relação entre ambos os trabalhos (Figura 5).

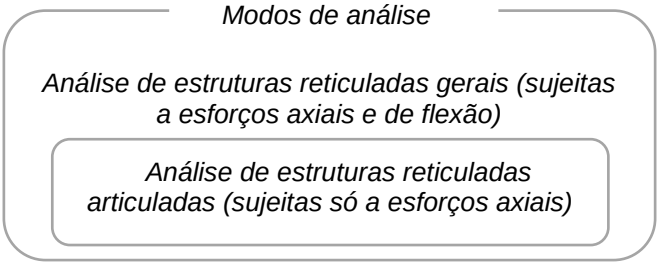


Figura 5: Compatibilização de códigos.

Sendo um programa interativo, procurou-se dar ao utilizador o poder de escolha do tipo de simulação pretendida, mais concretamente analisar a estrutura criada como sendo uma estrutura articulada ou como sendo uma estrutura sujeita à flexão. Para tal, foi criada uma opção no menu de propriedades do ambiente, na qual o utilizador escolhe o tipo de estrutura que deseja recorrendo às palavras '*beam'* e '*truss'*. Com base nesta escolha, o programa funcionará em concordância, ou seja, recorrerá ao trabalho feito na dissertação anterior se for introduzido '*truss*' ou usará informação de ambos os trabalhos se for introduzido '*beam*'.

2.4.2 Elementos

Todos os elementos têm características que poderão ser alteradas, umas por opção do utilizador, outras pelo decorrer da simulação, tal como apresentadas, resumidamente, no Quadro 1.

Quadro 1: Características principais dos elementos do programa.

Elemento	Características principais	
Partícula	▪ Aceleração ▪ Aceleração angular ▪ Massa ▪ Forças ▪ Momentos	▪ Posição ▪ Velocidade ▪ Deslocamentos ▪ Restrições (condições de apoio)
Barra	▪ Comprimento ▪ Deformações axiais e de flexão ▪ Raio de giração	▪ Rigidez axial e de flexão ▪ Densidade linear ▪ Capacidade plástica resistente (sem interação)
Grelha	▪ Espaçamento em x e em y	
Ambiente	▪ Tipo de análise ▪ Aceleração ▪ Aceleração angular ▪ Forças	▪ Momentos ▪ Amortecimento ▪ Escala do programa ("zoom")

A materialização de tais características em termos de código é feita através de dicionários, tuplos e listas (entidades próprias do *Python*), podendo ser consultada em detalhe em anexo (capítulo 0).

Cada um dos elementos é implementado segundo uma classe, ou seja, uma estrutura de dados na qual existem atributos e métodos associados. Todos os objetos criados numa classe terão os mesmos atributos e métodos disponíveis, podendo ser considerados como sendo do mesmo tipo. Tanto a organização de informação como a sua utilização são bastante facilitadas pelo facto de a linguagem de programação utilizada, *Python*, ser direcionada a objetos.

A implementação dos vários elementos é bastante semelhante, por isso será apresentado apenas o caso das partículas por forma a permitir ao leitor perceber a organização base. Cada um destes elementos surge na sua respetiva classe, sendo no caso das partículas na classe '*Particle'*, e onde existem três grupos de atributos: de identificação, físicos e gráficos.

Tanto os atributos físicos como os gráficos surgem também como classes dentro das já existentes (Figura 6), através do conceito de composição de classes. Torna-se assim possível a separação das várias componentes de cada elemento por forma a serem usadas pelos respetivos motores do programa, ao mesmo tempo que torna o código muito mais limpo e simples de consultar. Desta forma, toda a informação física e gráfica apenas existe se existir a informação do elemento. Além das partículas, apenas é utilizada esta organização para as barras; as grelhas e o ambiente, por não serem elementos alteráveis pelo "motor de física", são implementadas usando apenas uma classe.

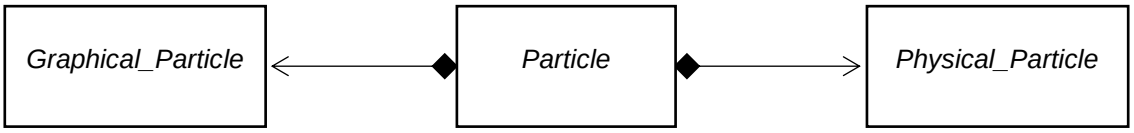


Figura 6: Organização do elemento partícula – Composição de classes (adaptado de (Fowler 2003)).

2.4.3 Interface Gráfica

Havendo o objetivo de criar um programa interativo no qual o utilizador possa navegar e interagir com a simulação, é necessário criar uma interface gráfica. Na fase inicial deste trabalho, ao tentar perceber o funcionamento do programa feito na dissertação anterior, foram sentidas algumas dificuldades, como utilizador, em saber o que o programa oferecia e como utilizar as suas funcionalidades. Embora o programa já tivesse uma interface gráfica, onde a cada modo ativo correspondia uma janela com a informação correspondente, o seu funcionamento era muito à base de comandos por teclas que tinham de ser consultadas no formato de documentos escritos fora do programa. Ou seja, faltava à interface gráfica existente painéis de navegação, que permitissem ao utilizador rápida e intuitivamente aceder e usar as funcionalidades do programa. Nesse seguimento, foi pensada uma forma de organizar um modo de navegação com recurso a botões interativos e comandos por teclas.

Atualmente, e tal como já referido anteriormente, existem módulos associados à linguagem *Python* que oferecem ferramentas para a criação de uma interface gráfica. Optou-se por utilizar o desenvolvimento de tal interface como mais uma oportunidade de aprendizagem para conhecer funcionalidades e desenvolver o conhecimento na área da programação em *Python*. Dessa forma, a interface foi desenvolvida de raiz.

Esta seção será então uma apresentação da interface gráfica por modos do programa, já apresentados no subcapítulo 2.2.

Modo '*Initial Menu*'

Aquando da abertura do programa, surge a janela associada ao modo '*Initial Menu'*, onde o utilizador tem um conjunto de botões interativos com as funcionalidades apresentadas na Figura 7.

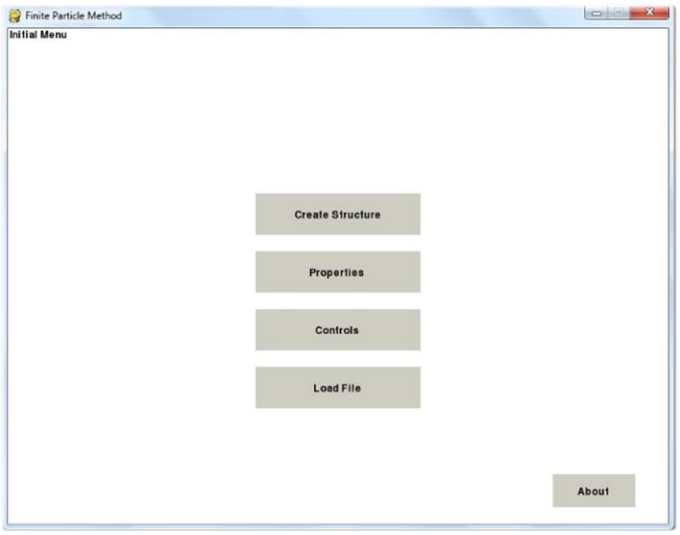


Figura 7: Modo menu inicial (ilustrativo).

Neste menu, o utilizador pode executar as seguintes funções de acordo com os respetivos botões:

Create Structure: *Pause mode* - O utilizador é enviado para o modo pausa onde pode criar uma estrutura e iniciar a sua simulação.

***Properties*:** *Properties mode* - Acessa ao menu de propriedades do ambiente, onde é possível alterar o tipo de análise pretendida e introduzir cargas e acelerações de ambiente, entre outras características. Estas serão aplicadas a todas as partículas criadas podendo ser utilizadas para simular condições exteriores, como por exemplo vento e gravidade.

***Controls*:** *Controls mode* - Consulta de todos os comandos existentes do programa, organizados por modo. Esta janela é apenas de consulta.

***Load*:** *Load mode* - Permite carregar ficheiros previamente guardados.

***About*:** *About mode* - Disponibiliza informação geral sobre o programa. Esta janela é apenas de consulta.

Modo '*pause*'

Caso não tenha sido carregada nenhuma estrutura previamente guardada, é necessário aceder ao modo *Pause* para criar uma nova. Neste, é possível executar várias ações tal como apresentadas de seguida:

Criar grelhas: Premir a tecla 'g'.

Alterar entre grelhas criadas: Premir a tecla 'h'

Criar partículas: Premir a tecla 'p', sendo a posição do rato a definir a localização da partícula (o programa tem ajuste automático (*snap*) com as interseções da grelha). É também possível, no menu de propriedades da partícula, alterar a sua posição através da introdução de dígitos por forma a obter uma maior precisão.

Criar barras:	Premir a tecla 'i' em ambas as partículas de extremidade. Novamente é a posição do rato que define a localização de clique.
Selecionar um elemento:	Clicar sobre elemento usando o rato.
Desseleccionar um elemento:	Clicar fora do elemento ou premir a tecla 'Esc'.
Apagar um elemento:	Premir tecla 'delete' depois de selecionar elemento.
Aceder ao menu de propriedades de um elemento:	Selecionar o elemento e premir a tecla 'enter'.
Iniciar ou retomar a simulação:	Premir a tecla 'Esc' sem ter nenhum elemento selecionado.
Voltar ao menu inicial:	Clicar no botão '*Initial Menu*'
Guardar um ficheiro:	Clicar no botão '*Save*'
Mostrar/esconder diagrama de esforço axial:	Premir a tecla 'n' (estrutura '*beam*') ou a tecla 'i' (estrutura '*truss*')
Mostrar/esconder diagrama de esforço transverso:	Premir a tecla 'v' (estrutura '*beam*')
Mostrar/esconder diagrama de momentos fletores:	Premir a tecla 'm' (estrutura '*beam*')
Mostrar/esconder reações:	Premir a tecla 'r' (estrutura '*beam*') ou tecla 'i' (estrutura '*truss*')
Mostrar/esconder forças e momentos exteriores:	Premir a tecla 'f' (estrutura '*beam*')
Exportar informação de um elemento para um ficheiro Excel:	Premir a tecla 'e', tendo o elemento selecionado (partícula ou barra).

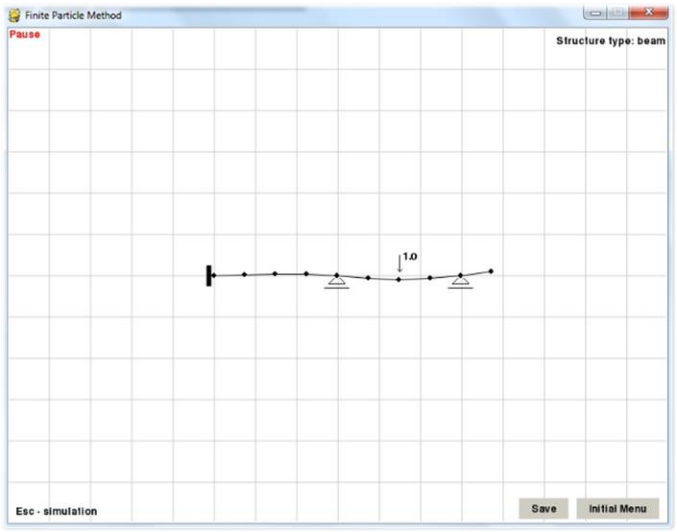


Figura 8: Modo pausa (ilustrativo).

Modo '*simulation*'

Neste modo, é possível acompanhar graficamente o processo da simulação, no entanto, qualquer alteração que se queira introduzir requer que o programa seja colocado no modo pausa.

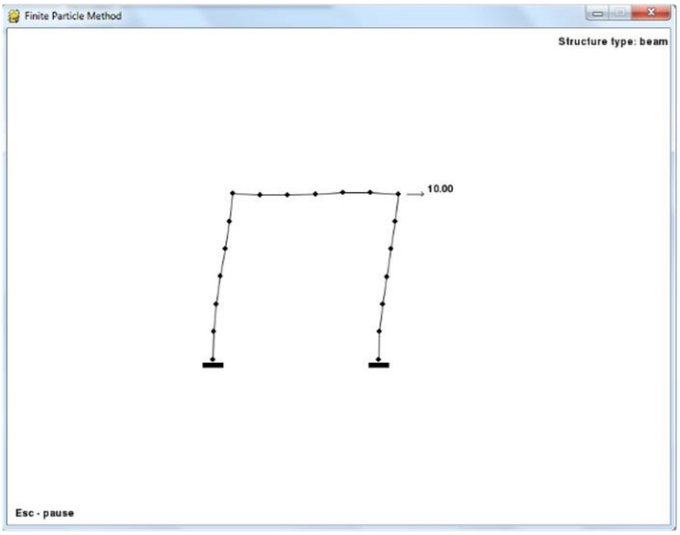


Figura 9: Modo simulação (ilustrativo).

Modo '*properties*'

Estando no modo pausa, é possível aceder a qualquer um dos menus propriedades dos vários elementos, podendo o menu de propriedades do ambiente também ser acedido pelo menu inicial, como já referido. No caso de partículas e barras, a informação disponibilizada depende do tipo de estrutura que se está a avaliar ('*beam*' ou '*truss*'), pois assim evitam-se redundâncias e oferece-se uma melhor organização ao utilizador. Na Figura 10, é apresentado o menu de propriedades das partículas como representativo, sendo os menus de propriedades dos restantes elementos (barras, grelha e ambiente) apenas diferentes nas opções disponíveis. O menu refere-se ao tipo de estrutura '*beam*', a que foi desenvolvida na presente dissertação.

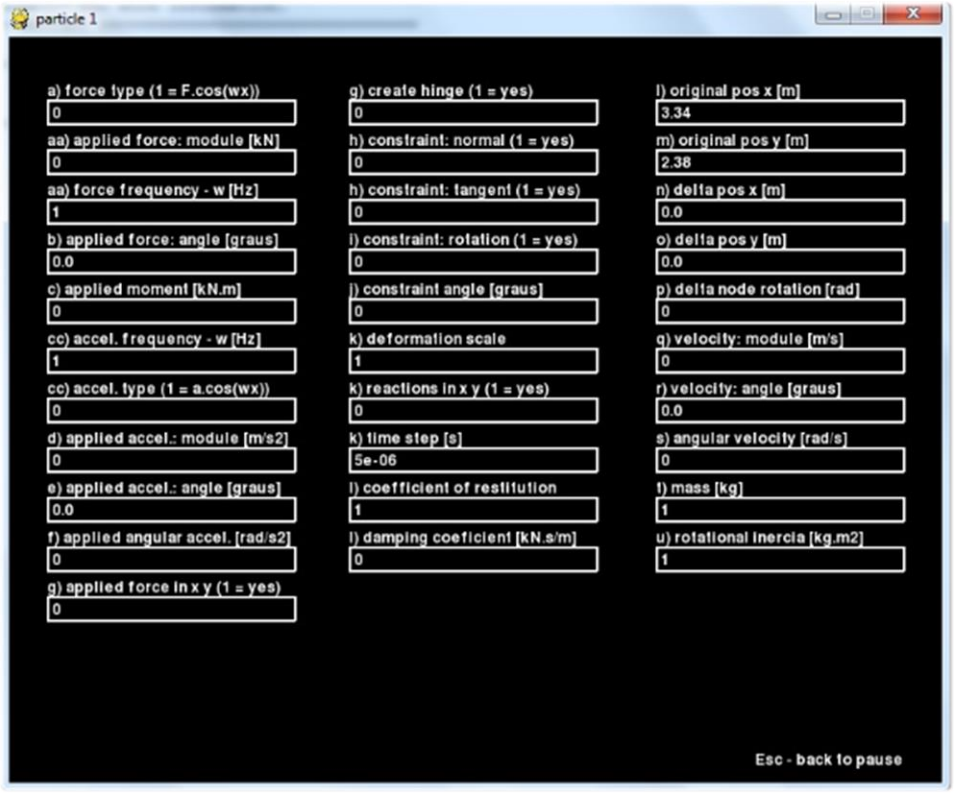


Figura 10: Menus de propriedades das partículas (ilustrativo).

Neste programa, os caracteres permitidos estão sujeitos a algumas restrições e seguem a nomenclatura ASCII – *American Standard Code for Information Interchange*, podendo tal atribuição ser consultada em (Lopes 2015).

Em termos de introdução de informação, apenas são aceites caracteres correspondentes a números, ao sinal de subtração e ao ponto decimal. Existe apenas uma propriedade nas propriedades do ambiente, '*Structure type*', na qual não são aceites quaisquer caracteres além das duas palavras '*beam*' e '*truss*'.

Modos '*save*' e '*load*'

Num programa existe sempre a necessidade de poder guardar o trabalho feito, quer seja para consultar no futuro ou para continuar a trabalhar nele mais tarde. Para tal foram implementados os modos '*save*' e '*load*' que correspondem a guardar e carregar ficheiros, respetivamente (Figura 11).

O modo '*save*' tem, no entanto, uma particularidade. Quando é guardado um ficheiro, é guardada a informação nesse preciso momento, isto é, caso esteja a decorrer uma simulação e se guarde um ficheiro, quando se volta a carregá-lo, utilizando o modo '*load*', este é aberto no instante em que foi guardado, em termos de deformada e valores alterados pela simulação. Isto significa que quando queremos voltar a analisar a mesma estrutura, não é possível recuperar o seu estado inicial, a não ser que o utilizador guarde um ficheiro no instante anterior ao inicio da simulação. Fazendo isso, sempre que tenha decorrido parte ou a totalidade de uma simulação, é possível carregar o ficheiro com a estrutura no instante inicial, fazer alterações, se desejado, e correr uma nova simulação.

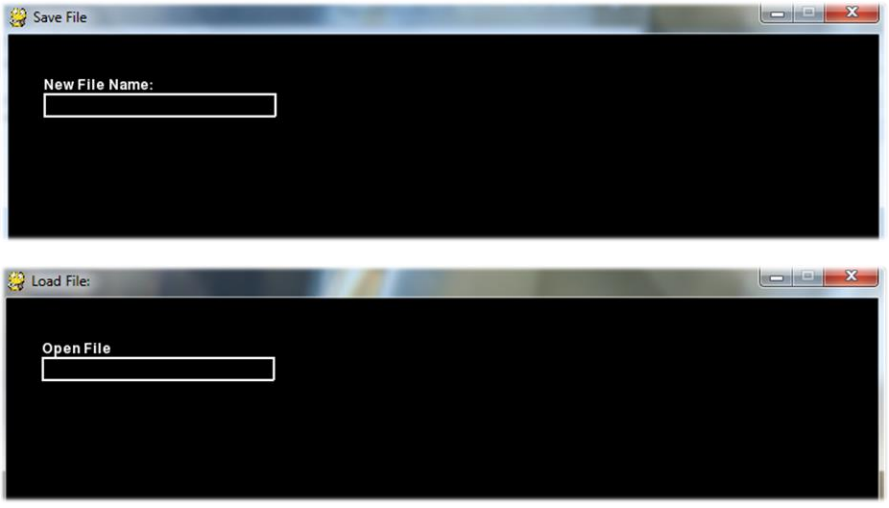


Figura 11: a) Modo guardar; b) Modo carregar (ilustrativo).

Para evitar que o utilizador feche o programa sem ter guardado as alterações e perca informação, foi criada uma mensagem sempre que se clique no ícone de fechar da janela do programa. Esta permite aceder diretamente ao modo '*save*' ou fechar o programa (Figura 12). Assim, a possibilidade perder informação é mais reduzida.

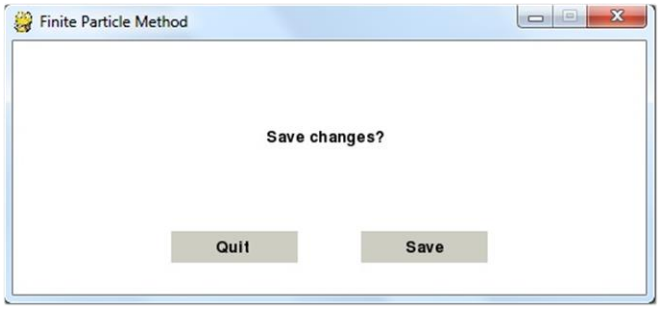


Figura 12: Mensagem de aviso para guardar ficheiro (ilustrativo).

Modo '*controls*'

Neste modo, o utilizador tem acesso a todos os comandos existentes no programa, com a sua respetiva funcionalidade para ambos os tipos de estrutura. Desta forma, em qualquer instante, é possível recorrer a este modo para aprender ou relembrar alguma funcionalidade, evitando a necessidade de recorrer a fontes externas ao programa.

Sendo a única forma de aceder a este modo através do menu inicial, o programa está implementado de forma a que não seja perdida nenhuma informação devido à navegação entre modos, tal como já referido, ou seja, se o utilizador estiver, por exemplo, a meio de uma simulação e quiser rever algum comando, basta colocar a simulação em pausa, aceder ao menu inicial, consultar o que pretende e voltar à simulação, sem qualquer preocupação de alteração do estado da simulação.

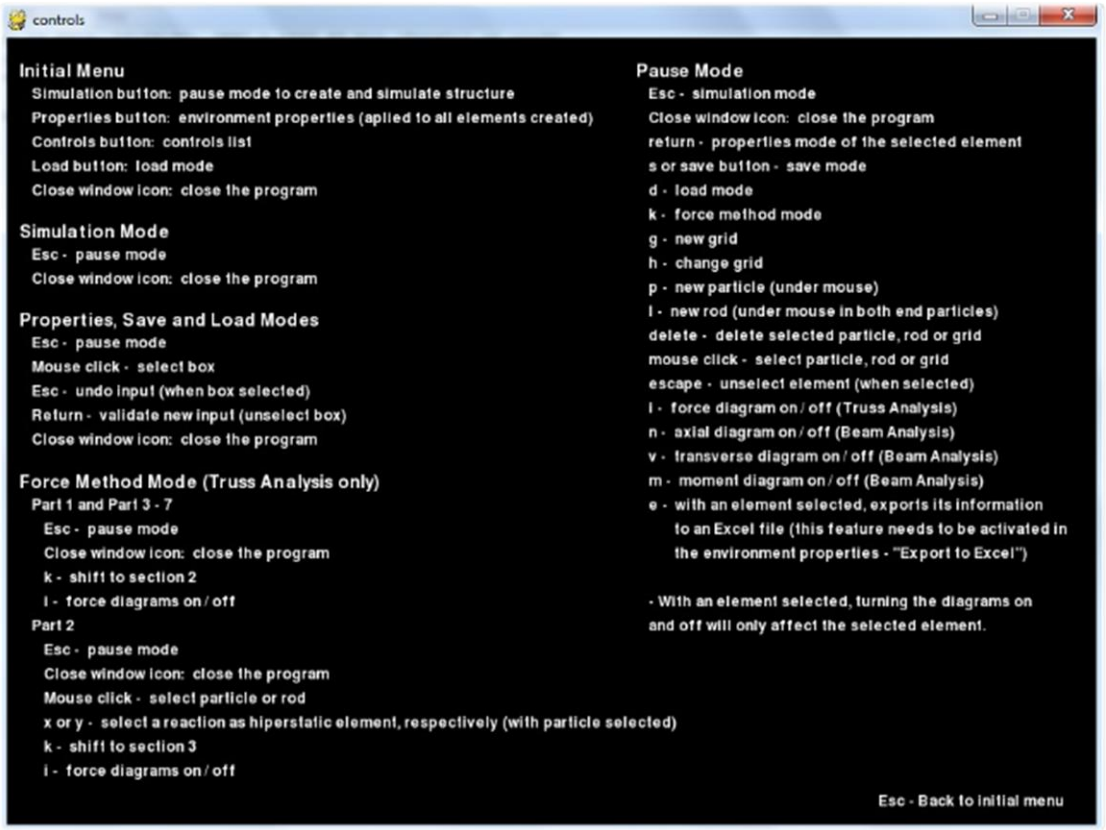


Figura 13: Modo comandos (ilustrativo).

3 Análise do Movimento de Estruturas

Neste capítulo será apresentada a formulação física na qual este trabalho foi baseado. Primeiro será apresentada uma breve introdução aos métodos discretos e sua necessidade. Numa segunda seção será exposto o método base utilizado no âmbito deste trabalho – o Método de Partículas Finitas (MPF), aplicado a elementos que possam apresentar esforços à flexão. Por último será apresentado a implementação computacional do método.

3.1 Métodos discretos

Um dos objetivos da área da Engenharia de Estruturas é prever o comportamento de estruturas sob o efeito das mais variadas ações, isto é, saber quais as distribuições de deformações, deslocamentos, rotações, tensões e esforços tanto em situações estáticas como dinâmicas. Para tentar simular os vários fenómenos físicos que influenciam o comportamento das estruturas têm vindo a ser estudados problemas de mecânica estrutural com recurso a um conjunto alargado de métodos.

O método mais usado, de grande aplicabilidade, é o Método dos Elementos Finitos (MEF). Existem muitas variantes deste método, mas de uma forma simplista pode dizer-se que se trata do mesmo. O MEF pode ser usado, em problemas de análise estrutural, para estudar problemas lineares ou não lineares, estáticos ou dinâmicos, sendo uma técnica suficientemente robusta para permitir a análise de vários problemas.

Contudo, em problemas geometricamente não lineares ou com grandes deformações, ou problemas de fendilhação, por exemplo, podem surgir descontinuidades na malha, o que pode criar algumas dificuldades na análise tradicional por elementos finitos. Nesses casos, a estratégia passa por modificar os algoritmos (exemplos de tal modificação são as formulações *total Lagrangian* (TL) e *updated Lagrangian* (UL)). Embora tenha sido comprovada a sua aplicabilidade em problemas de análise de grandes deslocamentos e rotações ((Bathe e Bolourchi 1979) entre outros), a interpretação física de análises geometricamente não lineares continua a ser uma área que requer mais atenção.

Ao longo dos anos, e com objetivo de colmatar alguns aspetos onde o MEF pode ser menos vantajoso, têm vindo a ser desenvolvidos outros métodos numéricos de discretização entre os quais aqueles que não usam uma malha na definição do domínio de estudo (*Meshfree methods*), ou seja, o domínio do problema é essencialmente definido apenas por um conjunto de "nós" relacionados entre si por funções de aproximação. Um dos primeiros *meshfree methods* a ser desenvolvido foi o *Smoothed Particle Hydrodynamics (SPH),* criado por Gingold e Monaghan [1977], que já foi várias vezes adaptado aos mais variados problemas, desde dinâmica de fluidos, mecânica dos solos, condução de calor, entre outros (Asprone, Auricchio et al. 2014). Depois deste, vários outros métodos sem malha têm vindo a ser apresentados.

Quando os "nós" atrás referidos adquirem o carácter de "partículas" pode falar-se de "métodos de partículas". Apresentam algumas semelhanças com os "*meshfree methods*", mas não são propriamente "*meshfree methods*". Nos métodos de partículas não se recorre a funções de aproximação, a ligação entre partículas é feita por elementos de ligação com comportamentos semelhantes ao de molas. Estes

elementos de ligação têm o intuito de influenciar o comportamento das partículas, cada uma a ser influenciada e influenciar outras. Métodos deste tipo, por serem formulações baseadas em partículas/massas, têm-se revelado bastante úteis na simulação de problemas dinâmicos de mecânica estrutural que envolvam grandes deformações/deslocamentos. A sua simplicidade resulta do facto de se basearem em partículas e na aplicação direta das leis de Newton. Nesse conjunto de métodos inclui-se o método utilizado neste trabalho e cuja formulação será apresentada em detalhe na seção seguinte.

3.2 Método das Partículas Finitas (MPF)

Este método é baseado principalmente nas partículas e em equações de equilíbrio dinâmico (que são equações vetoriais, isto é, escritas para cada componente espacial) tendo em conta as forças que lhe são aplicadas. As estruturas são representadas apenas por partículas e suas interações são feitas através de elementos de ligação.

Tais interações são avaliadas usando a metodologia que, na referência (Ting, Shih et al. 2004), (descrito em detalhe na seção 3.2.3), toma a designação VFIFE (*vector form intrinsic finite element method*). Cada partícula move-se em função da aplicação das leis de Newton e utilizando a integração no tempo, por via numérica e de forma explícita, das acelerações, obtêm-se valores de velocidade e, de seguida, de posições das partículas. Uma outra característica deste método é não formular explicitamente o equilíbrio total da estrutura, mas sim apenas o equilíbrio de cada partícula com base na força interna e externa nela aplicados. O equilíbrio global verifica-se então como consequência dos equilíbrios locais nas partículas.

A organização deste subcapítulo será feita de acordo com as etapas deste método: definição das partículas, discretização do seu movimento e equações utilizadas, cálculo de deformações e esforços e integração no tempo (Wu, Tsai et al. 2009). Será ainda apresentada uma seção sobre relaxação dinâmica (Liew, Mele et al. 2016) como complemento à explicação do método.

3.2.1 Definição das partículas

No método em estudo, tal como referido anteriormente, uma estrutura é representada por um conjunto de partículas e elementos de ligação entre elas. As partículas têm massa e estão sujeitas a forças exteriores e a forças interiores que lhes são transmitidas pelos elementos de ligação, pelas barras. Tais forças interiores resultam de deformações nas barras em função dos deslocamentos que as partículas de extremidade têm. Ao longo do tempo, embora haja atualização de vários valores, as propriedades das partículas e das barras não se alteram.

Considere-se o movimento de uma estrutura com base nas posições das partículas ao longo do tempo, como representado na Figura 14. Tendo em conta a massa da partícula β no instante t_N , m_β, e o vetor de posição $\vec{X}^T(t_N) = [x(t_N) \quad y(t_N) \quad \theta(t_N)]$, as interações entre partículas adjacentes são materializadas pelas barras.

As barras, devido aos movimentos das partículas, vão sofrer deformações que originam esforços internos em equilíbrio. Tais esforços, ou melhor, forças de igual intensidade e sinal contrário, são aplicados diretamente nas partículas na forma de forças nodais equivalentes, que, em conjunto com o

carregamento exterior, definem o movimento da estrutura com base na segunda lei de Newton (aplicada quer às componentes de translação, quer à componente de rotação). Salienta-se que o carregamento exterior é aplicado diretamente nas partículas. Caso se queira simular carregamento nas barras, este terá de ser aplicado nas partículas sob a forma de forças nodais equivalentes.

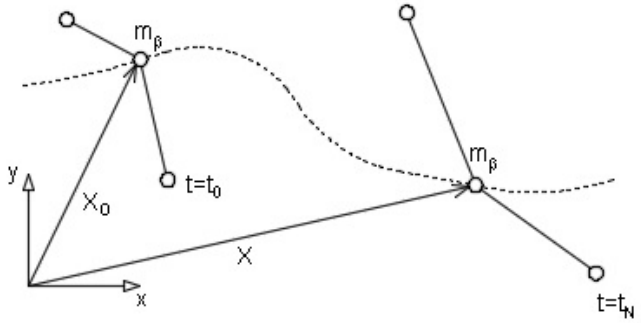


Figura 14: Representação do movimento de uma estrutura constituída por partículas e elementos de ligação (adaptado de (Wu, Tsai et al. 2009)).

Os cálculos da massa e do momento de inércia de massa (segundo o eixo perpendicular ao plano, ou seja, o momento polar de inércia) de uma partícula arbitrária β são definidos como,

$$m_\beta = M_\beta + \sum_{k=1}^{nc} m_{k\beta} \tag{3}$$

$$j_\beta = J_\beta + \sum_{k=1}^{nc} j_{k\beta} \tag{4}$$

onde M_β e J_β são, respetivamente, a massa e o momento de inércia de massa associados à partícula β; nc é o número de barras ligadas à partícula; $m_{k\beta}$ e $j_{k\beta}$ são as contribuições de massa e de momento de inércia de massa das barras conectadas à partícula, respetivamente, cujas expressões são as seguintes:

$$m_{k\beta} = \frac{1}{2}\lambda_k l_k \tag{5}$$

$$j = \frac{m_{k\beta} l_k^2}{12} \tag{6}$$

onde λ_k e l_k são, respetivamente, a densidade por unidade de comprimento e o comprimento do elemento k, e i é o raio de giração da barra representada pelo elemento k (Wu, Tsai et al. 2009).

3.2.2 Discretização do movimento e equações

Admitindo uma partícula arbitrária que se desloca entre os instantes t_0 e t com vetores de posição $\vec{X}_0$ e $\vec{X}$, respetivamente, pode ser observada a discretização do seu movimento na Figura 15.

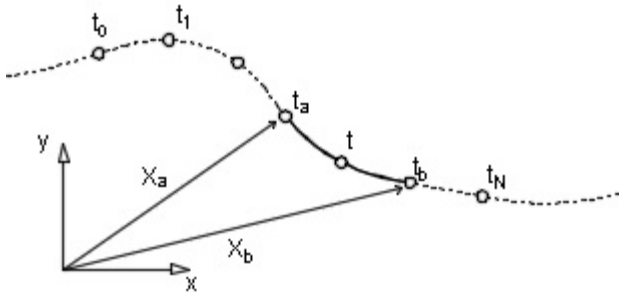


Figura 15: Discretização do movimento de uma partícula (adaptado de(Wu, Tsai et al. 2009)).

A discretização apresentada é materializada pela divisão do movimento em vários segmentos discretos com base nos incrementos no tempo, onde $t_0 < t_1 < \cdots < t_N$. Cada avanço no tempo corresponde a um destes segmentos ($t_a \leq t \leq t_b$), nos quais a estrutura é analisada com base na geometria em t_a e t_b . São então calculadas as deformações e esforços internos (seção 3.2.3) com base nos deslocamentos das partículas desde a geometria em t_a até à geometria em t_b. Por esta razão, todo o percurso da estrutura desde t_0 até t_a pode ser "esquecido" pois já foi tratado anteriormente.

Relativamente ao movimento de cada uma das partículas, este surge do cálculo de acelerações em função da segunda lei de Newton, nas componentes de translação e rotação:

$$\begin{gathered} \vec{F} = m\vec{a} \\ m\ddot{\vec{X}} = \vec{P} + \sum_{i=1}^{nc} \vec{p_i} - \sum_{i=1}^{nc} \vec{f_i} \ , \qquad t_a \leq t \leq t_b \\ j\ddot{\vec{\theta}} = \vec{Q} + \sum_{i=1}^{nc} \vec{q}_{iz} - \sum_{i=1}^{nc} \vec{m}_{iz} \ , \qquad t_a \leq t \leq t_b \end{gathered} \tag{7}$$

ou escrito na forma matricial,

$$\begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & j \end{bmatrix} \begin{Bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{\theta} \end{Bmatrix} = \begin{Bmatrix} P_x \\ P_y \\ Q_z \end{Bmatrix} + \sum_{i=1}^{nc} \begin{Bmatrix} p_{ix} \\ p_{iy} \\ q_{iz} \end{Bmatrix} - \sum_{i=1}^{nc} \begin{Bmatrix} f_{ix} \\ f_{iy} \\ m_{iz} \end{Bmatrix}, \qquad t_a \leq t \leq t_b \tag{8}$$

onde m e j são, respetivamente, a massa e momento de inércia; $\ddot{\vec{X}}$ é o vetor de aceleração no plano de eixos global (x, y), contendo as acelerações nas direções x, y, e a aceleração angular $\ddot{\theta}_z$ em torno do eixo z; P e p são, respetivamente, as forças exteriores nas direções x e y aplicados diretamente nas partículas e nas barras; Q e q são, respetivamente, os momentos em torno do eixo z, aplicados diretamente nas partículas e barras; f e m são, respetivamente, as forças e momentos internos resultantes das deformações; nc é o número de elementos ligados à partícula.

Em ambas as equações (7) e (8) são necessários parâmetros iniciais. Estes são introduzidos pelo utilizador na primeira iteração, sendo que nas restantes corresponde aos valores em t_a.

3.2.3 Cinemática e Esforços internos

Continua-se, nesta seção, a descrever o método utilizado como base nesta dissertação*, vector form intrinsic finite element* (VFIFE), para avaliar a interação entre partículas através dos seus elementos de ligação, as barras (Ting, Shih et al. 2004).

O método VFIFE já mostrou a sua aplicabilidade de forma bem sucedida em vários tipos de problemas, tais como: *motion analysis of planar framed structures* (Wu, Wang et al. 2006), *kinematically indeterminate bar assemblies* (Yu e Luo 2009a), *three-dimensional (3D) membrane structures* (Wu e Ting 2008) e *progressive Failure Simulation of Truss Structures* (Yu, Paulino et al. 2013). Dessa forma, e sendo o tema desta dissertação abrangido por tal formulação, foi assim escolhido este método como referência e base deste trabalho.

Como foi explicado anteriormente, o movimento das partículas gera esforços internos nas barras, que serão aplicados nas partículas através de forças nodais equivalentes.

Coordenadas de deformação

Para avaliar as deformações de uma barra entre duas partículas, é definido, em todos os intervalos temporais, um vetor com duas componentes de deslocamento, $\vec{u} = \{du \quad dv\}$, representativas do deslocamento relativo em x e y, respetivamente. Estas componentes são referidas num sistema de eixos local do elemento, por forma a simplificar os cálculos. Posteriormente, é realizada a transformação para o sistema de eixos global, onde é definido o movimento das partículas usando as equações (7).

Nesta formulação, é de interesse separar as componentes de corpo rígido dos deslocamentos do elemento. Para que o sistema de eixos local respeite os movimentos de corpo rígido, duas quaisquer partículas do elemento têm de cumprir um total de três condições arbitrárias. Nesta formulação (Figura 16) foram escolhidas as seguintes condições:

$$\begin{Bmatrix} du \\ dv \end{Bmatrix} = \begin{Bmatrix} u - u_c \\ v - v_c \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}, em\ C \tag{9}$$

$$dv'^{d} = 0, em\ D \tag{10}$$

onde C e D são dois pontos arbitrários do elemento; u e u_c são, respetivamente, os deslocamentos totais em relação ao eixo global do ponto C nas posições final e inicial do espaço de tempo e dv'^{d} o deslocamento relativo na direção y'.

Embora os pontos C e D pudessem ter sido outros quaisquer do elemento, optou-se por manter a formulação mais generalizada. No entanto, como neste trabalho cada elemento de ligação analisado apenas tem duas partículas de extremidade, a formulação passará a ser referida às mesmas.

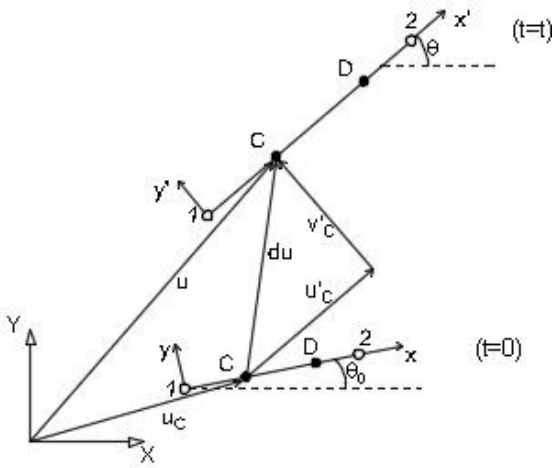


Figura 16: Coordenadas de deformação (adaptado de (Ting, Shih et al. 2004)).

Para simplificar o processo de cálculo (diminuir o número de deslocamentos independentes) e por possíveis erros devido a grandes diferenças de magnitude de valores, admite-se que alguns dos deslocamentos são de corpo rígido. Tal consideração permite tirar partido do facto dos deslocamentos de corpo rígido não gerarem deformações, levando à eliminação dos mesmos.

Assim, e de acordo com a Figura 16, onde θ_0 é o ângulo do elemento no instante de tempo zero (t_0) e θ no instante (t), calculado por $\theta = \theta_0 + \Delta\theta$, pode-se eliminar o deslocamento do ponto 1 como sendo uma translação de corpo rígido, passando o vetor de deslocamento relativo do ponto 2 a ser:

$$d\vec{u}_2 = \vec{u}_2 - \vec{u}_1 \tag{11}$$

sendo $\Delta\theta$ o ângulo entre ambos os eixos locais do elemento nos instantes t_0 e t (Figura 17) e admitindo tal valor como sendo uma rotação de corpo rígido (du_2^r), pode-se calcular a deformação do elemento como:

$$d\vec{u}_2^d = d\vec{u}_2 - d\vec{u}_2^r \tag{12}$$

É de notar que são verificadas as condições dadas pelas equações (9) e (10),

$$\begin{Bmatrix} du \\ dv \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}, em\ 1 \tag{13}$$

$$\begin{Bmatrix} du \\ dv \end{Bmatrix} = \begin{Bmatrix} du'^d_2 \\ 0 \end{Bmatrix}, em\ 2 \tag{14}$$

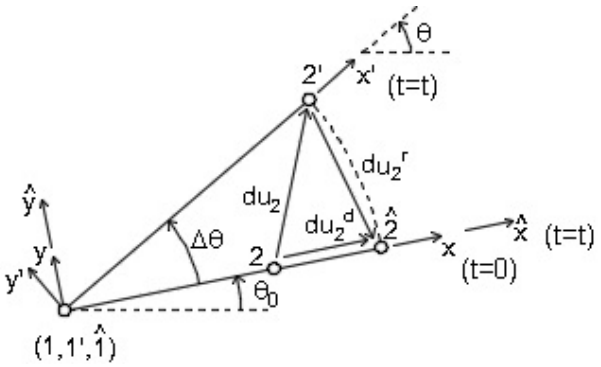


Figura 17: Deformação relativa dos pontos 1 e 2 devido aos movimentos de corpo rígido (adaptado de (Ting, Shih et al. 2004)).

Componentes de deformação

Caso fosse tida em conta a variação de curvatura do elemento devido ao deslocamento, o valor de rotação de corpo rígido, na Figura 17, poderia ser diferente do acréscimo de rotação ($\Delta\theta$) do eixo do elemento, e, com isso, a deformação resultante passaria a ter um valor diferente de zero em ambas as componentes u e v do ponto 2 (equação (14)).

Foi assim adotada uma simplificação que consiste na suposição de que o deslocamento transverso é tão pequeno que os efeitos da curvatura podem ser omitidos, (Ting, Shih et al. 2004). Desta forma, admite-se que o elemento se mantem sempre reto. Todavia, tal simplificação não significa que não existam deformações por rotação, pois, como será descrito de seguida, são calculados momentos fletores devidos às rotações das partículas.

O cálculo das deformações pode ser feito em relação ao instante zero (Figura 17) ou em relação ao instante imediatamente anterior (Figura 18). As fórmulas de cálculo são as mesmas, apenas com diferença nos valores (devido a diferentes geometrias de referência). Por isso, até ao final da seção serão apresentadas expressões relativas ao instante anterior.

Calculando os deslocamentos relativos com base no ponto 1 (como mostrado anteriormente) e conhecendo as posições dos pontos 1 e 2, conseguem-se obter os ângulos de orientação dos eixos nas posições inicial e final.

Com aplicação de um incremento no tempo, Δt ($t' = t + \Delta t$), surgem deslocamentos que originarão as deformações da iteração atual. Aplicando o mesmo raciocínio de cálculo, onde eliminamos parte dos deslocamentos como sendo de corpo rígido (Figura 18), obtém-se a deformação incremental do elemento,

$$\underline{R} = \begin{bmatrix} \cos(\Delta\theta) & sen(\Delta\theta) \\ -sen(\Delta\theta) & \cos(\Delta\theta) \end{bmatrix}, \qquad \underline{Q} = \begin{bmatrix} \cos(\theta) & sen(\theta) \\ -sen(\theta) & \cos(\theta) \end{bmatrix}$$

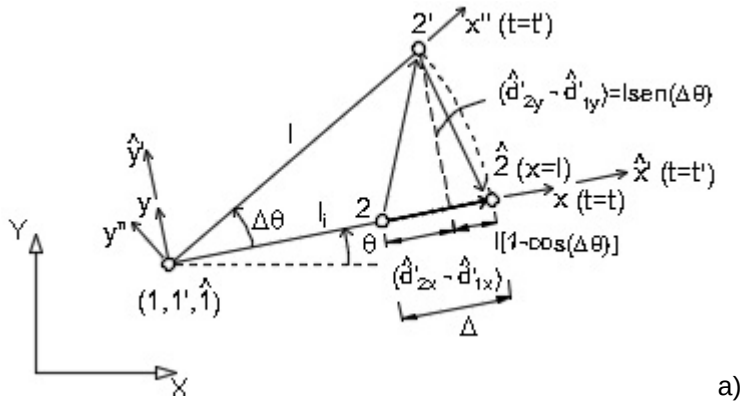
$$\begin{Bmatrix} x'' \\ 0 \end{Bmatrix} = \underline{Q}'' \begin{Bmatrix} x - x_1 \\ y - y_1 \end{Bmatrix}, \qquad \underline{Q}'' = \underline{R}\underline{Q}$$

$$d\hat{u}' = \begin{Bmatrix} \hat{u}' \\ \hat{v}' \end{Bmatrix} = \underline{Q} \begin{Bmatrix} u - u_1 \\ v - v_1 \end{Bmatrix}$$

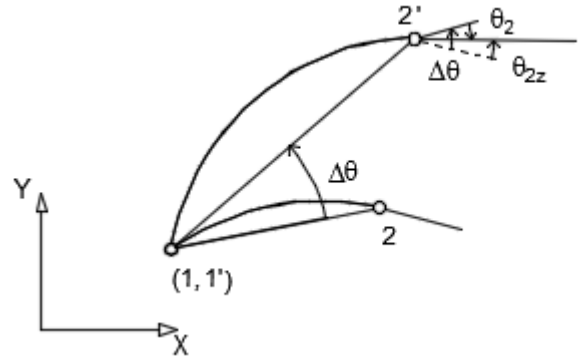
$$d\hat{u}'^r = \begin{Bmatrix} \hat{u}'^r \\ \hat{v}'^r \end{Bmatrix} = (\underline{R}^T - \underline{I}) \begin{Bmatrix} x'' \\ 0 \end{Bmatrix} \tag{15}$$

$$d\hat{u}'^d = \begin{Bmatrix} \hat{u}'^d \\ \hat{v}'^d \end{Bmatrix} = d\hat{u}' - d\hat{u}'^r = \begin{Bmatrix} \hat{u}' \\ \hat{v}' \end{Bmatrix} - \begin{Bmatrix} \hat{u}'^r \\ \hat{v}'^r \end{Bmatrix} \tag{16}$$

onde $d\hat{u}'$ é o deslocamento com base no ponto 1; $d\hat{u}'^r$ é deslocamento de corpo rígido; $\underline{R}$, $\underline{Q}''$ e $\underline{Q}$ são as matrizes de rotação dos respetivos ângulos e $\underline{I}$ a matriz identidade.



a)



b)

Figura 18: Representação detalhada das deformações relativas dos pontos 1 e 2 (adaptado de (Ting, Shih et al. 2004)). a) deformação axial. b) deformação de rotação.

Tal como identificado na Figura 18, os deslocamentos relativos das partículas de extremidade, com subtração dos deslocamentos de corpo rígido, são dados por,

$$\hat{d}'^{d}_{1x} = \hat{d}'^{d}_{1y} = 0$$

$$\hat{d}'^{d}_{2y} = \hat{d}'_{2y} - \hat{d}'^{r}_{2y} = \hat{d}'_{2y} - [\hat{d}'_{1y} + lsen(\Delta\theta)] = 0 \tag{17}$$

$$\hat{d}'^{d}_{2x} = \hat{d}'_{2x} - \hat{d}'^{r}_{2x} = \hat{d}'_{2x} - (\hat{d}'_{1x} - l[1 - cos(\Delta\theta)])$$

De acordo com as equações (17), observa-se que apenas três deslocamentos são independentes, sendo eles os deslocamentos de corpo rígido do elemento, $\hat{d}'_{1x} = \hat{d}'^{d}_{1x}$, $\hat{d}'_{1y} = \hat{d}'^{d}_{1y}$ e $\Delta\theta$. Contudo, o objetivo é obter as deformações independentes, para com estas calcular, posteriormente, os esforços internos. Para tal, começa-se por admitir um vetor com os seis deslocamentos nodais possíveis das partículas 1 e 2, sendo descrito como,

$$(\overrightarrow{\hat{d}'_e})^T = \{\hat{d}'_{1x} \quad \hat{d}'_{1y} \quad \theta_{1z} \quad \hat{d}'_{2x} \quad \hat{d}'_{2x} \quad \theta_{2z}\}$$

De forma análoga, procede-se à separação dos deslocamentos nas suas componentes de deslocamento de corpo rígido e deslocamento relativo.

$$\overrightarrow{\hat{d}'_e} = \overrightarrow{\hat{d}'_e}^{\,r} + \overrightarrow{\hat{d}'_e}^{\,d} \tag{18}$$

$$(\overrightarrow{\hat{d}'_e}^{\,r})^T = \{\hat{d}'_{1x} \quad \hat{d}'_{1y} \quad \Delta\theta \quad \hat{d}'^{r}_{2x} \quad \hat{d}'^{r}_{2y} \quad \Delta\theta\} \tag{19}$$

$$(\overrightarrow{\hat{d}'_e}^{\,d})^T = \{0 \quad 0 \quad \theta_1 \quad \Delta \quad 0 \quad \theta_2\} \tag{20}$$

onde $\overrightarrow{\hat{d}'_e}^{\,r}$ e $\overrightarrow{\hat{d}'_e}^{\,d}$ correspondem, respetivamente, aos vetores de deslocamentos de corpo rígido e relativos. As expressões de $\hat{d}'^{r}_{2x}$ e $\hat{d}'^{r}_{2y}$ estão representadas nas equações (17).

Utilizando as equações (17) a (20) obtêm-se assim as expressões dos termos do vetor de deslocamentos relativos correspondentes às deformações independentes e o respetivo vetor das deformações independentes $\overrightarrow{\hat{d}^*_e}$,

$$\Delta = \hat{d}'_{2x} - (\hat{d}'_{1x} - l[1 - cos(\Delta\theta)])$$

$$\theta_1 = \theta_{1z} - \Delta\theta \; ; \qquad \theta_2 = \theta_{2z} - \Delta\theta$$

$$\overrightarrow{\hat{d}^*_e} = \begin{Bmatrix} \Delta \\ \theta_1 \\ \theta_2 \end{Bmatrix} \tag{21}$$

Esforços internos

Como o intervalo de tempo utilizado é muito pequeno (seção 3.2.4), as variáveis tensão e deformação podem ser consideradas fixas durante a iteração e apenas serem alteradas no final da mesma. Pela mesma razão, os esforços podem ser calculados com base nestes valores de tensão e deformação (Yu, Paulino et al. 2013). Para deduzir as expressões dos esforços internos são necessárias funções de forma que aproximem as tensões e deformações ao longo do elemento.

Dos seis deslocamentos nodais possíveis, tal como referido anteriormente, apenas são necessários três para definir as deformações do elemento. Tendo isso em atenção, recorre-se a funções de forma polinomiais, de modo a simplificar o processo de cálculo. Também se verifica a imposição de condições de compatibilidade de acordo com as deformações independentes. A dedução encontra-se descrita de seguida.

$$u^d_m = a_1 + a_2 x \tag{22}$$

$$v^d = a_3 + a_4 x + a_5 x^2 + a_6 x^2 \tag{23}$$

$$u^d = u^d_m - y\frac{dv^d}{dx} \tag{24}$$

onde u^d e v^d são, respetivamente, as deformações longitudinal e transversal, x é o eixo local da barra, em torno do qual vão ser calculados os esforços internos e a_i são os coeficientes das funções polinomiais. A expressão de u^d apresenta-se de acordo com o modelo de viga de Euler-Bernoulli.

Ambas as partículas de extremidade têm de respeitar as seguintes condições:

$$\begin{aligned} &x = 0\,, \quad u^d_m = 0\,, \quad {dv^d}/{dx} = \theta_1\,, \quad v^d = 0 \\ &x = l\,, \quad u^d_m = \Delta\,, \quad {dv^d}/{dx} = \theta_2\,, \quad v^d = 0 \end{aligned} \tag{25}$$

Admitindo que $s = x/l$, de modo a simplificar a escrita e para facilitar a leitura, obtêm-se as expressões das equações (22), (23) e (24).

$$u_m^d = s\Delta$$

$$v^d = (s - 2s^2 + s^3)\, l\, \theta_1 + (-s^2 + s^3)\, l\, \theta_2 \tag{26}$$

$$u^d = s\Delta - [(1 - 4s + 3s^2)\, \theta_1 + (-2s + 3s^2)\, \theta_2]\, y$$

Com base no princípio dos trabalhos virtuais, são obtidos os esforços independentes, $\vec{f}_e^*$, correspondentes às deformações relativas independentes do elemento barra.

$$\delta U_e = \int_{V_0} \delta\varepsilon_x^T\, \sigma_x dV_0$$

$$\varepsilon_x = \vec{B}\vec{d}_e^* = \frac{1}{l}\{1 \quad (4-6s)y \quad (2-6s)y\} \begin{Bmatrix} \Delta \\ \theta_1 \\ \theta_2 \end{Bmatrix},\ \vec{B} = \frac{du^d}{ds}$$

$$\delta U_e = \left(\delta\vec{d}_e^*\right)^T \left(\int_{V_0} \vec{B}^T\, \sigma_x dV_0 \right) = \left(\delta\vec{d}_e^*\right)\vec{f}_e^*$$

$$\vec{f}_e^* = \int_{V_0} \vec{B}^T\, \sigma_x dV_0 = \begin{Bmatrix} f_{2x} \\ m_{1z} \\ m_{2z} \end{Bmatrix} \tag{27}$$

Como o método é aplicado sempre em regime elástico, temos que $\sigma_x = E\varepsilon_x = E\vec{B}\vec{d}_e^*$. Substituindo a relação constitutiva anterior na equação (27), obtém-se $\vec{f}_e^*$ na forma matricial,

$$\begin{Bmatrix} f_{2x} \\ m_{1z} \\ m_{2z} \end{Bmatrix} = \begin{bmatrix} EA/l & 0 & 0 \\ 0 & 4EI/l & 2EI/l \\ 0 & 2EI/l & 4EI/l \end{bmatrix} \begin{Bmatrix} \Delta \\ \theta_1 \\ \theta_2 \end{Bmatrix} \tag{28}$$

Da equação (28), tal como referido acima, apenas se obtêm os esforços internos associados às deformações independentes. Os restantes esforços são calculados através do equilíbrio do elemento barra, de forma a obter os seis esforços internos no eixo local do elemento (Figura 19).

$$\sum F_x = 0 \quad \Leftrightarrow \quad f_{1x} = -f_{2x}$$

$$\sum M_1 = 0 \quad \Leftrightarrow \quad f_{2y} = -(m_{1z} + m_{2z})/l \tag{29}$$

$$\sum F_y = 0 \quad \Leftrightarrow \quad f_{1y} = -f_{2y}$$

$$\vec{f}_e^{int} = \begin{Bmatrix} f_{1x} \\ f_{1y} \\ m_{1z} \\ f_{2x} \\ f_{2y} \\ m_{2z} \end{Bmatrix} = \begin{Bmatrix} -EA\Delta/l \\ (6EI/l^2)(\theta_1 + \theta_2) \\ (2EI/l)(2\theta_1 + \theta_2) \\ EA\Delta/l \\ -(6EI/l^2)(\theta_1 + \theta_2) \\ (2EI/l)(\theta_1 + 2\theta_2) \end{Bmatrix} \tag{30}$$

com E representando o módulo de Young (ou de elasticidade), I a inércia da seção em torno do eixo z do elemento barra e A a área da seção do elemento barra.

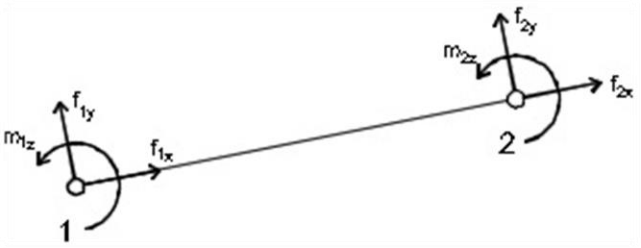


Figura 19: Esforços num elemento barra (adaptado de (Wu, Tsai et al. 2009)).

O presente método evolui com base na análise do comportamento das partículas, sendo necessário aplicar, nas mesmas e como forças interiores, forças nodais equivalentes aos respetivos esforços criados pelas deformações do elemento barra. Como as partículas se localizam num sistema de eixos fixo ao longo do tempo (eixo global) e os esforços interiores são obtidos no eixo local do elemento barra, é necessário proceder-se a uma transformação de coordenadas. Com isso, obtemos os esforços nodais equivalentes que serão aplicados nas partículas correspondentes, tal como descrito de seguida,

$$\vec{f}_e^{int} = \underline{T}^T \, \vec{\hat{f}}_e^{int} \tag{31}$$

$$\underline{T}^T = \begin{bmatrix} Q & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & Q & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \underline{Q} = \begin{bmatrix} cos\theta & sen\theta \\ -sen\theta & cos\theta \end{bmatrix}$$

onde θ é o ângulo entre os eixos global e local.

Nesta dissertação, por forma a eliminar a necessidade de uma transformação de coordenadas, as forças são representadas em coordenadas polares. Desta forma, podem ser referidas sempre ao eixo global.

3.2.4 Integração no tempo

Ao longo deste capítulo foi descrita a forma de obter os termos das equações (7), tratando-se de um processo de cálculo que acontece todo durante o mesmo intervalo de tempo. No entanto, como o método no qual tal processo é aplicado é um método iterativo, tem de existir forma de refletir o incremento temporal no movimento das partículas. Para tal, existem vários métodos de integração explícita no tempo, tais como, por exemplo, a aproximação de segunda ordem através da série de Taylor, a aproximação de quarta ordem do método de Runge-Kutta ou o método de diferença central. Nesta dissertação foi escolhida a aproximação através da série de Taylor, tal como apresentada de seguida.

Nas equações (7), apenas não são conhecidas, no respetivo espaço temporal, as acelerações de translação e rotação. Para obter tais valores, é necessário resolver o sistema de equações (8). Sendo $\vec{X}_N$ o vetor dos três deslocamentos possíveis ($x, y \, e \, \theta$) de uma partícula no instante de tempo N e aplicando a série de Taylor, as componentes da aceleração podem ser aproximadas por:

$$\ddot{\vec{X}}_N = \frac{2}{\Delta t^2}(\vec{X}_{N+1} - \vec{X}_N - \dot{\vec{X}}_N \Delta t) \tag{32}$$

Aplicando a equação (32) ao sistema de equações (8) obtém-se os deslocamentos da partícula no próximo instante de tempo. Com base nestes deslocamentos, todo o processo até então apresentado neste capítulo é repetido para calcular deformações e esforços internos.

$$\vec{X}_{N+1} = \vec{X}_N + \dot{\vec{X}}_N \Delta t + \frac{\Delta t^2 (F^{ext} - F^{int})}{2\underline{M}} \tag{33}$$

A expressão da velocidade é apresentada de seguida, a qual é necessária também no caso de ser tido em conta amortecimento no sistema (seção 3.2.5). Considerando tal amortecimento, as equações (7) poderiam passar a ter mais um termo.

$$\dot{\vec{X}}_{N+1} = \dot{\vec{X}}_N + \ddot{\vec{X}}_N \Delta t = \dot{\vec{X}}_N + \frac{(F^{ext} - F^{int})}{\underline{M}} \Delta t \tag{34}$$

Nas expressões acima apresentadas, os valores de deslocamentos e forças podem ser absolutos ou relativos dependendo da abordagem adotada. Independentemente da escolha, a expressão mantém-se.

Neste trabalho, os valores de Δt são atribuídos pelo utilizador, no entanto, podem ser calculados e calibrados. No subcapítulo seguinte será referida tal opção.

3.2.5 Relaxação dinâmica

Ao longo deste capítulo tem vindo a ser explicado um método discreto que permite analisar estruturas sujeitas a grandes deslocamentos, mas nada foi dito sobre a convergência do método e obtenção da solução pretendida. Neste tipo de problemas pode-se analisar as estruturas de modo a traçar o seu comportamento dinâmico ou a obter uma solução estática. Neste trabalho, um dos objetivos é conseguir que os deslocamentos das diferentes partículas levem a que a estrutura convirja para uma solução estática correspondente à solução elástica do problema. Para tal, utiliza-se o conceito de relaxação dinâmica (Liew, Mele et al. 2016).

Independentemente da geometria inicial da estrutura, a aplicação de cargas exteriores gera movimento com base no desequilíbrio de forças nas partículas. Para obter uma solução estática é preciso estagnar, ao longo de algum tempo, tal movimento e aplicar um critério de convergência (paragem) ao sistema. A relaxação dinâmica consegue concretizar esse objetivo, simulando a existência de amortecimento no sistema com base na dissipação de energia cinética, o que faz com que o movimento da estrutura se traduza em oscilações dinâmicas que vão diminuindo ao longo do tempo.

Recorrer à relaxação dinâmica tem algumas vantagens, tais como a mais fácil replicação de propriedades de materiais e maior estabilidade em simulações longas, algo que noutros métodos pode levar a divergências (Liew, Mele et al. 2016).

Para conseguir obter uma boa solução, estável e rápida, é necessário ter em conta valores de determinadas características do sistema, tais como a massa, viscosidade e intervalo de tempo. Existem

trabalhos em que foram realizados testes para conseguir boas calibrações de parâmetros em função do tipo de amortecimento utilizado, (Papadrakakis 1981) e (Pajand e Hakkak 2006).

Existem duas formas, apresentadas de seguida, através das quais é possível dissipar energia e fazer coincidir o estado estável da estrutura com a solução do respetivo problema estático. As duas formas dizem respeito à relaxação dinâmica, distinguindo-se pelo tipo de amortecimento a que recorrem, ora viscoso ora cinético ((Alamatian 2012), (Rezaiee-Pajanda e Rezaeea 2014) e (Liew, Mele et al. 2016)).

No subcapítulo 4.5 serão aplicados ambos os amortecimentos num caso de estudo e apresentada a influência de ambos nos resultados e comportamento da estrutura.

Amortecimento Viscoso

Para simular este tipo de amortecimento, o sistema de equações (8) pode ser reescrito da seguinte forma compacta:

$$\underline{M}\ddot{\vec{X}}_N + \underline{C}\dot{\vec{X}}_N + \underline{K}\vec{X}_N = \vec{P}, \qquad em\ t = t_N \tag{35}$$

onde $\vec{X}, \dot{\vec{X}}\ e\ \ddot{\vec{X}}$ são, respetivamente, o vetor dos deslocamentos (x, y, θ), velocidades e acelerações; $\underline{M}, \underline{C}$ e $\underline{K}$ as matrizes de massa, amortecimento e rigidez e $\vec{P}$ o vetor das forças exteriores.

Aplicando desta vez o método da diferença central, a solução do sistema de equações (35) passa a ser:

$$\ddot{\vec{X}}_N = \frac{\dot{\vec{X}}_{N+1/2} - \dot{\vec{X}}_{N-1/2}}{\Delta t} \tag{36}$$

Usando as equações (35) e (36), o valor da velocidade pode ser obtido através da seguinte equação.

$$\dot{\vec{X}}_{N+1/2} = \frac{\underline{M}/\Delta t - \underline{C}/2}{\underline{M}/\Delta t + \underline{C}/2}\dot{\vec{X}}_{N-1/2} + \frac{P - \underline{K}\vec{X}}{\underline{M}/\Delta t + \underline{C}/2}$$
$$\dot{\vec{X}}_{N-1/2} = \frac{\vec{X}_N - \vec{X}_{N-1}}{\Delta t} \tag{37}$$

A velocidade, no instante atual, pode então ser obtida como um valor médio.

$$\dot{\vec{X}}_N = \frac{\dot{\vec{X}}_{N-1/2} + \dot{\vec{X}}_{N+1/2}}{2} \tag{38}$$

Sendo a matriz de massa uma matriz diagonal e admitindo que a matriz de amortecimento é proporcional à massa por um coeficiente de amortecimento ($C = cM$), é possível ter as expressões para o processo iterativo.

$$\dot{\vec{X}}_{N+1/2} = \frac{2 - c\Delta t}{2 + c\Delta t}\dot{\vec{X}}_{N-1/2} + \frac{2\Delta t}{2 + c\Delta t}\frac{P - \underline{K}\vec{X}}{\underline{M}} \tag{39}$$

$$\vec{X}_{N+1} = \vec{X}_N + \Delta t\, \vec{\dot{X}}_{N+1/2} \tag{40}$$

Desta forma, já se podem obter soluções para os problemas analisados. Contudo, não basta apenas obter uma solução, mas sim obter a melhor solução possível, e, para tal, calibrar os valores de massa, o coeficiente de viscosidade e o intervalo de tempo, pode melhorar significativamente os resultados. Existem várias estratégias para calibrar tais parâmetros, (Pajand e Hakkak 2006), mas uma vez que não foram utilizadas nesta dissertação, e por uma questão de brevidade, não serão descritas.

Amortecimento Cinético

Nesta abordagem, a estagnação da solução é obtida com base nos valores de energia cinética. Quando é identificado um pico de energia cinética, os valores de velocidade são colocados a zero e o processo iterativo volta a ser repetido (Liew, Mele et al. 2016).

A cada pico de energia cinética detetado, a reiniciação de valores de velocidade da partícula simula a dissipação de energia e o próximo pico passa a corresponder a uma energia cinética menor. Com o avançar do tempo, a estrutura converge para um estado estático.

A expressão utilizada para obter o valor da energia cinética no instante de tempo N é dada por:

$$\vec{U}_N = \frac{1}{2}\underline{M}\vec{\dot{X}}^2 \tag{41}$$

Aplicando este tipo de amortecimento não é necessário o uso de matriz de amortecimento ou coeficientes de amortecimento, permitindo a simplificação das expressões utilizadas. A equação (37) passa então a ser a seguinte:

$$\vec{\dot{X}}_{N+1/2} = \vec{\dot{X}}_{N-1/2} + \frac{\Delta t}{\underline{M}}(P - \underline{K}\vec{X}) \tag{42}$$

Os deslocamentos no instante de tempo seguinte são obtidos pela mesma expressão (40), usada no amortecimento viscoso.

Em termos de expressões, embora haja alguma simplificação, o processo é praticamente idêntico. A grande diferença é que, nesta abordagem, é necessário reconhecer os picos de energia cinética. Para tal, e com base na expressão (41), sempre que entre duas iterações consecutivas o valor da energia cinética diminuir, admite-se que no intervalo de tempo anterior foi atingido o pico. Enquanto não for detetada tal diminuição, o processo de cálculo do método continua. O objetivo é então saber o instante em que tal aconteceu, de modo reiniciar o processo de cálculo partindo desse instante.

Perante uma situação de diminuição da energia cinética, ($\vec{U}_{N-1} < \vec{U}_N > \vec{U}_{N+1}$), no instante $t = t_{N+1}$, significa que o pico de energia ocorreu entre $t = t_{N-1}$ e $t = t_N$. Para simplificar, admite-se a sua ocorrência no instante $t = t_{N-1/2}$. Obtém-se então a expressão para os deslocamentos, (Rezaiee-Pajanda e Rezaeea 2014),

$$\vec{X}_{N-1/2} = \vec{X}_{N+1} - \frac{3}{2}\Delta t\vec{\dot{X}}_{N+1/2} + \frac{\Delta t^2}{2}\frac{P - \underline{K}\vec{X}}{\underline{M}} \tag{43}$$

Os valores de deslocamentos são calculados no instante de pico. Tal acontece, pois aquando da deteção da diminuição de energia, o instante atual era $t = t_{N+1}$ e caso se utilizem os valores de deslocamentos do instante $t = t_{N+1}$, a convergência pode não ser atingida. É necessário então que os valores estejam de acordo com o instante de reiniciação, (Alamatian 2012). Por esta razão, o cálculo de energia cinética é a primeira coisa a ser feita no ciclo iterativo. Assim, se ocorrer a diminuição de energia, não vão ser calculadas deformações ou esforços internos desse instante e a estrutura será "enviada" para o instante de pico.

Sendo a velocidade reposta em zero, $\vec{\dot{X}}_N = 0$, pode ser obtida a seguinte relação com base na expressão (38),

$$\vec{\dot{X}}_{N-1/2} = -\vec{\dot{X}}_{N+1/2} \tag{44}$$

Substituindo a expressão (44) na expressão (42), obtém-se como reiniciar o cálculo de velocidade,

$$\vec{\dot{X}}_{N+1/2} = \frac{1}{2}\frac{\Delta t}{\underline{M}}(P - \underline{K}\vec{X}) \tag{45}$$

Uma explicação mais detalhada do processo de reiniciação e da consideração de valores de pico pode ser consultada em (Alamatian 2012) e (Barnes 1999).

Tal como referido no amortecimento viscoso, a calibração de parâmetros é importante para a convergência da solução. No caso de uma análise com amortecimento cinético, e cujo objetivo seja obter uma solução estática, os valores da matriz de massa e de incremento de tempo são fatores a ter em conta. Neste trabalho não foram necessárias, ou adotadas, calibrações de ambos os valores de massa e incremento de tempo, no entanto, será apresentado, resumidamente, para cada um dos parâmetros uma forma de o fazer.

Na calibração de valores de massa, existem várias estratégias, tais como a teoria de $Gerschgörin$ (Alamatian 2012). Por questões de simplicidade, será apenas apresentada a versão resultante da aplicação da teoria de $Gerschgörin$ a sistemas de um grau de liberdade, equação (46), uma vez que a estrutura geral da expressão é mantida nesta versão simplificada.

$$m = \frac{\Delta t^2}{2}k \tag{46}$$

Caso seja necessário, é também possível calibrar o intervalo de tempo Δt. Para tal, será apresentada uma forma com base numa análise de erro do método de amortecimento cinético (Rezaiee-Pajanda e Rezaeea 2014). Pela extensão da dedução, serão apresentadas apenas as expressões finais.

$$\Delta t_{N+1} = \frac{\left(\frac{2}{\Delta t_N} + \Delta t_N \lambda_1^{M^{-1}K}\right) - \sqrt{(\frac{2}{\Delta t_N} + \Delta t_N \lambda_1^{M^{-1}K})^2 - 4(\frac{1}{\Delta t_N} - \Delta t_N \lambda_1^{M^{-1}K})^2}}{2(\frac{1}{\Delta t_N} - \Delta t_N \lambda_1^{M^{-1}K})^2} \tag{47}$$

$$\lambda_1^{M^{-1}K} = \frac{\vec{X}_N{}^T \vec{F}}{\vec{X}_N{}^T \underline{M} \vec{X}_N}$$

Embora não exista calibração automática do valor de incremento de tempo por parte da simulação, é possível ao utilizador ajustar tal valor ao longo da simulação. É, no entanto, necessário ter em atenção que a atribuição do valor de incremento no tempo não é indiferente. Tal questão será referida ao longo desta dissertação, bem como apresentada em maior detalhe no subcapítulo 4.6.

3.3 Procedimento Computacional

Será aqui apresentado, resumidamente, o procedimento de cálculo aplicado no programa. Tal exposição serve apenas para o leitor ter uma forma de agrupar e encadear a informação até aqui apresentada neste capítulo. Para uma consulta em maior detalhe é apresentado o código computacional em anexo (capítulo 0).

1. Recebe informação inicial inserida pelo utilizador: deslocamentos iniciais, forças exteriores, restrições, etc.
2. Calcula as forças residuais, por equilíbrio de forças nas partículas.
3. Calcula as acelerações resultantes de forças residuais, pela 2ª lei de Newton (seção 3.2.2).
4. Calcula novas posições e velocidades, por integração explícita no tempo (seção 3.2.4).
5. Calcula a energia cinética.
6. Calcula deformações das barras (seção 3.2.3).
7. Calcula esforços interiores e respetivas forças equivalentes nas partículas (seção 3.2.3).
8. Verifica possíveis alterações da informação inserida no programa.
9. Caso a simulação não seja parada, volta ao ponto 2 e repete o procedimento.

4 Resultados e validação

Neste capítulo serão apresentados os casos de estudo analisados nesta dissertação e, através dos quais, serão também expostas funcionalidades do programa, bem como a sua validação por comparação dos resultados obtidos do programa com os resultados exatos.

A organização deste capítulo foi pensada de acordo com as análises nas quais foi aplicado o método, sendo elas: análise estática, análise elasto-plástica incremental e análise da resposta da estrutura a carregamentos variáveis no tempo em osciladores de um grau de liberdade (análise dinâmica).

Nas análises elasto-plástica incremental e dinâmica, existem várias questões que não foram levantadas, pois o foco deste trabalho não foi estudar tais fenómenos ao pormenor, mas sim mostrar que o método em questão, utilizando sempre as mesmas equações, tem capacidade de reconhecer os comportamentos associados aos vários fenómenos. Dessa forma, este programa mostra ser uma ferramenta didática interessante e com potencial para analisar estruturas variadas.

4.1 Comportamento do material

Tal como já referido anteriormente, o movimento de partículas gera deformações nas barras (elementos de ligação), as quais por sua vez geram tensões internas. Consequentemente, surgem esforços internos nas barras que afetam o movimento das partículas.

Na equação (2) ($\sigma = E\,\varepsilon$), é visível uma relação linear entre deformações e tensões, pois o módulo de elasticidade E é constante. No entanto, os materiais não têm um comportamento linear quando sujeitos a carregamentos e, dependendo das suas características intrínsecas, podem ter trajetórias de carregamento bastantes distintas.

Em engenharia, por forma a simplificar os cálculos, são adotados modelos que replicam, de forma aproximada, o comportamento dos materiais tendo em conta as suas capacidades elástica e plástica. Embora, em geral, o modelo elástico seja bastante utilizado para obtenção de esforços na estrutura, o modelo que se aplica à grande maioria dos materiais utilizados em engenharia civil, para o dimensionamento de estruturas, é o que se baseia num regime elasto-plástico, podendo ser caracterizado de forma generalizada pelo comportamento indicado na Figura 20.

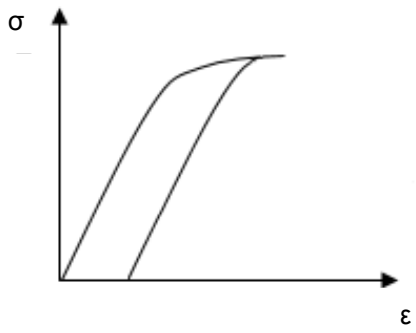


Figura 20: Modelo representativo do comportamento elasto-plástico de um material.

Materiais regidos por este comportamento generalizado têm capacidade elástica até um determinado valor de tensão (tensão de cedência) e até ao qual conseguem recuperar deformações associadas a

cargas e descargas de carregamento. Para esforços acima dos de cedência, os materiais apresentam um comportamento plástico onde surgem deformações não recuperáveis.

Como referido anteriormente, a modelação em concreto de problemas elastoplásticos ou rigidoplásticos não foi um dos objetivos deste trabalho, daí que se tenha optado por definir o comportamento dos materiais sempre segundo as equações associadas ao regime elástico. Para tal, é necessário que a tensão máxima não seja superior à tensão de cedência, ou seja, os elementos não sejam sujeitos a tensões superiores aos de cedência, (Virtuoso 2012).

Simplifica-se assim o modelo da Figura 20 por forma a que o valor de cedência corresponda ao valor máximo que o elemento suporta e a partir do qual passam a estar associadas deformações plásticas. Neste trabalho, são então adotados os modelos indicados na Figura 21. Por defeito, o programa funciona segundo o comportamento apresentado na Figura 21 a), contudo, por opção do utlizador, é possível introduzir o comportamento da Figura 21 b), sendo este último comportamento demostrado no subcapítulo 4.3 através de dois exemplos.

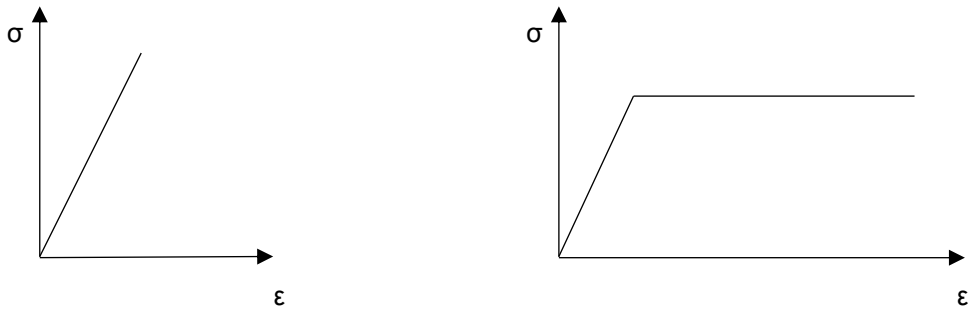


Figura 21: a) Regime linear elástico. b) Regime linear elástico-perfeitamente plástico.

Neste trabalho não é tida em conta a interação de esforços ao nível de elementos de ligação.

4.2 Análise estática

Neste subcapítulo, é apresentado e validado o primeiro e principal objetivo deste trabalho em termos de resultados e análise de estruturas - a análise estática de estruturas reticuladas (sujeitas a esforços axiais e de flexão), através do cálculo de esforços e deslocamentos em regime elástico.

Dos casos apresentados, apenas no primeiro será feita uma análise mais pormenorizada dos resultados, de modo a validar o programa. Os restantes casos serão apresentados apenas com o propósito de mostrar funcionalidades do programa e a abrangência do mesmo em termos de diversidade de estruturas possíveis de simular.

No primeiro exemplo apresentado, além de verificada a proximidade entre os resultados exatos e os obtidos pelo programa, é também verificada a forma como o método avalia as partículas e converge para a solução, através de trajetórias de deslocamento e esforços ao longo do tempo.

No final deste subcapítulo, são apresentados alguns exemplos de pórticos baseados em exemplos académicos utilizados no curso de Engenharia Civil.

Viga de dois tramos

Neste exemplo, avalia-se o comportamento de uma viga de dois tramos com uma consola de extremidade, sujeita a um carregamento constante no tempo. Geometricamente, a estrutura é composta por um perfil IPE100 (perfil adotado por defeito pelo programa para estrutura do tipo '*beam*') e tem as dimensões indicadas na Figura 22 a).

Tendo em conta a simplicidade do exemplo, o incremento de tempo utilizado pode ser um pouco superior aos valores usuais do programa, correspondendo a 7E-5 segundos. Será referido em pormenor, no subcapítulo 4.6, a influência do incremento de tempo nas simulações.

Para criar a estrutura no programa, recorreu-se à funcionalidade de discretização de uma barra, onde através do menu de propriedades da mesma se escolhe o número de segmentos pretendido através da opção '*number of segments*'. Os segmentos são criados todos com o mesmo comprimento, de acordo com o número de segmentos escolhido e comprimento total da barra a discretizar, e adotam as propriedades da barra inicial, evitando assim a necessidade de repetir em todos os segmentos, alterações previamente adotadas.

Ao longo da simulação é possível acompanhar graficamente a deformada da estrutura, bem como o andamento dos diagramas de esforços, caso estejam ativos (Figura 22 a Figura 24).

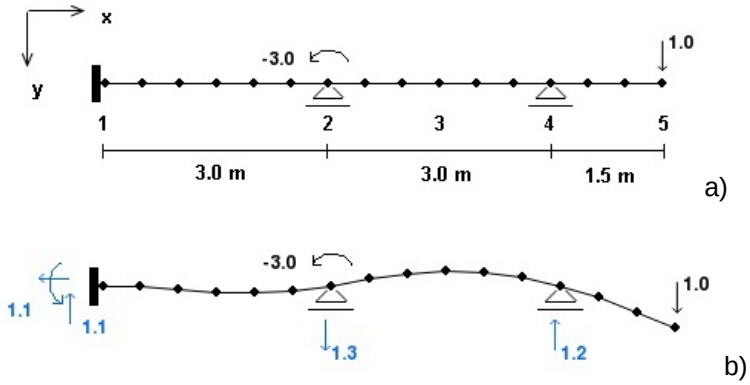


Figura 22: a) Configuração indeformada; b) Configuração deformada (fator de escala da deformada – 50).

Figura 23: Diagrama de esforço transverso – viga de dois tramos (fator de escala dos diagramas – 20).

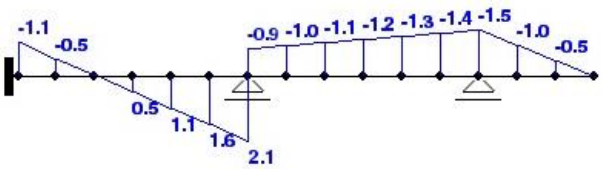


Figura 24: Diagrama de momentos (fator de escala dos diagramas – 20).

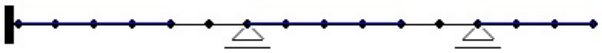


Figura 25: Diagrama de esforço axial – viga de dois tramos (fator de escala dos diagramas – 20).

Graficamente, os valores apresentados pelo programa são arredondados a uma casa decimal de modo a não sobrecarregar a imagem. No entanto, acedendo ao menu de propriedades de uma barra é possível consultar os valores dos esforços com várias casas decimais, bem como no menu de propriedades das partículas aceder aos valores de deslocamentos.

Nos Quadro 2 e Quadro 3, é possível fazer a comparação entre os valores obtidos recorrendo ao programa e valores exatos de esforços e deslocamentos elásticos.

Quadro 2: Comparação de esforços elásticos para uma solução estática – viga de dois tramos.

Esforços		**Partículas de extremidade**							
		1	**2**		**3**		**4**		**5**
		Dir.	**Esq.**	**Dir.**	**Esq.**	**Dir.**	**Esq.**	**Dir.**	**Esq.**
Momento Fletor [kN.m]	Extraído do programa	-1,081	2,133	-0,867	-1,189	-0,189	-1,499	-1,499	3,9E-7
	Exato	-1,100	2,100	-0,900	-1,200	-1,200	-1,500	-1,500	0,000
Esforço Transverso [kN]	Extraído do programa	1,081	1,081	-0,212	-0,212	-0,212	-0,212	0,999	0,999
	Exato	1,100	1,100	-0,200	-0,200	-0,200	-0,200	1,000	1,000
Esforço Axial [kN]	Extraído do programa	6,7E-4	3,3E-3	8,3E-4	2,2E-4	1,3E-4	9,3E-4	6,3E-3	8,4E-3
	Exato	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000

De acordo com o quadro anterior, verifica-se que os valores obtidos da simulação não são totalmente iguais aos exatos. Tal diferença deve-se ao facto de ser deixado ao critério do utilizador quando se dá por terminada a simulação e, dessa forma, os valores serão tão mais próximos dos valores exatos quando maior o tempo de simulação. No entanto, por mais longa que a simulação se torne, existirá sempre algum erro associado. Dessa forma, os resultados nunca serão iguais aos valores exatos.

Quadro 3: Comparação de deslocamentos elásticos para uma solução estática – viga de dois tramos.

Deslocamentos		**Partículas**				
		1	**2**	**3**	**4**	**5**
Extraído do programa	x [m]	0,00	-3,820E-6	-8,775E-6	-1,496E-5	-5,623E-5
	y [m]	0,00	0,00	-3,692E-3	0,00	1,119E-2
	θ [rad]	1,6E-16	4,475E-3	2,235E-4	-5,369E-3	-8,500E-3
Exato	x [m]	0,00	0,00	0,00	0,00	0,00
	y [m]	0,00	0,00	-3,744E-3	0,00	1,134E-2
	θ [rad]	0,00	4,521E-3	2,262E-4	-5,442E-3	-8,59E-3
Erro	x [%]	-	-	-	-	-
	y [%]	0,00	0,00	1,39	0,00	1,32
	θ [%]	-	1,02	1,19	1,34	1,05

Com base nos deslocamentos obtidos são verificadas as condições de apoio nas respetivas partículas, mais concretamente, o impedimento de deslocamentos vertical e horizontal e a rotação na partícula 1, devido à existência de um apoio encastrado, e o impedimento do deslocamento vertical nas partículas 2 e 4, devido à existência de um apoio móvel. Os deslocamentos na direção x não são exatamente zero, pois dependem da deformação axial dos segmentos e, por mais pequeno que sejam os movimentos das partículas, existirá uma deformação axial residual em função desses movimentos. Os deslocamentos apresentam uma boa proximidade de valores nunca tendo erros superiores a 1,4%. Tal como referido acerca das forças, os valores serão tão mais próximos dos exatos quanto mais longa for a simulação.

Sendo um programa com objetivo de utilização futura para suporte ao ensino, o seu desempenho é um critério importante, não só ao nível de resultados, mas também ao nível de tempos de simulação. Nesta simulação, o critério de paragem escolhido é do tipo qualitativo, isto é, a simulação para por ordem do utilizador quando este se apercebe, por simples análise visual do que vai sendo apresentado no monitor, que os valores dos diagramas já não sofrem alterações. Desta forma, e tendo em conta o arredondamento gráfico a uma casa decimal, os erros dos valores obtidos podem ser um pouco superiores ao aceitável.

A opção de não criar nenhum critério de paragem automático foi para dar ao utilizador total liberdade de explorar o método, pois desta forma pode ser o utilizador a estabeler o limite de simulação. No subcapítulo 4.6, será feita uma breve análise de tempos de simulação ao nível do desempenho do programa, bem como de parâmetros que influenciam este fator.

Como o método utilizado se baseia na análise de partículas, é interessante poder avaliar o comportamento de uma ou várias partículas ao longo da simulação. Embora seja percetível graficamente a deformação da estrutura, esta resulta do acumular de milhares de pequenos incrementos de movimento e deformação dos vários elementos da estrutura, não sendo por isso visível o comportamento das partículas isoladamente.

Por forma a dar a conhecer a influência do método no comportamento das partículas, bem como validar os resultados obtidos, foram extraídos do programa valores dos deslocamentos e esforços interiores de algumas partículas com características de referência do caso avaliado.

A extração de informação foi feita recorrendo à funcionalidade do programa que cria um ficheiro Excel com a informação da partícula. Selecionando a partícula pretendida e premindo a tecla ‘e’, é criado um ficheiro Excel (.xlsx) com colunas respetivas aos deslocamentos, esforços e energia cinética (associada ao amortecimento cinético), contendo os valores das várias iterações, bem como os gráficos com as trajetórias ao longo do tempo de cada uma das colunas.

Para esta exportação acontecer, é necessário indicar ao programa, através do menu de propriedades do ambiente (opção ‘*Export to Excel*’), que se pretende que os valores das iterações sejam guardados. Tal condição surge pelo elevado consumo de memória exigido ao computador, em simulações longas, para guardar os valores. Desta forma, o utilizador consegue melhorar o desempenho do programa caso não pretenda fazer exportação, bem como fazer exportação apenas de parte da simulação.

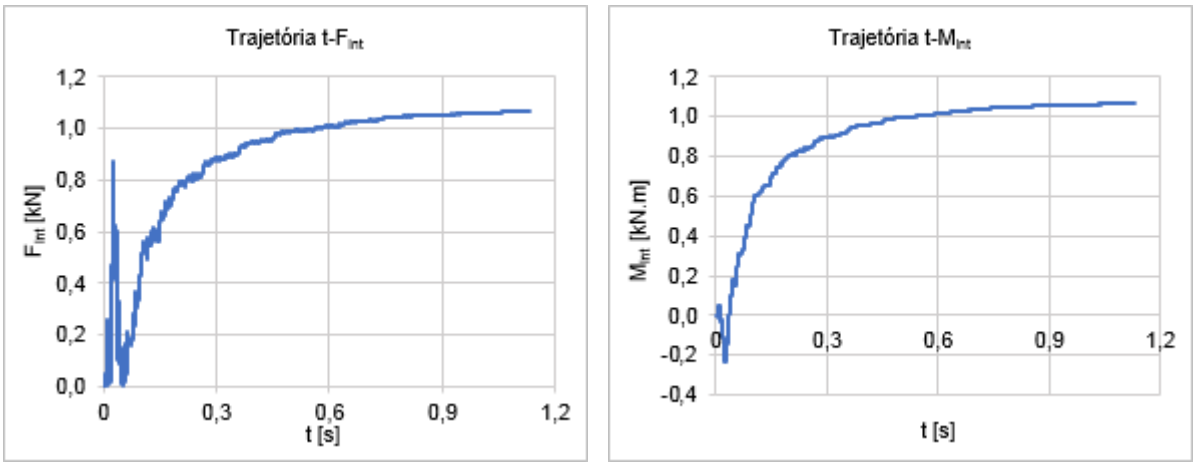
Embora esta funcionalidade resulte de uma função criada para este programa, a qual reúne e organiza a informação a exportar, bem como cria os respectivos gráficos, a criação do ficheiro .xlsx foi conseguida recorrendo a funções já existentes numa biblioteca de Python, mostrando-se assim, mais uma vez, a versatilidade desta linguagem.

Neste exemplo, foi exportada informação de cinco partículas que diferem nas suas condições de apoio e carregamento, as quais estão identificadas na Figura 22 a). A sua escolha, bem como a da estrutura, foi feita por forma a mostrar várias situações presentes em modelos de estruturas correntes.

Neste subcapítulo, como o objetivo é validar o equilíbrio e condições de apoio das partículas, apenas é necessário o andamento geral das trajetórias. No entanto, serão apresentadas no subcapítulo 4.5 trajetórias em maior detalhe, por forma a mostrar influência de ambos os tipos de amortecimento, viscoso e o cinético, na convergência da solução.

Partícula 1

Correspondendo a um encastramento, as condições de fronteira cinemáticas associadas a esta partícula é a de que são nulos todos os deslocamentos. As forças interiores sobre a partícula são iguais aos valores das reações. Relativamente aos deslocamentos, sendo esta uma partícula com restrições de movimento, o programa não impõe que esses deslocamentos sejam zero, mas sim cria reações que vão contribuir para o equilíbrio da partícula contrariando os respetivos deslocamentos. Na realidade, a partícula está em constante movimento devido a forças residuais resultantes do equilíbrio, no entanto, tais movimentos são desprezáveis e indetetáveis na simulação. Nesta partícula devido aos deslocamentos serem “nulos”, os gráficos seriam redundantes e por isso não serão representados. Na Figura 26, tal como esperado, as forças interiores convergem para o valor da reação respetiva.



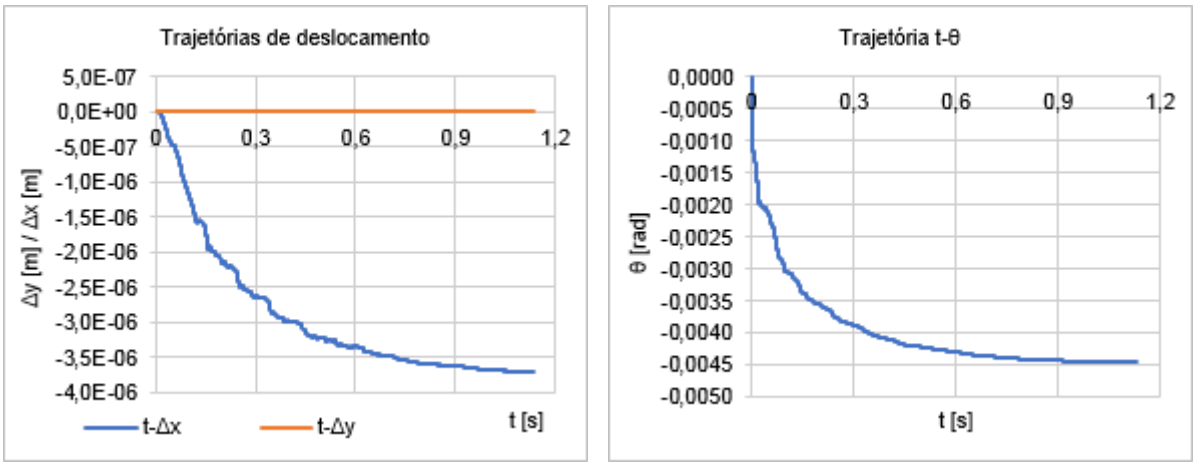
a) b)

Figura 26: Trajetórias de forças interiores ao longo do tempo – Partícula 1. a) Forças. b) Momentos.

Partícula 2

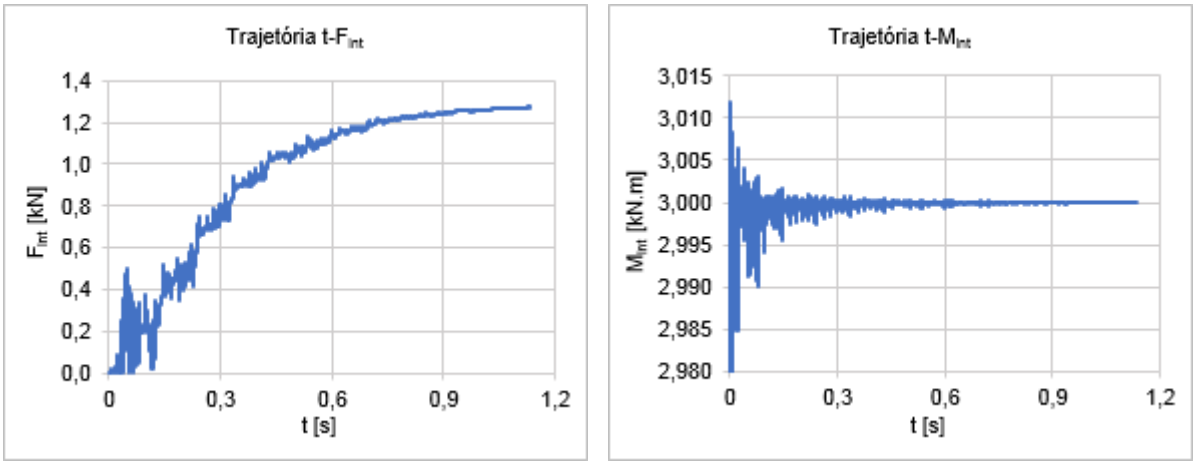
Esta partícula corresponde a um apoio de continuidade entre os dois tramos da estrutura, o qual é representado pela restrição de movimento na direção vertical (segundo o eixo y). Avaliando as trajetórias da Figura 27 e da Figura 28 a), são visíveis a restrição de movimento do apoio, a convergência dos deslocamentos para os valores elásticos e a convergência da força interior para o valor da reação do apoio, valores esses já indicados nos Quadro 2 e Quadro 3.

Em termos de forças exteriores, a partícula está sujeita a um momento aplicado. Por isso, embora haja liberdade de rotação na partícula, o momento interior não converge para zero, mas sim para o valor do momento exterior.



a) b)

Figura 27: Trajetórias de deslocamento ao longo do tempo – Partícula 2. a) Translação. b) Rotação.



a) b)

Figura 28: Trajetórias de forças interiores ao longo do tempo – Partícula 2. a) Forças. b) Momentos.

Partícula 3

Sendo uma partícula situada a meio vão, não existem quaisquer restrições de deslocamentos e, por isso, a sua convergência será para os deslocamentos elásticos da correspondente solução estática exata (Figura 29). Em termos de forças, os acréscimos de forças interiores convergem para zero, pois não existem cargas aplicadas nem reações de apoio (Figura 30).

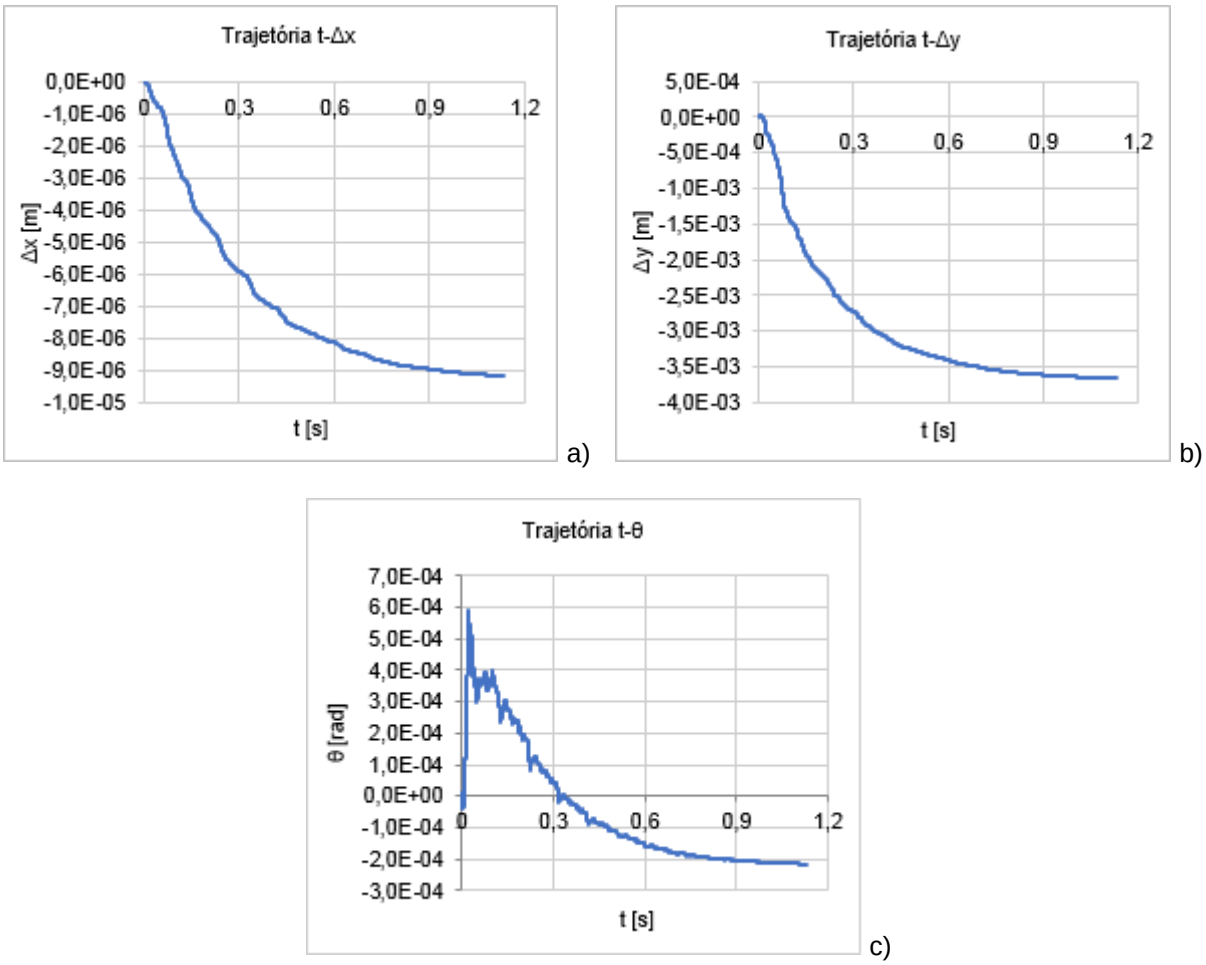


Figura 29: Trajetórias de deslocamento ao longo do tempo – Partícula 3. a) Translação segundo a direção x. b) Translação segundo a direção y. c) Rotação.

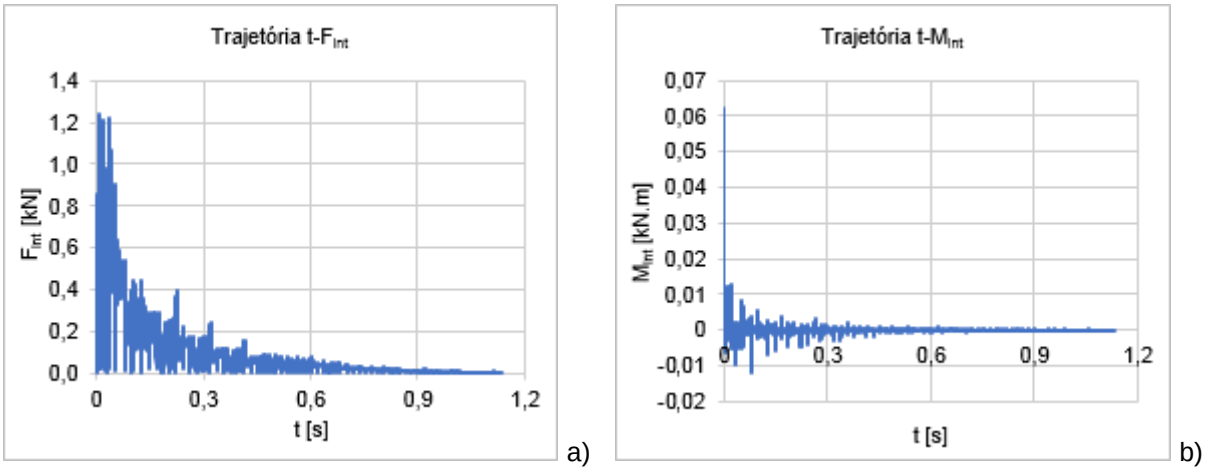
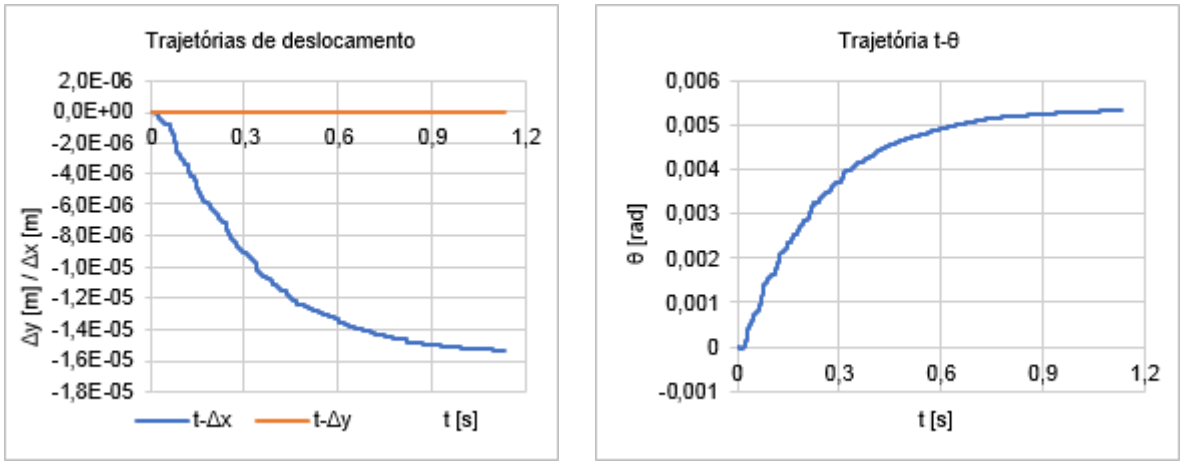


Figura 30: Trajetórias de forças interiores ao longo do tempo – Partícula 3. a) Forças. b) Momentos.

Partícula 4

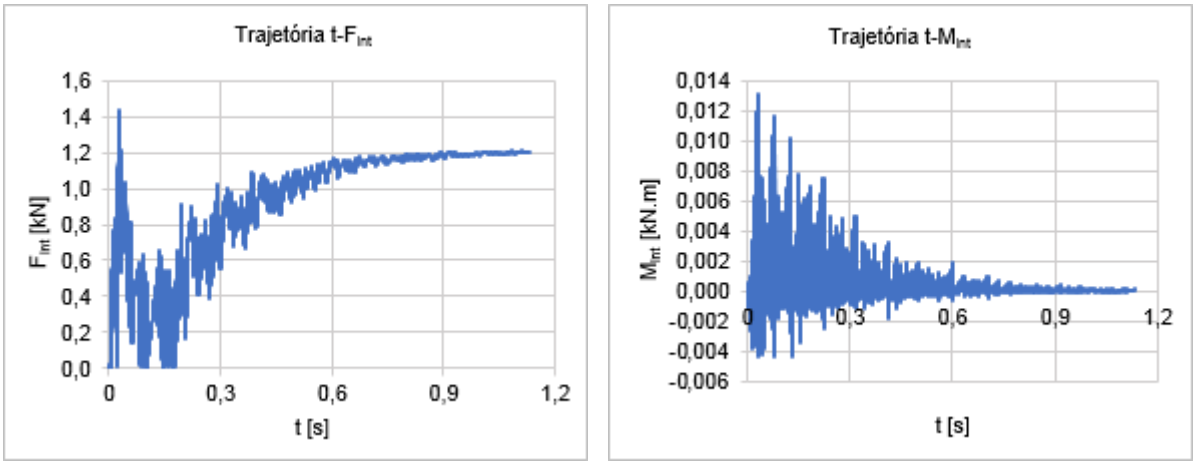
Tal como a partícula dois, a partícula quatro representa um apoio de continuidade com restrição de movimento apenas na direção segundo y, sendo que os restantes deslocamentos convergem para os valores exatos (Figura 31).

Relativamente aos esforços, os valores de força e momento interiores convergem de forma a que o equilíbrio na partícula seja obtido. Através da Figura 32, é possível verificar que a força interior converge para o valor da reação vertical e o momento interior converge para zero, pois não existe reação de rotação ou momento exterior aplicado.



a) b)

Figura 31: Trajetórias de deslocamento ao longo do tempo– Partícula 4. a) Translação segundo as direções x e y. b) Rotação.



a) b)

Figura 32: Trajetórias de forças interiores ao longo do tempo – Partícula 4. a) Forças. b) Momentos.

Partícula 5

Nesta ultima partícula, sendo a de extremidade livre da consola, não existem restrições de deslocamentos (Figura 33) e, em termos de forças interiores, está sujeita a uma força aplicada, para a qual convergirá a força interior (Figura 34).

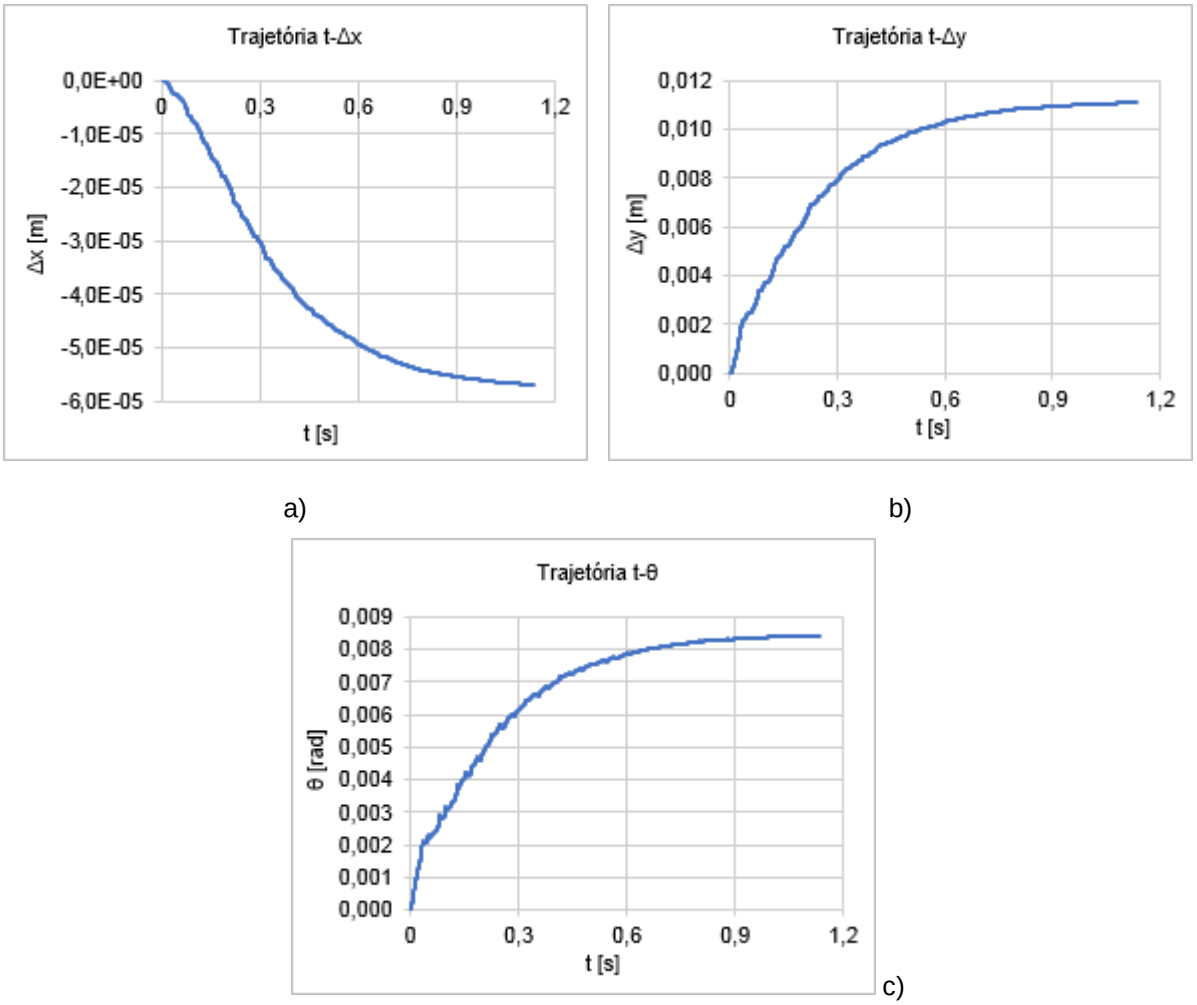


Figura 33: Trajetórias de deslocamento ao longo do tempo – Partícula 3. a) Translação segundo a direção x. b) Translação segundo a direção y. c) Rotação

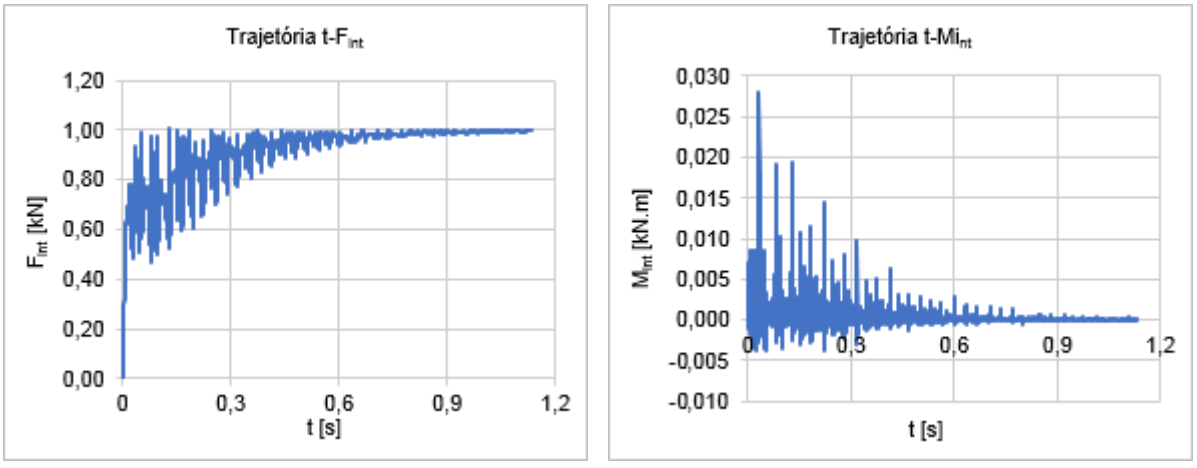


Figura 34: Trajetórias de forças interiores ao longo do tempo – Partícula 5. a) Força. b) Momento.

Pórticos

Nesta seção, serão indicadas algumas funcionalidades do programa para a análise de estruturas porticadas, tendo por base modelos de estruturas correntes (representativos de edifícios, reservatórios, pontes, entre outros).

Os exemplos seguintes são apresentados por forma a mostrar a capacidade do programa em obter uma solução estática de pórticos simples constituídos por perfis IPE100 sujeitos a cargas horizontais.

Nestes exemplos, foi utilizada a funcionalidade correspondente à alteração da escala do programa, a qual se encontra no menu de propriedades de ambiente através da opção '*Program scale*'. Esta permite alterar fisicamente a dimensão da estrutura sem que a isso corresponda alteração gráfica. Criando a mesma estrutura segundo dois valores de escala diferentes, o utilizador vê a mesma dimensão gráfica, mas a posição das partículas, bem como o comprimento das barras estão multiplicados pelo valor escolhido. Utilizando esta funcionalidade, o utilizador deixa de estar restringido pela dimensão da janela do programa, podendo criar estruturas com quaisquer dimensões.

Num primeiro exemplo apresenta-se a simulação de um pórtico sujeito a uma carga horizontal (Figura 35 e Figura 36).

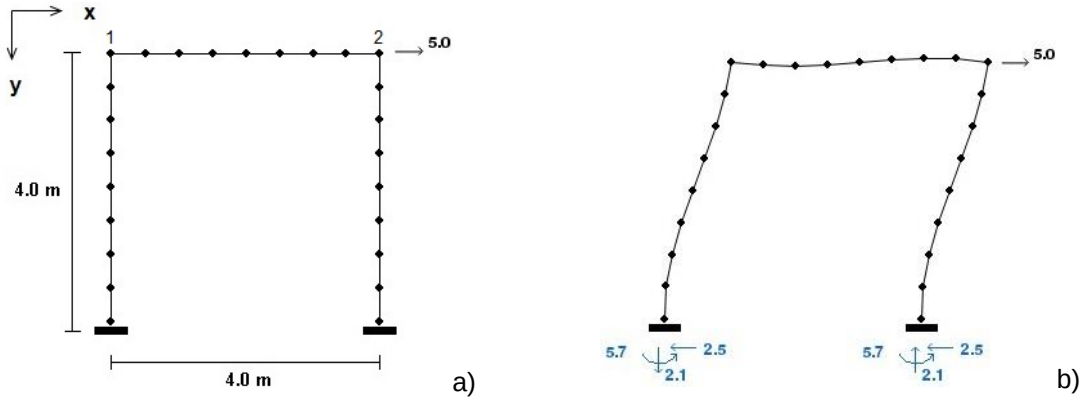


Figura 35: a) Configuração indeformada; b) Configuração deformada (fator de escala da deformada -15).

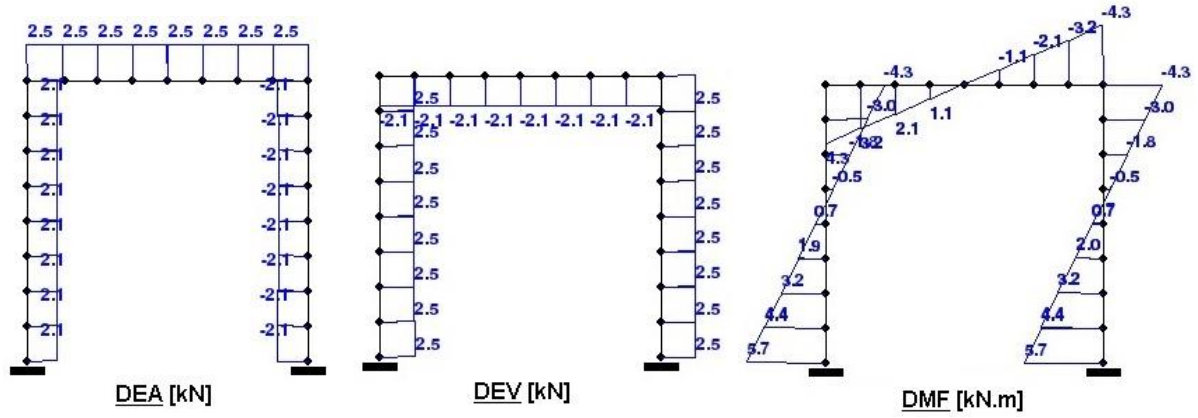
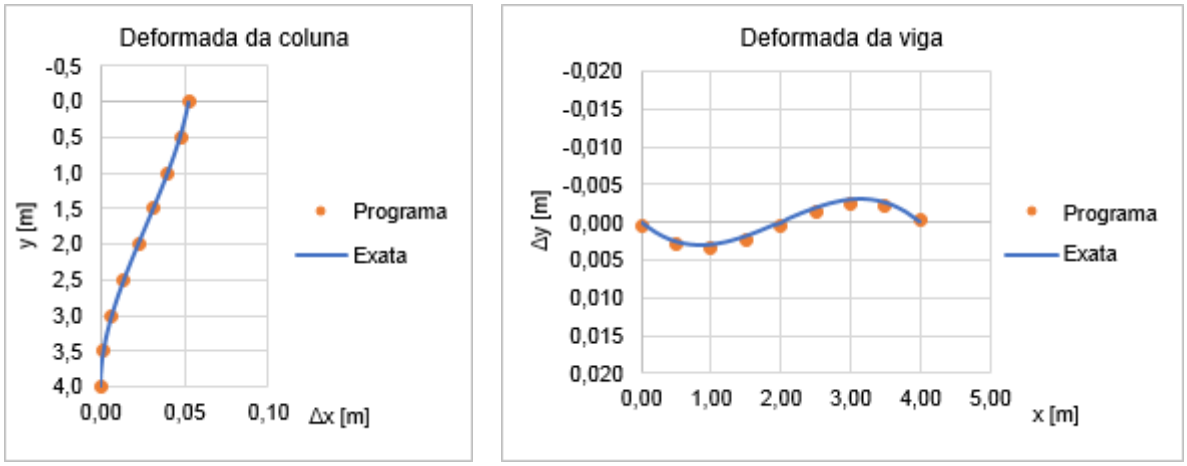


Figura 36: Diagramas de esforços (fator de escala dos diagramas- 10).

Neste exemplo, é também apresentada a comparação entre as deformadas exatas e as obtidas utilizando o programa, dos elementos coluna e viga (Figura 37), por forma a validar a deformada apresentada na Figura 35 b).



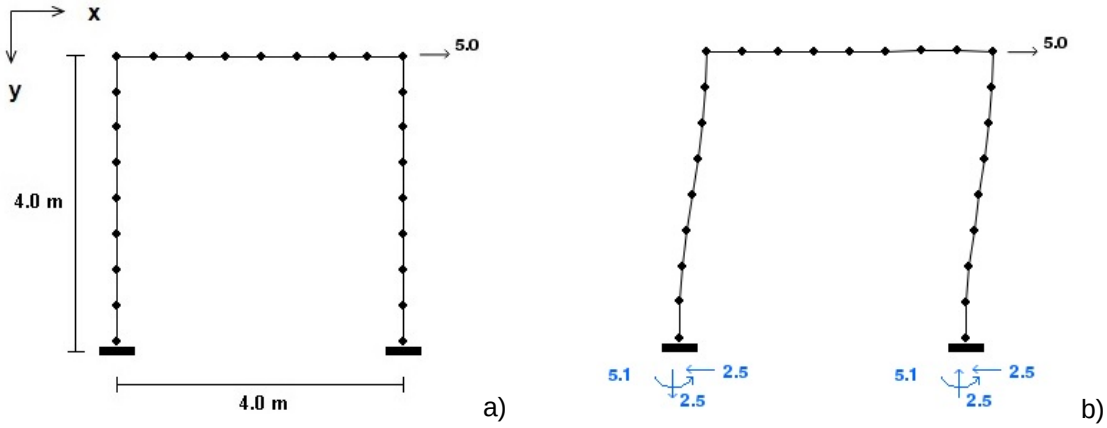
a) b)

Figura 37: Comparação de deformadas exatas e obtidas com o programa. a) Coluna. b) Viga.

Uma simplificação que é habitual fazer-se na análise de pórticos (em particular na análise face a forças horizontais como as que resultam da ação sísmica) é considerar que os pisos (as vigas que o constituem) são rígidos e é essencialmente a flexão dos elementos verticais que resiste às forças.

Neste programa, é possível simular o comportamento de uma barra indeformável à flexão, bastando para isso aumentar a sua rigidez de flexão. No entanto, tal aumento não é indiferente. Para um aumento exagerado de rigidez de flexão, os esforços serão muito elevados para pequenas deformações e o programa irá divergir. É possível evitar a divergência diminuindo o incremento de tempo da simulação, contudo, a simulação torna-se demorada.

De seguida, é apresentado, nas Figura 38 a Figura 40, a estrutura apresentada anteriormente com a diferença de ter a viga como elemento “indeformável” à flexão. Foi aplicado um aumento de rigidez de flexão de 10 vezes, pois é um valor que, para esta estrutura, já oferece uma boa representação do comportamento de “viga rígida” à flexão sem implicar um valor de incremento no tempo que diminua consideravelmente o tempo de simulação.



a) b)

Figura 38: a) Configuração indeformada; b) Configuração deformada (fator de escala da deformada – 6).

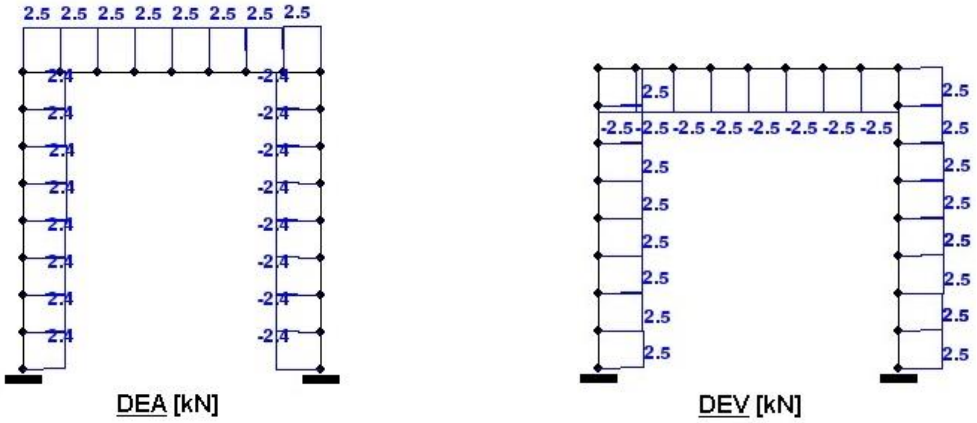


Figura 39: Diagramas de esforços (fator de escala dos diagramas - 10).

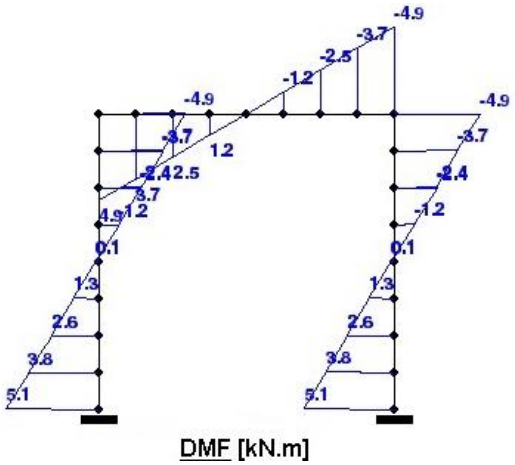


Figura 40: Diagramas de esforços (fator de escala dos diagramas - 10).

Outra característica bastante presente em estruturas porticadas é a existência de rótulas, de libertações de momento fletor, o que elimina a continuidade de momentos fletores. A introdução de rótulas em ambas as extremidades de uma barra faz com que esta (caso não tenha qualquer carregamento transversal no seu vão) fique apenas sujeita a esforços axiais. É o modelo de barra que se utiliza quando se pretende modelar escoras e/ou tirantes.

Neste programa, para rotular todas as ligações do nó, basta aceder ao menu de propriedades da respetiva partícula e através da opção '*create hinge'* rotular o nó. Para rotular apenas a ligação associada a uma determinada barra, acede-se ao menu de propriedades da barra e através das opções '*left end hinge*' ou '*right end hinge'* rotula-se a extremidade pretendida.

Na Figura 41 e Figura 42, será apresentado o mesmo exemplo da Figura 38 , mas desta vez com as partículas de ligação entre a viga e as colunas rotuladas. Pelo carregamento aplicado, a viga estará apenas sujeita a esforço axial e, por isso, não tendo deformação transversal, não foi necessário discretizar tal elemento.

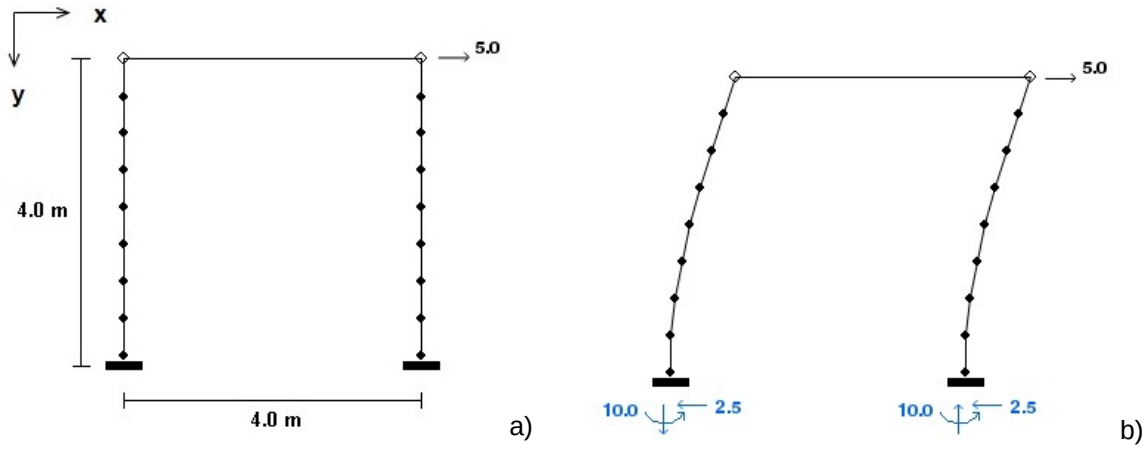


Figura 41: a) Posição indeformada; b) Posição deformada (fator de escala gráfico – 6).

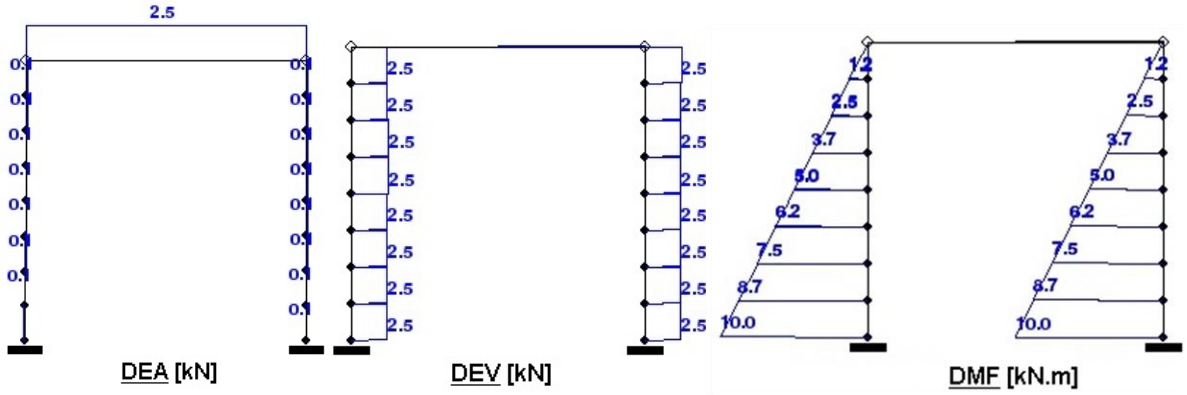


Figura 42: Diagramas de esforços (fator de escala dos diagramas - 10).

Para finalizar esta seção, apresenta-se um exemplo onde são rotuladas as ligações de extremidades de duas das vigas da estrutura e mantida a continuidade nas restantes ligações da partícula. Esta característica é utilizada, por exemplo, em modelos de cálculo de esforços de sistemas mistos com pilares e paredes resistentes. Quando sujeitos a ações horizontais, e para analisar a interação entre pilares/vigas e paredes estruturais, as vigas responsáveis pela ligação de ambos os tipos de elementos são rotuladas nas extremidades e é mantida a continuidade nas restantes ligações do nó (Appleton 2013).

No exemplo seguinte, apresenta-se um modelo representativo da situação referida (Figura 43 a)). Nas vigas biarticuladas foi atribuída uma rigidez axial superior à das restantes barras, para que haja uma menor deformação axial e ambas as partes da estrutura, entre as referidas vigas, deformem em "paralelo".

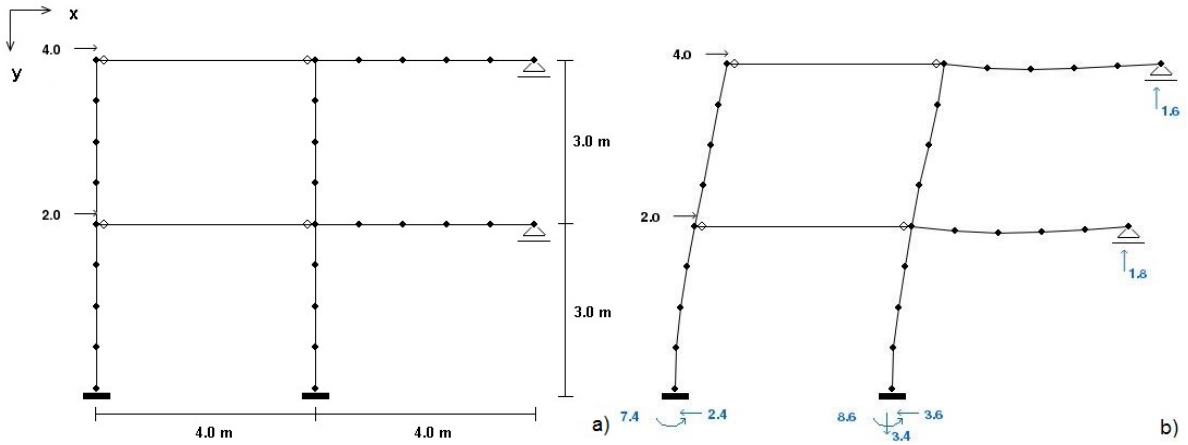


Figura 43: a) Configuração indeformada. b) Configuração deformada (fator de escala da deformada – 6).

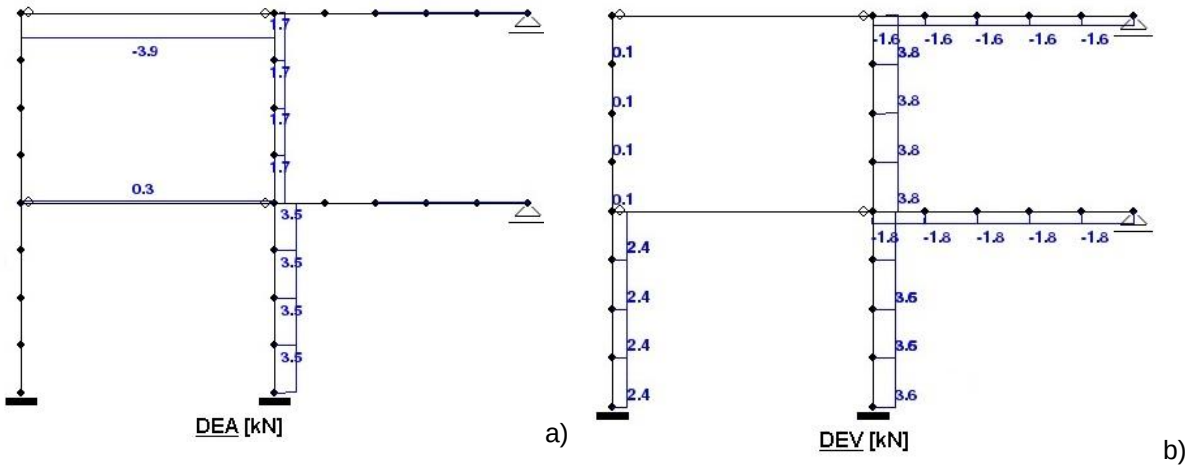


Figura 44: Diagramas de esforços (fator de escala dos diagramas - 5). a) Esforço axial. b) Esforço transverso.

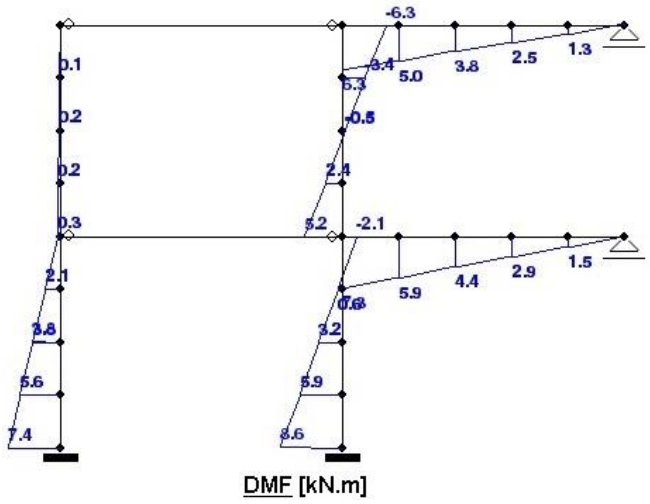


Figura 45: Diagrama de momentos fletores (fator de escala do diagrama- 5)

4.3 Análise elasto-plástica incremental

Uma vez que os métodos das partículas finitas têm vindo a ser aplicados na análise de comportamentos geométrica e fisicamente não lineares, fez sentido também neste trabalho abordar essa temática ainda que de forma bastante simplificada.

Sendo este um programa onde o comportamento dos materiais é maioritariamente em regime elástico, o próximo passo passaria pela aplicação do método a comportamentos não lineares, como por exemplo avaliar a capacidade plástica de estruturas. No entanto, o estudo da plasticidade, devido à sua extensão, requer um trabalho bastante complexo e moroso que por si só daria para fazer uma dissertação inteira sobre tal assunto. Não fazendo parte desta dissertação tal detalhe e abrangência, optou-se por restringir esta seção a uma análise elasto plástica incremental.

Embora a plastificação de um barra ocorra seção a seção, tendo em conta o modelo elástico perfeitamente plástico deste trabalho (subcapítulo 4.1), adotou-se o conceito de rótula plástica, onde é admitido que um elemento plastifica quando a sua seção mais esforçada plastificar.

Para poder utilizar tal hipótese, é necessário adotar uma relação momentos-curvatura elástica perfeitamente plástica por forma a que o comportamento do elemento tenha apenas duas fases, uma elástica enquanto o momento da seção mais esforçada seja inferior ao momento plástico e uma perfeitamente plástica quando for igual. É ainda admitida a hipótese de que as seções apenas plastificam por flexão, não sendo tida em conta a possível plastificação por via do esforço axial.

Embora tal simplificação não replique com exatidão a realidade, é uma aproximação aceitável (bem como conservativa) do comportamento da estrutura, principalmente para o tipo de perfil escolhido para as simulações deste trabalho (perfil em 'I'), por ter um fator de forma próximo de um e, por isso, uma menor diferença entre este modelo e a realidade, em termos de "patamar" de plasticidade.

Neste programa, a escolha do comportamento do material, entre ser sempre em regime elástico ou em regime elástico perfeitamente plástico, é feita pelo utilizador através do menu de propriedades das barras. Neste, podem-se atribuir valores de esforço axial plástico, nas opções '*tensile strength*' e '*compressive strength*', e de momento plásticos, na opção '*bending strength*'. Caso sejam introduzidos valores nestas opções, irá ser criado um patamar de plasticidade na barra, fazendo com que, ao atingir essa resistência do elemento, este continue a ter deformações indefinidamente sem aumento de carga. Contudo, como já referido anteriormente, esta versão do programa não contabiliza ambos os efeitos em conjunto.

Para exemplificar tal comportamento, serão então apresentados dois casos nos quais foi aplicado um carregamento incremental (incremento esse que no programa é feito pelo utilizador). A informação da simulação é reunida na forma de gráficos, que mostram a trajetória de carregamento da estrutura. Para além destes, apresentam-se também imagens da deformada e diagramas de esforços.

No primeiro caso, será feita uma análise em pormenor dos resultados obtidos, bem como a sua comparação com os valores exatos, enquanto que no segundo caso será apenas feita uma apresentação gráfica dos resultados.

Viga encastrada-apoiada

Neste exemplo, será apresentada a analise incremental de uma viga encastrada-apoiada de um tramo, sujeita a uma carga pontual a meio vão. As barras são constituídas por perfis IPE100.

Ao longo da simulação foi sendo incrementada a carga aplicada por forma a identificar a carga última da estrutura, isto é, aquela à qual corresponde a criação de um mecanismo de colapso. Na Figura 46, são apresentadas as deformadas da estrutura para valores de carga associados às rótulas plásticas. Para melhor entender a posição das rótulas plásticas, são também apresentados, na Figura 47, Figura 48 e Figura 49, os respetivos diagramas de momentos fletores.

Para os valores exatos, as cargas associadas a cada rótula plástica foram calculadas com base nos esforços elásticos, igualando a expressão do maior momento fletor ao momento plástico (equação (48) e (49). Os deslocamentos, também calculados pelas expressões de flechas elásticas (equação (50)), dependem de um fator k que varia consoante as condições de apoio. Esta estrutura requer então os valores de k de uma viga encastrada-apoiada, enquanto não se forme a primeira rótula plástica, e depois o de uma viga simplesmente apoiada até se formar o mecanismo de colapso.

As alterações de condições de apoio devem-se ao facto de a primeira rótula se formar no encastramento e, por isso, ao perder resistência à rotação nesse apoio, passa a ter um comportamento semelhante ao de uma viga simplesmente apoiada.

$$M_{encastramento} = -\frac{3PL}{16} \tag{48}$$

$$M_{pl} = W_{pl} \times f_y = 10{,}84\ kN.m \tag{49}$$

onde W_{pl} é o módulo de resistência à flexão do perfil utilizado (consultado em tabelas do respetivo perfil IPE100) e f_y é a tensão do aço, em que, neste caso foi adotado o valor de 275 MPa.

$$\Delta y = k\frac{PL^3}{EI}, \qquad declive = \frac{P}{\Delta y} = \frac{EI}{kL^3} \tag{50}$$

$$k_{encastrado-apoiado} = \frac{7}{768}, \qquad k_{apoiado-apoiado} = \frac{1}{48}$$

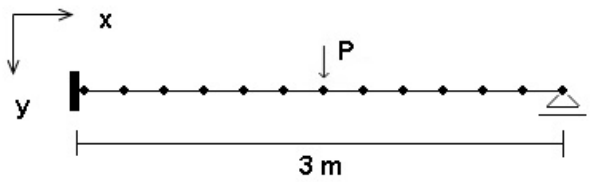


Figura 46: a) Configuração indeformada.

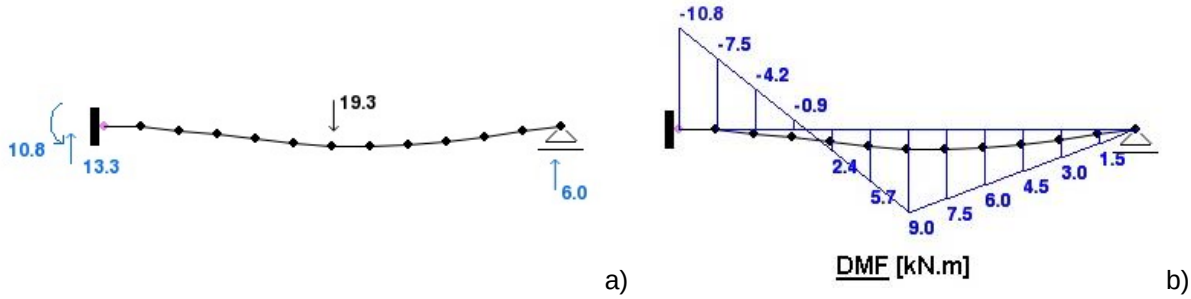


Figura 47: Primeira rótula plástica. a) Configuração deformada (fator de escala da deformada – 10). b) Diagrama de momentos (fator de escala do diagrama – 6).

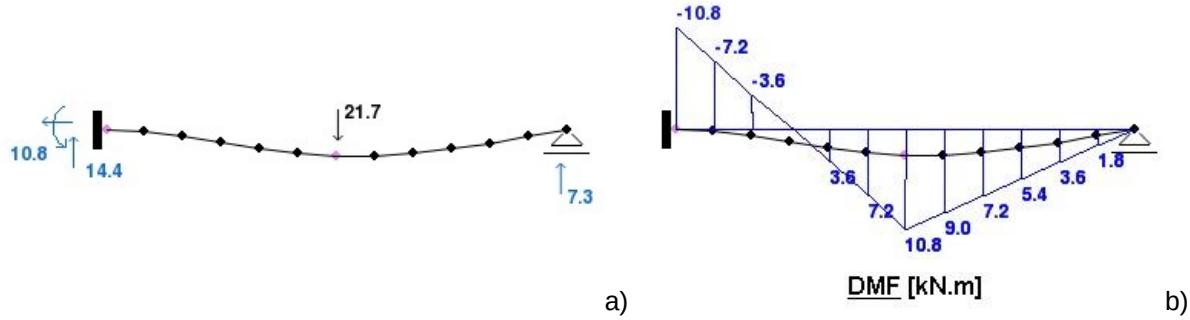


Figura 48: Segunda rótula plástica. a) Configuração deformada (fator de escala da deformada – 10). b) Diagrama de momentos (fator de escala do diagrama – 6).

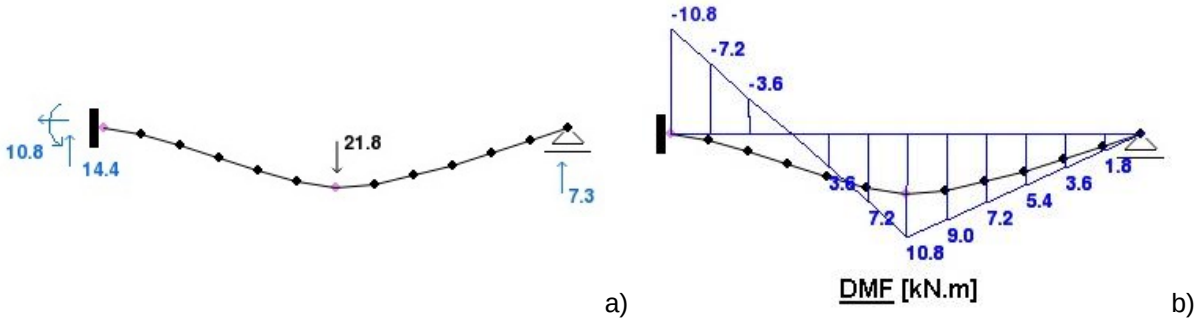


Figura 49: Mecanismo de colapso. a) Configuração deformada (fator de escala da deformada – 10). b) Diagrama de momentos (fator de escala do diagrama – 6).

A cada incremento de carga foram retirados os respetivos valores do deslocamento a meio vão e com os quais se criou a trajetória de carga apresentada na Figura 50.

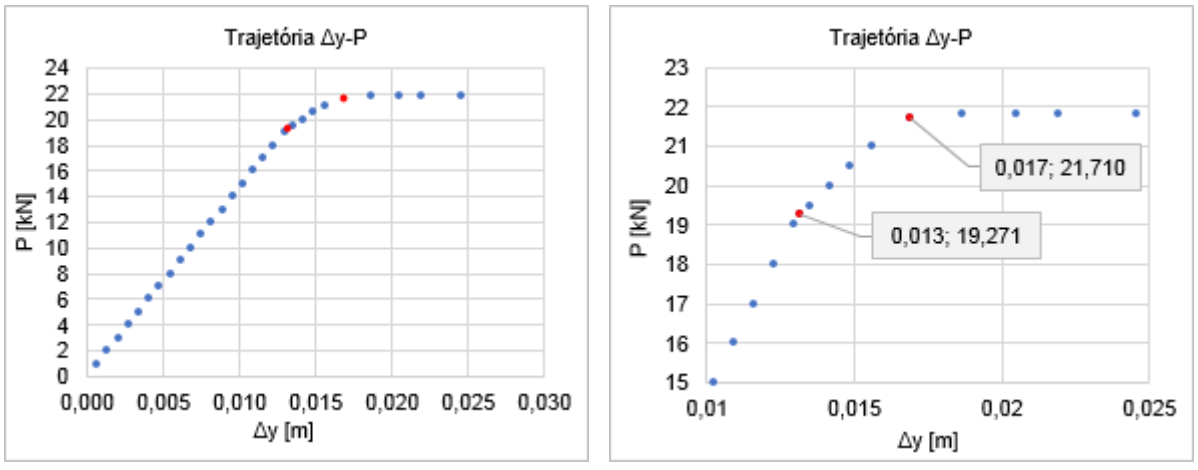


Figura 50: a) Trajetória Δy-P – viga encastrada apoiada. b) Detalhe de a).

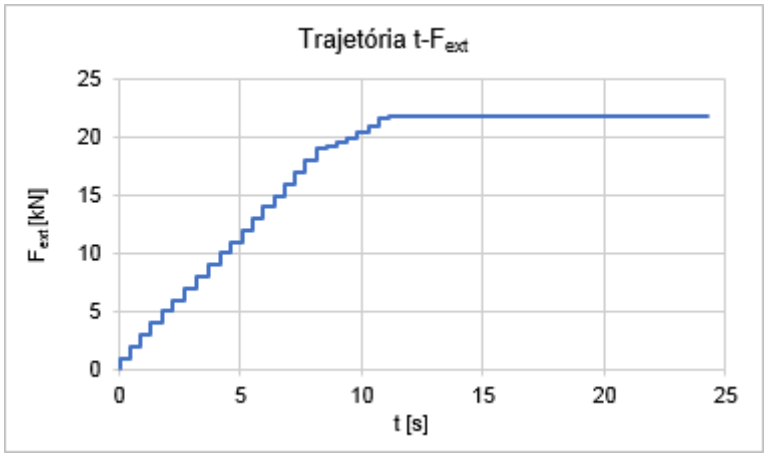


Figura 51: História de carregamento ao longo do tempo.

Com base na Figura 50, é facilmente visível o efeito das rótulas plásticas, pois o declive da trajetória diminui devido à perda de rigidez da estrutura. Neste caso, sendo uma estrutura hiperstática de primeiro grau, são necessárias duas rótulas plásticas para a estrutura se tornar um mecanismo, simulando o colapso da mesma. A posição das rótulas e a ordem do seu aparecimento dependem das seções nas quais o valor de momento fletor atingisse o momento plástico.

Para validar os resultados obtidos, será feita a comparação (Quadro 4) do valor de carga associado a cada uma das rótulas plásticas e do valor de declive de trajetória com os respetivos valores exatos para os três instantes de simulação apresentados nas Figura 47 a Figura 49.

Quadro 4: Comparação de valores do programa e exatos da análise elasto-plástica.

Instante	**Carga [kN]**		**Δy [m]**		**Declive da trajetória [kN/m]**	
	Exato	**Extraído do programa**	**Exato**	**Extraído do programa**	**Exato**	**Extraído do programa**
1ª rótula	19,267	19,271	0,0132	0,0132	1459,200	1459,920
2ª rótula	21,676	21,710	0,0170	0,0169	638,400	659,190
Mecanismo	21,676	21,800	0,0346	0,0346	0,000	0,000

Relativamente aos valores de carga obtidos, existe uma proximidade bastante aceitável entre os valores exatos e experimentais, verificando-se erros não superiores a 0,57% na carga crítica, 0,58% no deslocamento e 3,15% no declive da trajetória. Para o aparecimento da segunda rótula plástica, é visível que foi necessário um valor ligeiramente maior ao valor exato, no entanto, pode ser justificado pelo processo de convergência.

Tal como já referido no subcapítulo anterior, a convergência gráfica ocorre para valores ligeiramente menores que os exatos, no entanto, como já são bastante próximos, os incrementos de valores das iterações seguintes serão muito pequenos e com o avançar do tempo serão ainda menores, tornando a simulação demorada para conseguir um aumento de algumas centésimas nos resultados. Dessa forma, o utilizador pode pensar, com base na estagnação gráfica de valores, que a solução já convergiu e que o carregamento actuante não cria rótula plástica quando tal poderia acontecer eventualmente com o avançar da convergência. O utilizador pode assim acabar por incrementar o carregamento e a

rótula surgir para um valor ligeiramente superior ao exato, contudo, tal não sacrifica a qualidade dos resultados.

Em termos de rigidez, é notória uma diferença de valores de declive depois do aparecimento da primeira rótula plástica. Contudo, sendo o valor de declive resultante do programa calculado com base nos valores de deslocamento e carga obtidos do programa, uma pequena diferença ao nível das casas decimais pode gerar uma variação de algumas dezenas no valor de declive. Tendo os valores de deslocamento e carga sido arredondados à quarta casa decimal, pode assim ser justificada a diferença apresentada.

Já em mecanismo de colapso, o valor de carga experimental é superior aquele que originou a segunda rótula plástica, pois, tal como já referido e devido à estagnação da solução, os incrementos são muito pequenos, o que eleva bastante o tempo de simulação. Dessa forma, para obter deslocamentos plásticos elevados (identificativos do mecanismo) num menor tempo de simulação, introduz-se um ligeiro incremento de carga uma vez que não afeta os resultados.

O valor de deslocamento no mecanismo de colapso refere-se ao instante em que a simulação foi terminada e foi escolhido por forma a que seja visível, pelas deformadas, o incremento de deformação sem aumento de carga.

Pórtico

Caso se queira controlar a zona do aparecimento da rótula plástica, é possível consegui-lo através das características dos elementos. No caso de um pórtico, por exemplo, pode ser necessário forçar o aparecimento das rótulas plásticas nas vigas e não nos pilares, pois, para ações horizontais, os últimos são mais importantes para a estabilidade da estrutura.

Para tal, atribuem-se propriedades às vigas de modo a que tenham um momento plástico menor que o dos pilares e, com isso, assim que a viga atinga o momento plástico, a ligação só suportará mais esforço através dos restantes elementos.

Nos casos seguintes serão apresentados, em paralelo, os resultados finais da análise incremental de uma estrutura constituída por barras com diferentes capacidades plásticas, sendo indicadas as propriedades no Quadro 5.

Embora tenham sido utilizados perfis tabelados, os momentos plásticos foram ajustados pela tensão de cedência de forma a mostrar o comportamento desejado na estrutura. A atribuição do tipo de perfil aos elementos barra foi feita de acordo com o apresentado na Figura 52.

Quadro 5: Propriedades geométricas e mecânicas das barras.

Elemento barra	EA [kN]	EI [kN.m^2]	i [m]	λ [kg/m]	W_{pl} [m^3]	f_y* [MPa]	M_{pl} [kN.m]
IPE100	216720	359,10	0,0407	8,10	39,41E-6	215	8,47
IPE140	345030	1136,52	0,0574	12,90	88,34E-6	335	29,59

*valores para perfis laminados a quente com espessura nominal superior a 40mm.

Nesta seção, o objetivo não é comparar a resistência de ambas as estruturas, mas sim mostrar a diferença de comportamento face ao elemento que plastifica primeiro. Foram assim adotadas características diferentes para as vigas e colunas para ser visível a diferença de tais comportamentos, entre a formação de rótulas plásticas ocorrer na viga ou nas colunas de uma estrutura porticada e sujeita a ações horizontais. Desta forma, é possível ao utilizador perceber graficamente a importância de um dimensionamento onde se formem rótulas plásticas primeiro nas vigas, pois com isso consegue-se manter a integridade global da estrutura.

Desta análise apresentam-se resultados referentes ao valor de carga que origina o aparecimento das primeiras rótulas plásticas, bem como referentes a um incremento de carga depois da plastificação dos elementos. São também apresentadas as respetivas deformadas (Figura 53 a Figura 56).

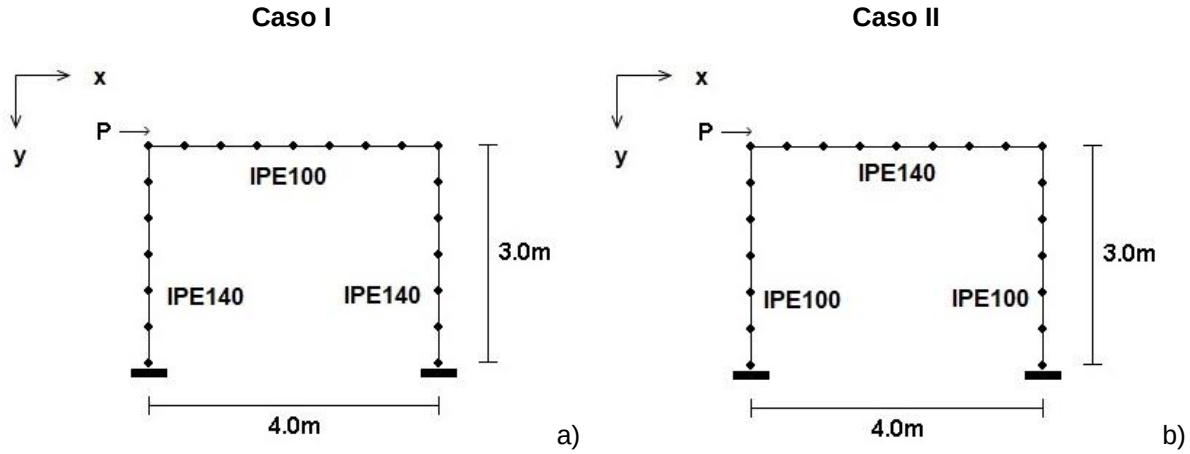


Figura 52: Configuração indeformada. a) Caso I. b) Caso II.

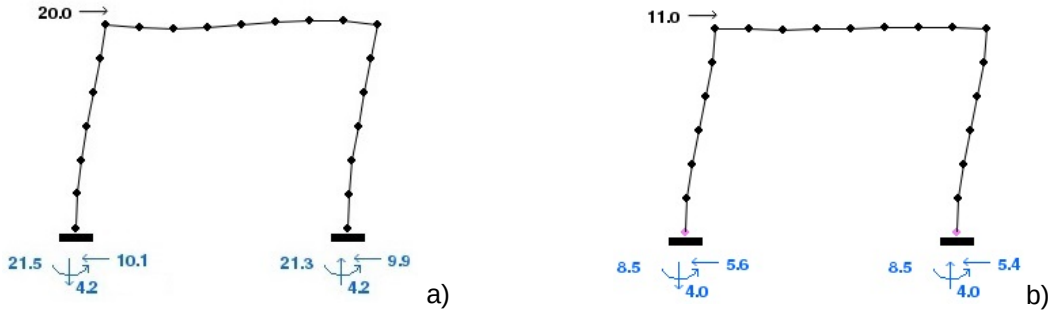


Figura 53: Configuração deformada aquando do aparecimento das rótulas plásticas (fator de escala da deformada – 10). a) Plastificação das extremidades da viga – Caso I. b) Plastificação da base das colunas – Caso II.

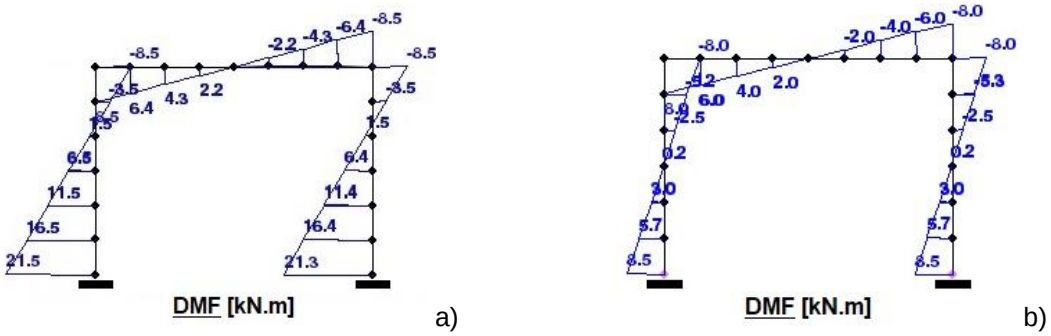


Figura 54: Diagrama de momentos fletores aquando do aparecimento das rótulas plásticas (fator de escala dos diagramas – 3). a) Plastificação das extremidades da viga – Caso I. b) Plastificação da base das colunas – Caso II.

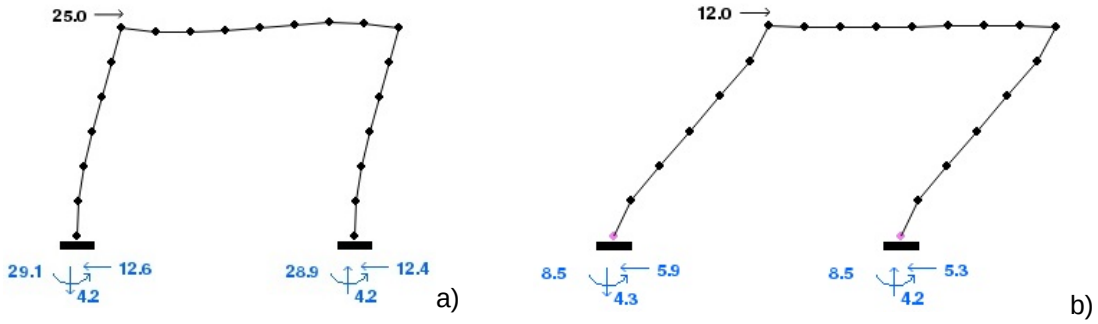


Figura 55: Configuração deformada para incremento de carga após o aparecimento das rótulas plásticas (fator de escala da deformada – 10). a) Caso I. b) Caso II.

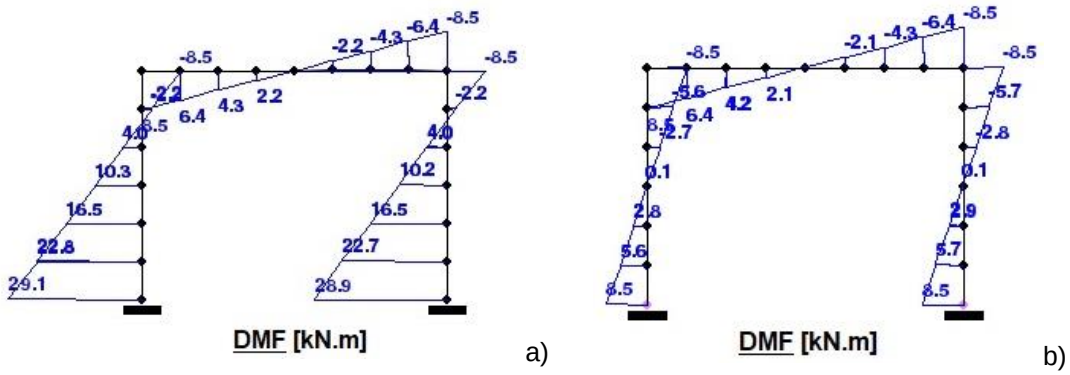


Figura 56: Diagrama de momentos fletores para incremento de carga após o aparecimento das rótulas plásticas (fator de escala dos diagramas – 3). a) Caso I. b) Caso II.

Graficamente são visíveis algumas diferenças entre ambos os casos. Uma delas é o facto de, no caso II, as rótulas plásticas estarem representadas a cor diferente, ao contrário do caso I. Tal acontece porque o programa altera a cor da partícula apenas quando todas as ligações do nó plastificam e este deixa de conseguir resistir a rotações. No caso II, a rótula plástica é formada no nó e todas as ligações desse nó plastificam (tal como aconteceu no exemplo anterior deste subcapítulo), enquanto que, no caso I, apenas uma das ligações do nó (ligação da viga) plastifica (a ligação da coluna embora não vá suportar mais esforço ainda não atingiu a sua capacidade resistente) e, por isso, o nó continua a manter as suas capacidades de resistência à rotação.

Em situações semelhantes ao caso I, onde apenas parte das ligações do nó plastificam e não há alteração de cor, o utilizador consegue obter informação da plastificação pelo diagrama de momentos fletores, uma vez que este deixa de aumentar ao atingir o valor de momento plástico introduzido pelo utilizador.

Outra diferença, a mais relevante deste exemplo, é relativa ao comportamento da estrutura. Em ambos os casos, a estrutura é hiperestátisa de terceiro grau e, por isso, entrará em colapso apenas quando se formarem quatro rótulas plásticas, no entanto, o comportamento até se formar um mecanismo de colapso é distinto.

No caso I, é visível, pelas Figura 53 - 56, a plastificação das ligações da viga através da estagnação do diagrama de momentos fletores. Mas, depois de um incremento de carga, a estrutura continuou a manter a sua integridade estrutural, pois as colunas continuaram a conseguir suportar o aumento de

carregamento por terem um momento plástico superior ao da viga. O mecanismo de colapso surgirá apenas quando as colunas esgotarem a sua capacidade resistente.

No caso II, ao serem as colunas as primeiras a plastificar, mais concretamente a base das mesmas, a estrutura perde grande parte da sua rigidez bem como capacidade resistente. Como o topo das colunas ainda não plastificou, a estrutura ainda conseguirá absorver algum esforço. No entanto, é possível verificar que, para um ligeiro incremento de carga, a estrutura esgota rapidamente a sua capacidade resistente e entra em mecanismo total, como é visível pela deformada da Figura 55 b).

Comparando o comportamento de ambos os casos, é visível que o caso II, no qual foram as colunas os primeiros elementos a plastificar, a estrutura mostrou uma menor resistência a ações horizontais, tornando-se numa solução pouco desejável em dimensionamento.

Com esta pequena simulação, embora de forma bastante simples e com condições escolhidas para evidenciar os comportamentos desejados, é novamente mostrado o potencial deste programa para auxiliar alunos de Engenharia Civil no entendimento de conceitos importantes de dimensionamento e análise de estruturas.

Neste caso, o comportamento de pórticos sujeitos a ações horizontais é importante por estar associado ao dimensionamento por capacidade real, um conceito bastante importante no dimensionamento de estruturas.

Em estruturas mais complexas, ou com maior numero de rótulas plásticas, o programa mostrou erros superiores (na ordem dos 8%) ao nível da redistribuição de esforços aquando da plastificação de seções e nos valores de carga última, no entanto, o mecanismo de colapso foi sempre corretamente reconhecido. Uma vez que não é tida em conta interação do esforço axial e se admite plastificação apenas por flexão, é possível que a deformação axial (a qual é inevitável neste método) seja uma das razões de tais erros. Por forma a verificar tal influência, foi aumentada, consideravelmente, a rigidez axial tendo sido notória uma ligeira diminuição dos erros (na ordem dos 2%), porém, não foi suficiente para validar tais casos.

Embora já seja possível, com este programa, reconhecer mecanismos de colapso, é necessário um trabalho mais pormenorizado nesta questão por forma melhor entender a origem dos erros existentes.

4.4 Carregamento variável no tempo

Para além das análises estáticas e elasto-plásticas incrementais apresentadas, faz sentido estender o leque de aplicações do Método das Partículas Finitas à análise dinâmica de estruturas.

Avaliar o comportamento de uma estrutura sob ações variáveis no tempo permite prever, por exemplo, o comportamento de uma estrutura sob a ação de um sismo e, com isso, dimensionar a estrutura em conformidade.

Nesta dissertação, pretendeu-se assim mostrar também a capacidade do Método de Partículas Finitas em avaliar estruturas sob a ação de cargas variáveis no tempo, mais concretamente carregamentos harmónicos do tipo sinusoidal. A sobreposição de diferentes ondas sinusoidais, com diferentes

amplitudes e frequências de excitação, permite simular adequadamente qualquer tipo de ação variável no tempo, pelo que é suficiente validar o método para a ação de uma única onda sinusoidal.

Por forma a validar o método neste tipo de análises e criar uma base no programa na qual futuramente possam ser desenvolvidas novas funcionalidades, análises dinâmicas um pouco mais complexas ou mesmo um melhoramento do trabalho feito, serão apresentados neste subcapítulo os resultados de simulações de um caso de estudo com apenas um grau de liberdade e comparados tais resultados com as respetivas soluções exatas.

4.4.1 Osciladores de um grau de liberdade

Por forma a controlar o amortecimento aplicado à estrutura, bem como escolher as partículas nas quais seria aplicado, optou-se por aplicar amortecimento viscoso nas análises desta seção, pois ao contrário do amortecimento cinético (Liew, Mele et al. 2016), o amortecimento viscoso depende de parâmetros ajustáveis (Pajand e Hakkak 2006).

Para conseguir simplificar o comportamento da estrutura, optou-se por restringir o seu movimento apenas a um grau de liberdade, mais concretamente a translações horizontais segundo a direção do eixo x (Figura 57).

Na seção 3.2.5, foi apresentada a equação (35), a qual representa o movimento de osciladores amortecidos para três graus de liberdade. Reescrevendo a referida equação aplicada apenas ao grau de liberdade pretendido, obtém-se a equação geral de osciladores amortecidos segundo a direção x,

$$m\ddot{x} + c\dot{x} + kx = F(t) \tag{51}$$

onde m é a massa, c o coeficiente de amortecimento, k a rigidez a estrutura e $F(t)$ a ação variável no tempo sobre a estrutura.

A equação (51) pode ainda ser reescrita dividindo todos os termos pela massa, obtendo-se,

$$\ddot{x} + 2\zeta p\dot{x} + p^2 x = \frac{F(t)}{m} \tag{52}$$

onde ζ é o coeficiente de amortecimento e p é a frequência própria da estrutura (ou do oscilador), sendo ambos representados pelas seguintes expressões,

$$p = \sqrt{\frac{k}{m}} \tag{53}$$

$$\zeta = \frac{c}{2mp} \tag{54}$$

A solução da equação diferencial (52) depende de o valor do coeficiente de amortecimento ser superior, igual ou inferior à unidade, correspondendo a amortecimento sobrecrítico, crítico e subcrítico, respetivamente. Nesta dissertação, serão utilizados valores de amortecimento correspondentes a amortecimento subcrítico, isto é, a estrutura oscilará diminuindo gradualmente a amplitude da resposta, no caso de oscilações não forçadas.

A estrutura escolhida para simular no programa foi um pórtico simples, apenas com movimento horizontal do piso (viga) (Figura 57 a)). Devido ao método utilizado ser um método de análise de partículas, os elementos barra não têm qualquer massa e, por isso, concentrou-se toda a massa da estrutura nas duas partículas de extremidade da viga. Por simplificação de cálculos, optou-se por utilizar uma estrutura equivalente ao pórtico simulado, para a obtenção dos valores exatos (Figura 57 b)).

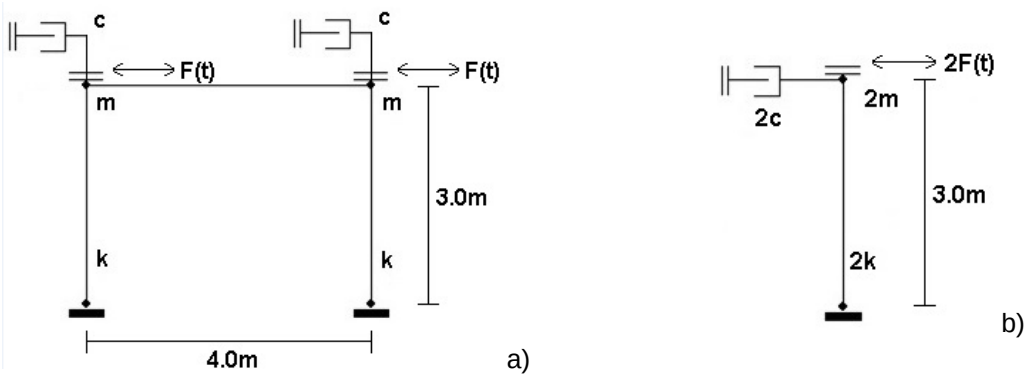


Figura 57: Equivalência entre estruturas exata e simulada no programa (ilustrativo). a) Estrutura simulada no programa. b) Estrutura exata.

Para cálculos exatos, o valor de rigidez utilizado na estrutura equivalente foi referente a uma barra encastrada-encastrada deslizante, expressão (55).

$$k_{exato} = \frac{12EI}{L^3} \tag{55}$$

Relativamente ao elemento viga, o programa não tem opção de tornar um elemento axialmente rígido. Dessa forma, para simular tal característica aumentou-se o valor da rigidez axial, no entanto, a escolha do valor de rigidez axial não é completamente livre. Tal como já referido no subcapítulo 4.2, para valores de rigidez exageradamente elevados, são gerados esforços muito grandes para pequenas deformações e, tais valores de esforços, levam o programa a divergir. Para valores de rigidez muito pequenos, a barra tem deformações muito elevadas e, neste exemplo, as colunas deixam de ter um movimento conjunto, afetando o comportamento global da estrutura.

Neste subcapítulo, será apresentada a simulação da estrutura, exposta anteriormente, em ambos os regimes livre e forçado. Primeiro avaliou-se o comportamento da estrutura para a imposição de um deslocamento inicial e sem amortecimento, ou seja, em regime livre não amortecido. De seguida, simulou-se o comportamento em regime livre amortecido, novamente com a imposição de um deslocamento inicial, mas com a aplicação de amortecimento, e, finalmente, em regime forçado, com amortecimento e sob o efeito de uma ação harmónica.

Para o tipo de amortecimento considerado, amortecimento subcrítico, a solução da equação diferencial (52) para os regimes livre amortecido e não amortecido ($F(t) = 0$), pode ser apresentada na seguinte forma (Leitão 2017),

$$x(t) = e^{-\zeta p t}(\bar{x}\cos(p_d t + \theta)) \tag{56}$$

onde p_d é a frequência amortecida e $\bar{x}$ e θ representam as condições iniciais, sendo as suas expressões apresentadas de seguida,

$$p_d = p\sqrt{1-\zeta^2} \tag{57}$$

$$\bar{x} = \frac{\sqrt{(\dot{x}_0 + \zeta p x_0)^2 + p_d^2 x_0^2}}{p_d} \tag{58}$$

$$\theta = arctg\left(\frac{\dot{x}_0 + \zeta p x_0}{p_d x_0}\right) \tag{59}$$

Regime Livre não Amortecido

Na ausência de ação exterior e amortecimento, a equação (56) toma uma forma simplificada, pois os valores de ζ e $F(t)$ são nulos.

Nestas condições, embora se esteja na situação de amortecimento subcrítico ($\zeta < 1$), havendo assim comportamento oscilatório, o facto de o valor de amortecimento ser zero, faz com que tal oscilação seja constante ao longo do tempo, e, por isso, a estrutura repete infinitamente tal oscilação.

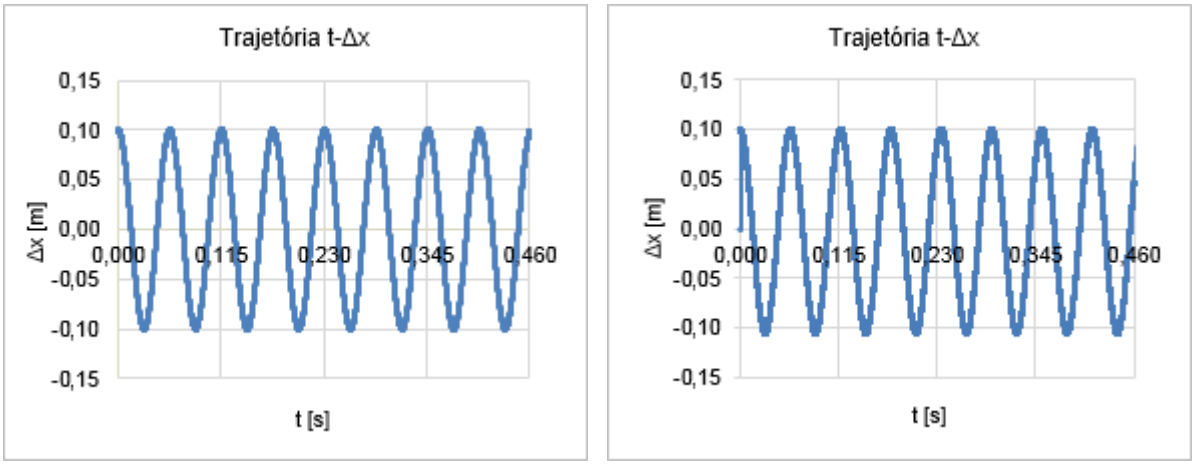
Com base na estrutura da Figura 57 a), foi feita uma simulação com as características apresentadas no quadro seguinte,

Quadro 6: Características da simulação em regime livre não amortecido.

m [kg]	x_0 [m]	c [N.s/m]	F(t)
50*	0,1	0	0

*Repartição de 50% para cada extremidade da viga do pórtico.

Na Figura 58, apresenta-se as trajetórias de deslocamento exata e extraída do programa, correspondendo a trajetória extraída do programa à partícula da ligação viga-coluna da direita (Figura 57 a)).



a) b)

Figura 58: Trajetórias de deslocamento, ao longo do tempo, segundo a direção x em regime livre não amortecido. a) Exata. b) Extraída do programa.

Através das trajetórias é visível a proximidade de comportamento entre os valores exatos e os valores obtidos do programa. O comportamento oscilatório constante associado ao regime em questão é confirmado e os valores de frequência são próximos (Quadro 7).

Quadro 7: Período e frequências em regime livre não amortecido.

	T [s]	f [Hz]	p [rad/s]	ζ
Valores exatos	0,0577	17,336	108,924*	0
Resultados do Programa	0,0585	17,094	107,405	0
Erro [%]	1,39	1,40	1,39	0,00

*Calculado usando a expressão (53).

Regime Livre Amortecido

Neste regime, tal como no anterior, continua a não existir a aplicação de carregamento variável, no entanto, passa a haver amortecimento. A solução da equação (52) toma assim a forma da expressão (56).

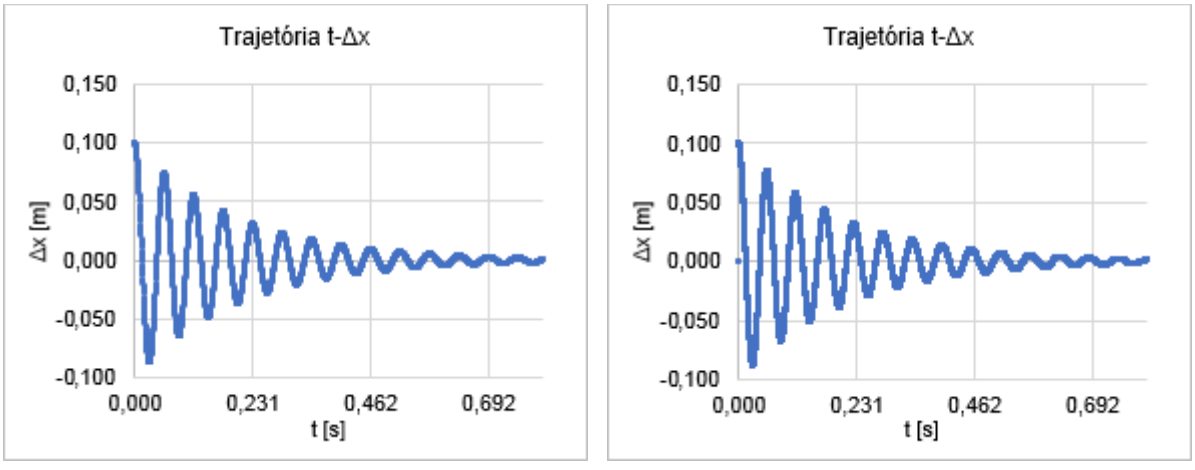
Em termos do comportamento da estrutura, é esperado que este seja oscilatório, devido ao amortecimento subcrítico, com diminuição de amplitude ao longo do tempo por efeito do amortecimento.

Nas Figura 59 e Figura 60, são apresentadas as trajetórias deste regime, bem como uma comparação em detalhe entre os valores obtidos do programa e os valores exatos.

Quadro 8: Características da simulação em regime livre amortecido.

m [kg]	x_0 [m]	c [N.s/m]	F(t)
50*	0,1	500*	0

*Repartição de 50% para cada extremidade da viga do pórtico.



a) b)

Figura 59: Trajetórias de deslocamento, ao longo do tempo, segundo a direção x em regime livre amortecido. a) Exata. b) Extraída do programa.

Para um maior detalhe foram sobrepostas as trajetórias da Figura 59. Admite-se, na solução exata, o termo harmónico da equação (56) unitário, por forma a comparar apenas amplitudes máximas e não sobrecarregar a imagem.

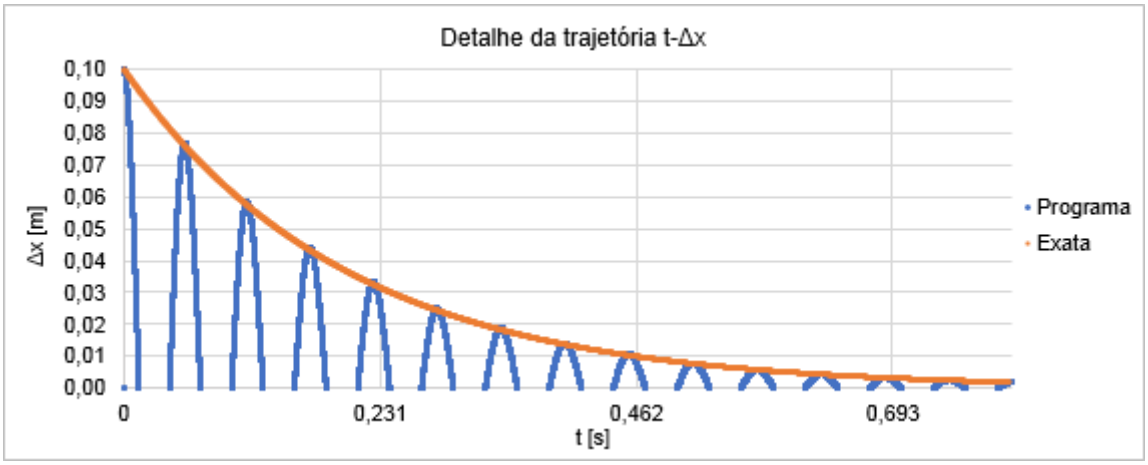


Figura 60: Comparação de amplitudes máximas em regime livre amortecido.

Nas Figura 59 e Figura 60, é possível verificar uma grande proximidade entre resultados exatos e os obtidos do programa. A trajetória gerada pelo programa apresenta-se de acordo com o regime simulado, tendo um comportamento oscilatório que decresce ao longo do tempo. O período e frequência de oscilação, bem como algumas das amplitudes máximas, podem ser consultados no Quadro 9 e no Quadro 10.

Quadro 9: Período e frequências em regime livre amortecido.

	T_d [s]	**p_d [rad/s]**	**ζ**
Valores exatos	0,0577	108,810 *	0,0459 **
Resultados do Programa	0,0568	110,620	0,0452
Erro [%]	1,56	1,66	1,53

*Calculado usando a expressão (53). **Calculado usando a expressão (54).

Quadro 10: Amplitudes máximas em regime livre amortecido.

	Amplitude [m]				
	2º pico	**3º pico**	**4º pico**	**5º pico**	**6º pico**
Valores exatos	0,07492	0,05613	0,04206	0,03151	0,02361
Resultados do Programa	0,07671	0,05830	0,04406	0,03317	0,02493

O valor de amortecimento resultante da simulação do programa, apresentado no Quadro 9, foi calculado recorrendo ao método do decremento logarítmico. Tomando os valores de deslocamento de dois picos sucessivos (separados por T_d) e admitindo que o decremento é logarítmico, obtém-se a seguinte expressão,

$$2\pi\zeta = ln\left(\frac{x_i^{max}}{x_{i+j}^{max}}\right), \qquad com \ \ \frac{2\pi}{T_d} = p_d \tag{60}$$

onde x_i^{max} e x_{i+j}^{max} são os deslocamentos dos ciclos sucessivos i e $i+j$, respetivamente.

Regime Forçado Amortecido

Para simular de forma simplificada o comportamento das estruturas sujeitas a ações exteriores variáveis no tempo, foi induzida, na estrutura da Figura 57 a), uma excitação ao nível do grau de liberdade sob a forma de duas forças harmónicas, aplicadas uma em cada massa, e cuja expressão geral é a seguinte (Azevedo e Proença 1991),

$$F(t) = \bar{F}\cos(wt) \tag{61}$$

onde $\bar{F}$ é a amplitude máxima da ação e w a sua frequência angular.

Substituindo a equação (61) no respetivo termo da equação (52) e resolvendo a equação diferencial, obtém-se a solução correspondente à trajetória da estrutura. No entanto, neste regime, a solução é composta pela sobreposição de duas soluções, a homogénea (correspondente à solução do regime livre amortecido) e a particular (associada à aplicação da ação variável). Tendo a solução homogénea já sido apresentada anteriormente (equação (56)), a solução particular pode ser obtida pela seguinte expressão,

$$x_p(t) = \bar{x}_p\cos(wt - \emptyset) \tag{62}$$

onde $\bar{x}_p$ é a amplitude máxima da solução em regime permanente, β_1 é o fator de amplificação dinâmica e $\emptyset$ é o desfasamento entre a ação e a resposta da estrutura, dados pelas expressões (63) a (65).

$$\bar{x}_p = \beta_1 \frac{\bar{F}}{k} \tag{63}$$

$$\beta_1 = \frac{1}{\sqrt{(1-\overline{w}^2)^2 + (2\zeta\overline{w})^2}} \tag{64}$$

$$\emptyset = arctg\left(\frac{2\zeta\overline{w}}{1-\overline{w}^2}\right) \tag{65}$$

com,

$$\overline{w} = \frac{w}{p}$$

Relativamente à resposta da estrutura neste regime, é esperado um comportamento oscilatório que na fase inicial seja influenciado maioritariamente pela solução em regime transitório (associada às condições iniciais da estrutura). De seguida, e à medida que se começa a fazer sentir o efeito da força aplicada, atinge-se o regime permanente. Para obter tal comportamento e ver a sobreposição de soluções na trajetória do programa, foi aplicado também um deslocamento inicial à estrutura. As condições iniciais são apresentadas no Quadro 11.

Quadro 11: Características da simulação em regime forçado.

m [kg]	x_0 [m]	c [N.s/m]	F(t) [kN]
50*	0,1	500*	10.cos(wt)*

*Repartição de 50% para cada extremidade da viga do pórtico.

Tendo definida a ação exterior que irá atuar sobre a estrutura, e recorrendo às expressões anteriores, é possível obter a resposta da estrutura. No entanto, é importante garantir que a estrutura responde à ação sempre em segurança, sem deslocamentos acima da sua capacidade. Para tal, é necessário saber qual o comportamento natural da estrutura, mais concretamente a sua frequência própria, pois assim consegue-se evitar que ambas as frequências, da ação e da estrutura, sejam próximas, pois caso contrário poderiam surgir efeitos de ressonância e, com esses, o possível colapso da estrutura (Azevedo e Proença 1991).

Em casos simples como o apresentado neste subcapítulo, (Figura 57 a)), é possível, através da simulação do programa, saber a frequência natural da estrutura através da obtenção da curva β_1. Esta curva representa a variação do coeficiente de amplificação dinâmica, β_1, em função da frequência angular da ação. Os deslocamentos dinâmicos da estrutura serão maiores quando as frequências da ação e da estrutura forem próximas, situação que se quer evitar como já referido.

Os coeficientes de amplificação dinâmica são também úteis por representarem a relação entre os deslocamentos dinâmicos e os deslocamentos estáticos da estrutura.

Nesta seção, simulou-se a estrutura sob o efeito de uma ação harmónica com diferentes frequências (equação (61)) e, com base nos deslocamentos dinâmicos obtidos, calcularam-se os respetivos valores de coeficiente de amplificação dinâmica. Depois de obtida a curva β_1, foram comparados os resultados com os valores exatos (Figura 61) e aproximada a frequência natural da estrutura (Figura 62).

Por simulação da estrutura com aplicação da força exterior com frequência nula (força constante no tempo), obteve-se o deslocamento estático, Δx_e, da estrutura e com este, juntamente com os valores de deslocamento dinâmico obtidos também de simulações, obtiveram-se os valores de coeficientes de amplificação dinâmicos (Quadro 12).

Quadro 12: Frequências de ação, valores de deslocamento dinâmico e correspondentes coeficientes de amplificação dinâmica.

f [Hz]	w [rad/s]	Δx_d [m]	Δx_e [m]	β_1	$\beta_{1\ exato}$
5,00	31,42	0,0184	0,0168	1,097	1,090
10,00	62,83	0,0260		1,549	1,496
12,50	78,54	0,0347		2,069	2,068
15,00	94,25	0,0577		3,440	3,815
16,00	100,53	0,1105		6,586	5,911
17,00	106,81	0,1337		7,969	10,291
18,00	113,10	0,1556		9,275	8,016
19,00	119,38	0,0598		3,564	4,400
20,00	125,66	0,0493		2,939	2,855
22,50	141,37	0,0258		1,540	1,423
25,00	157,08	0,0170		1,014	0,916
30,00	188,50	0,0100		0,598	0,498

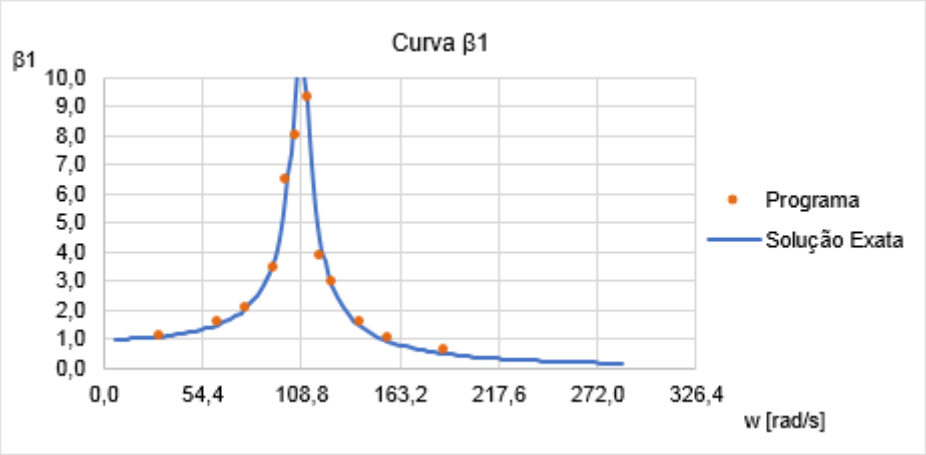


Figura 61: Comparação de valores exatos e experimentais da curva β_1.

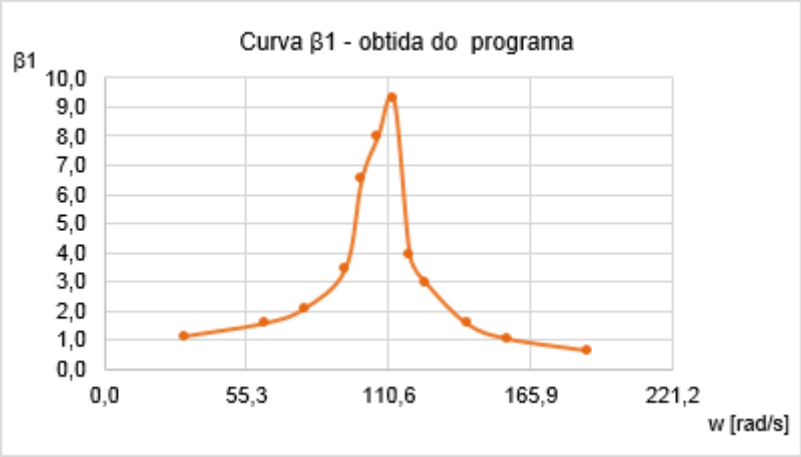


Figura 62: Valores da curva β_1 obtidos do programa.

Pela análise do Quadro 12 e da Figura 61, verifica-se que existem diferenças entre os valores dos coeficientes de amplificação exatos e os obtidos do programa. No entanto, para frequências de ação abaixo dos 16Hz e acima dos 19Hz, aproximadamente, tais diferenças diminuem e os resultados são mais aceitáveis.

Avaliando a Figura 62, é possível ver a capacidade do programa em reconhecer o comportamento da estrutura sob ação de forças harmónicas, sendo possível detetar a frequência natural da estrutura com alguma proximidade. No caso estudado, a solução exata da frequência natural da estrutura é de 108,81rad/s e, pela Figura 62, obtém-se o valor, aproximadamente, de 110,6 rad/s (erro de 1,65%). É possível também verificar que ambos os valores de frequência própria amortecida obtidos do programa, em regime livre amotecido e em regime forçado, são bastante próximos (Quadro 9).

Ao longo do subcapítulo 4.4, foi simulada uma estrutura porticada por forma a mostrar a capacidade do programa em avaliar comportamentos dinâmicos e, com isso, oferecer uma ferramenta didática na qual alunos possam explorar conceitos de dinâmica de estruturas. No entanto, o trabalho feito foi apenas introdutório e necessita de ajustes/desenvolvimentos para que futuramente, não só ofereça melhores resultados, como também mais funcionalidades neste tipo de análises.

4.5 Amortecimento cinético e amortecimento viscoso

Tal como apresentado no subcapítulo 3.2.5, a obtenção de soluções estáticas através do método das partículas (que na realidade não é mais do que a implementação da segunda lei de Newton) recorre à aplicação de amortecimento no sistema de partículas por forma a introduzir a necessária dissipação de energia. No caso agora em apreço, o que se pretende é introduzir amortecimento específico associado aos osciladores.

Nesta dissertação, foi utilizado o amortecimento cinético para obtenção de soluções estáticas (subcapítulo 4.2) e amortecimento viscoso em simulações com carregamentos variáveis (subcapítulo 4.4). Neste programa, é também utilizado amortecimento viscoso na obtenção se soluções estáticas em treliças, através do tipo de estrutura "*truss*" (referente à dissertação anterior).

Será então feita, neste subcapítulo, uma análise de ambos os tipos de amortecimento, com o intuito de se perceber as diferenças, bem como vantagens e desvantagens de ambas as formulações. Por forma aos resultados serem comparáveis, simulou-se a mesma estrutura recorrendo a ambos os tipos de estrutura do programa ('*truss*' e '*beam*'). A estrutura analisada foi a treliça apresentada na Figura 63, com perfis tubulares circulares Celsius® 355 e sujeita a uma força horizontal de 10kN.

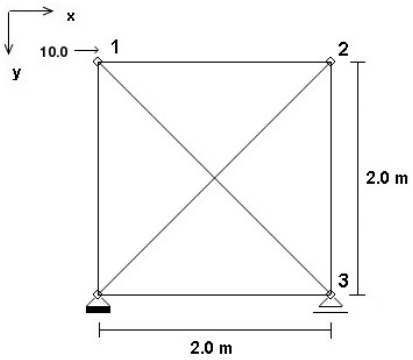


Figura 63: Configuração indeformada em ambos os tipos de análise, '*truss*' e '*beam*'.

Ao contrário das estruturas "*truss*", o modo de análise das estruturas "*beam*" não é inteiramente dedicado a estruturas articuladas. No entanto, pela opção de rotular partículas ('*create hinge*') no menu de propriedades de cada partícula), também é possível simular tais estruturas.

Antes de avaliar ambos os amortecimentos, serão apresentadas de seguida as deformadas, diagramas de esforço axial e deslocamentos, obtidos em ambas as simulações.

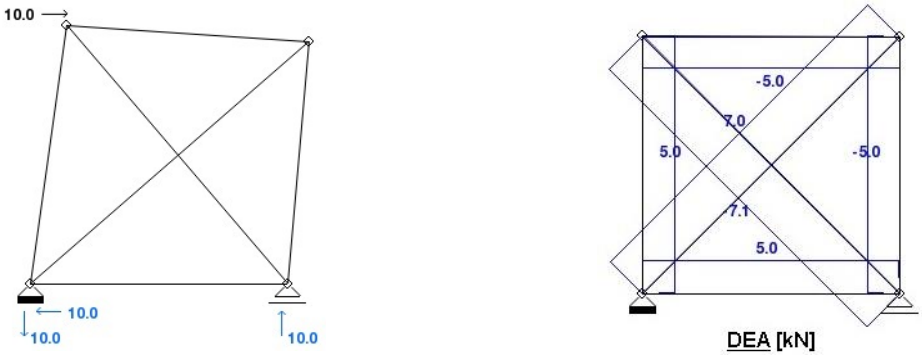


Figura 64: Deformada (fator de escala da deformada -1000) e diagrama de esforço axial (fator de escala do diagrama -5) da estrutura "*beam*".

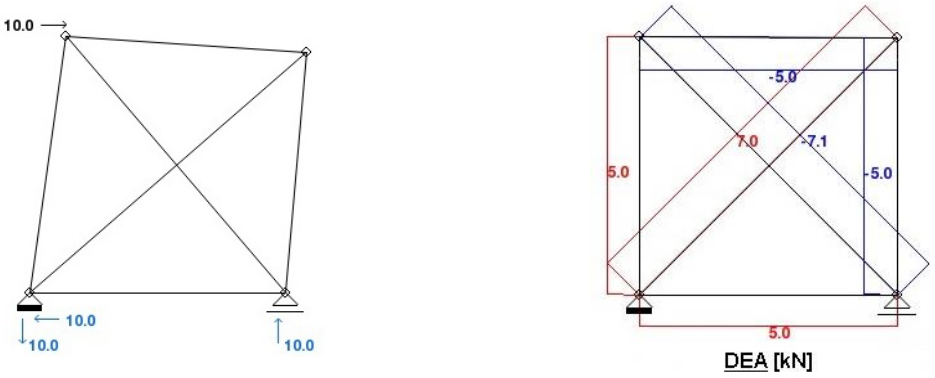


Figura 65: Deformada (fator de escala -1000) e diagrama de esforço axial (fator de escala -5) da estrutura "*truss*".

Quadro 13: Deslocamentos obtidos em ambas as simulações "*beam*" e "*truss*".

Estrutura	**Partícula**					
	1		2		3	
	Δx [m]	**Δy [m]**	**Δx [m]**	**Δy [m]**	**Δx [m]**	**Δy [m]**
'*beam*'	2,929E-4	-6,071E-5	2,323E-4	6,046E-5	6,062E-5	0,000
'*truss*'	2,920E-4	-6,018E-5	2,312E-4	6,088E-5	6,018E-5	0,000
Valores exatos	2,838E-4	-5,870E-5	2,251E-4	5,870E-5	5,870E-5	0,000
Erro '*beam*' [%]	3,21	3,42	3,20	3,00	3,27	0,00
Erro '*truss*' [%]	2,89	2,52	2,72	3,71	2,52	0,00

Para análise dos amortecimentos, recorreu-se a trajetórias extraídas da simulação e referentes à informação da partícula 1 (Figura 66 a Figura 71).

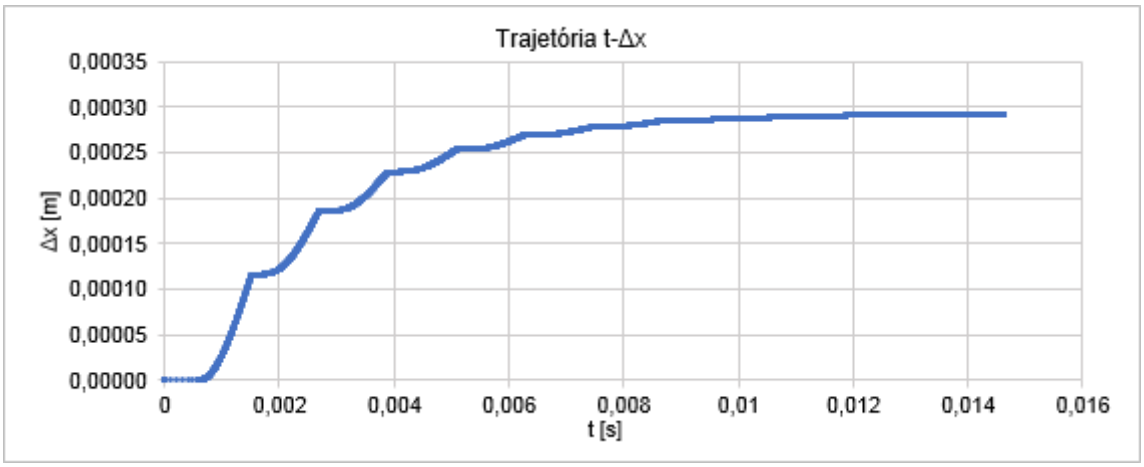


Figura 66: Trajetória da análise '*beam*'- Ponto 1. Deslocamento segundo x.

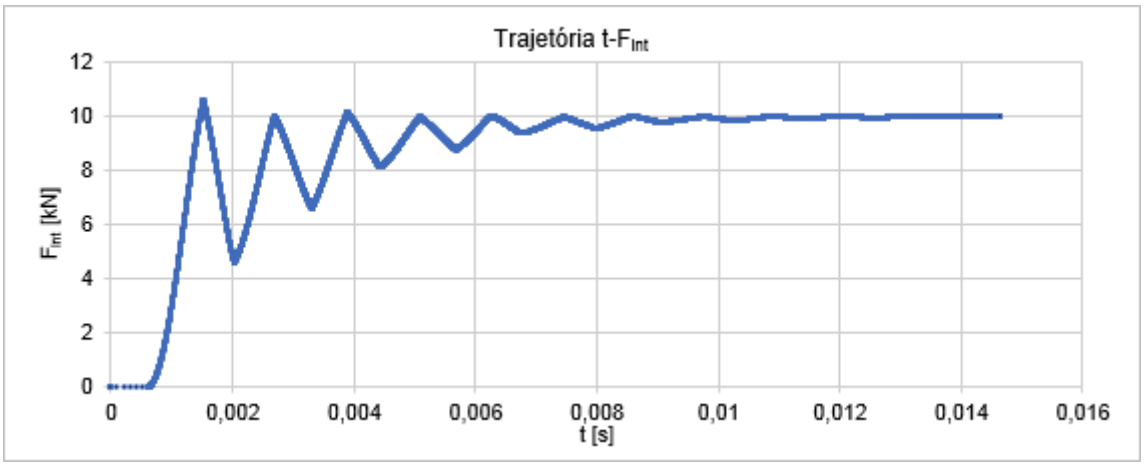


Figura 67: Trajetória da análise '*beam*'- Ponto 1. Força interior.

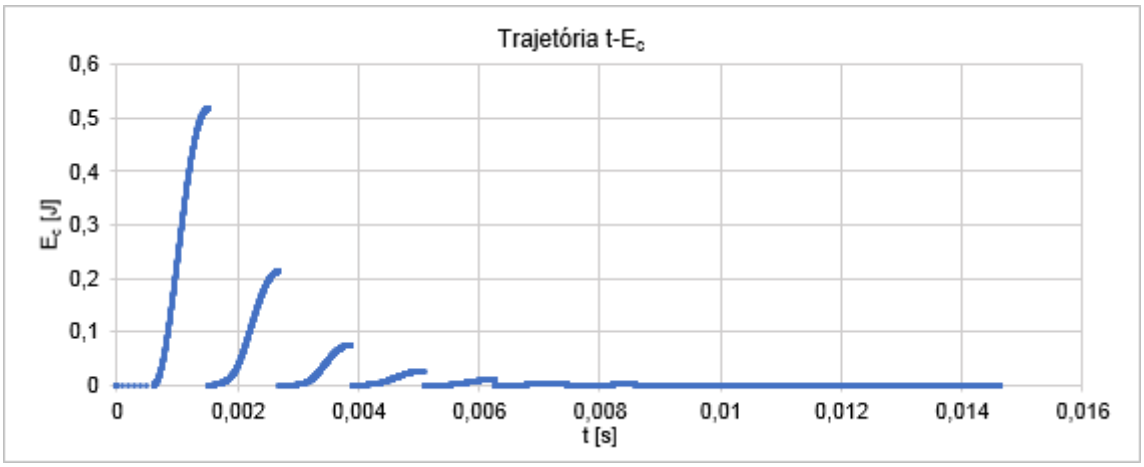


Figura 68: Trajetórias da análise '*beam*'- Ponto 1. Energia cinética.

Com base na Figura 68, é visível o efeito do amortecimento cinético no comportamento da estrutura, tanto a nível de deslocamento como de forças. Recorrendo este amortecimento à energia cinética do sistema, sempre que é detetado um decréscimo de energia assume-se que, instantes antes, houve passagem por um pico de energia e remete-se o sistema para esse ponto. Com isso, surgem "quedas" de energia (Figura 68 c)) cada vez menores ao longo do tempo que diminuem a velocidade do sistema de partículas. Tal diminuição de velocidade juntamente com o constante equilíbrio de esforços do sistema resulta na convergência da estrutura.

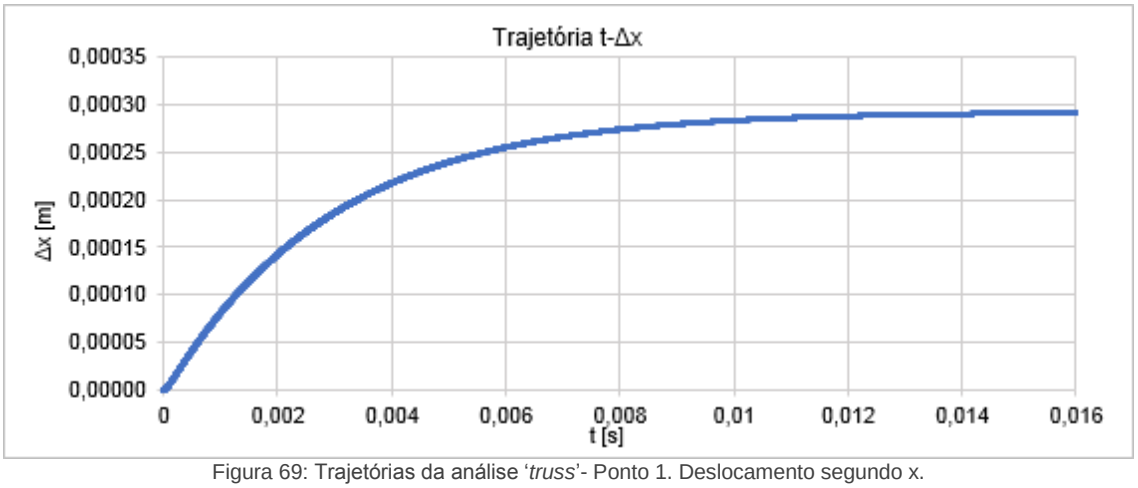


Figura 69: Trajetórias da análise '*truss*'- Ponto 1. Deslocamento segundo x.

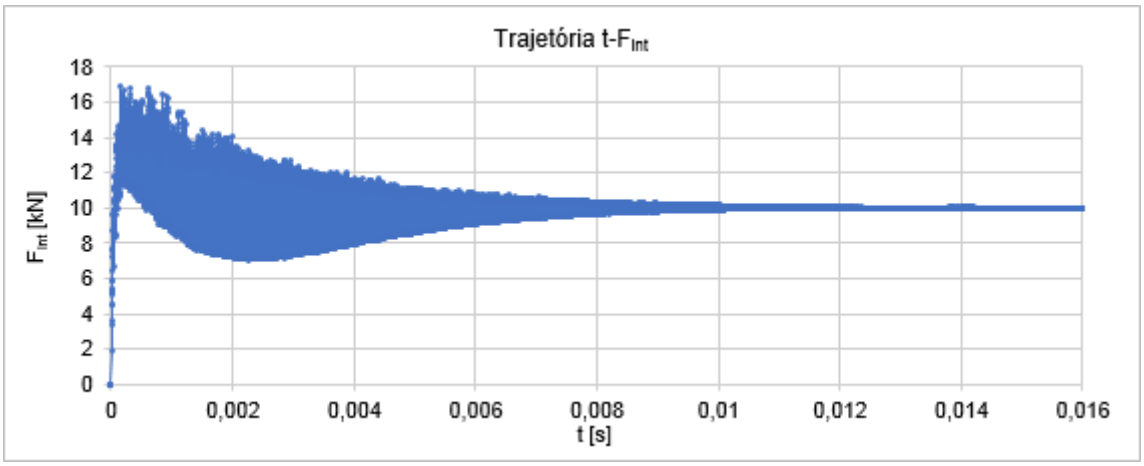


Figura 70: Trajetórias da análise '*truss*'- Ponto 1. b) Força interior.

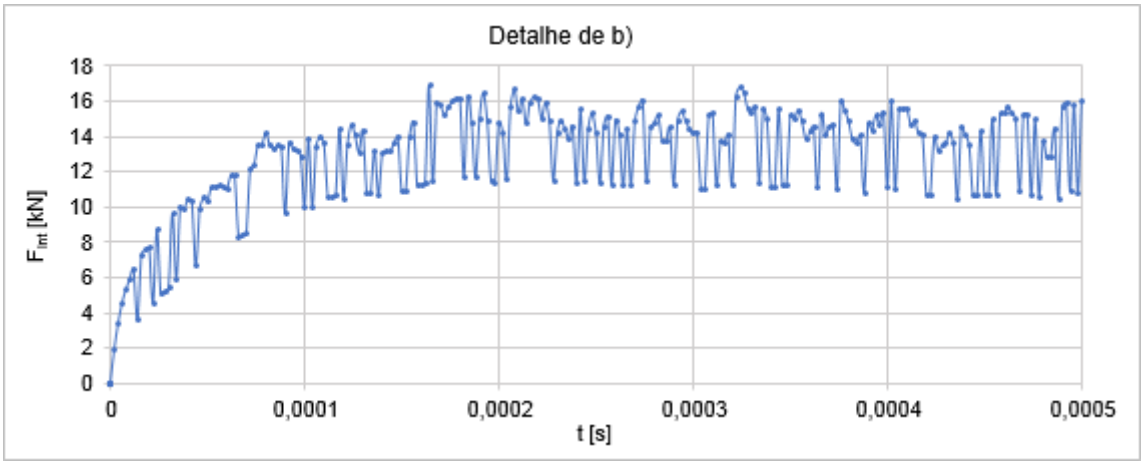


Figura 71: Trajetórias da análise '*truss*'- Ponto 1. c) Detalhe de b).

Relativamente à Figura 71, o amortecimento viscoso atua como um esforço interior que contraria a ação a que a partícula está sujeita. Tal como o amortecimento cinético, o viscoso atua com base nos valores de velocidade, no entanto, enquanto o cinético influencia diretamente a velocidade das partículas, o viscoso atua como esforço interior e dessa forma o seu efeito é bastante mais oscilatório, algo notório na trajetória de força interior.

Embora ambos os amortecimentos ofereçam bons resultados de simulação (Quadro 13), o seu comportamento e influência na estrutura é distinto, fazendo com que tenham diferenças relevantes. Tais diferenças tornam ambos os amortecimentos vantajosos em diferentes situações, tendo cada um a sua utilidade.

O amortecimento cinético é mais direto e consistente em termos de efeito na estrutura e, por isso, para obter soluções estáticas, é mais vantajoso que o viscoso. Outra vantagem do amortecimento cinético é conseguir suportar maiores valores de incremento no tempo, pois por não ter um comportamento tão oscilatório é menos propício a divergências (esta estrutura suportou um Δt máximo de 5E-4s com amortecimento cinético, em comparação com o amortecimento viscoso que divergiu para um Δt de 1E-5s).

Contudo, o amortecimento cinético também tem as suas limitações, sendo uma delas a impossibilidade de calibrar e ajustar o amortecimento, pois depende inteiramente dos valores de velocidade das partículas. Desta forma, caso se queira simular determinadas condições de amortecimento, este não o permite.

Torna-se assim relevante recorrer ao amortecimento viscoso, pois depende de parâmetros que podem ser calibrados e com isso manipular condições de amortecimento para simular determinados comportamentos. Nesta dissertação, foi aplicado amortecimento viscoso no subcapítulo 4.4, pois o objetivo era submeter a estrutura a amortecimento exterior apenas em algumas partículas, bem como controlar a intensidade do seu efeito.

4.6 Desempenho computacional

Neste subcapítulo, serão abordados alguns tópicos sobre o desempenho computacional do programa. Não se tratando de um programa cuja função é ser apenas um auxiliar de cálculo, mas sim um programa interativo e didático, é importante ter em atenção alguns fatores que não só a validade e qualidade dos resultados.

Tempos de simulação, tempos de resposta a ações do utilizador, fluidez gráfica e sensibilidade de resultados são alguns dos pontos importantes num programa para utilização regular, mas ainda mais importantes num programa com propósitos didáticos onde o utilizador pode não ter ainda conhecimento ou sensibilidade sobre os fenómenos que está a avaliar.

Neste programa, que tem por base um método incremental iterativo, um fator de grande importância é a sensibilidade dos resultados, mais concretamente a sensibilidade a divergências. Existem alguns parâmetros a ter em conta nesta questão: o valor de carregamento face aos vãos e perfis utilizados, a discretização da estrutura e o incremento de tempo são condicionantes para o sucesso da solução.

Quanto menor for a discretização, ou maior o comprimento das barras, maior será a massa das partículas de extremidade e consequentemente menor será a aceleração (para igual força), obtida do equilíbrio da partícula através da segunda lei de Newton. Dessa forma, quanto menor for a discretização ou maior o comprimento das barras, maior poderá ser incremento de tempo utilizado.

O valor do carregamento aplicado também influencia a possibilidade de divergência, pois para valores elevados surgirão deformações que originam esforços grandes. Nesse caso pode ser necessário um valor de incremento no tempo mais baixo para o método não dar “saltos” grandes de valores entre iterações e, devido a isso, não conseguir detetar a solução.

Assim, ter uma estrutura com uma discretização menor, ou com maiores comprimentos de barra, pode permitir maiores valores e incremento de tempo, contudo, é também necessário contabilizar a magnitude dos carregamentos, pois poderão impedir o aumento dos valores de incremento de tempo.

Na maioria das simulações, tanto as dimensões da estrutura como o carregamento aplicado são características bem definidas no problema e, por isso, não é possível alterá-las. Assim, para controlar tanto a convergência como o tempo da simulação, pode-se “jogar” com a discretização da estrutura e o valor de incremento de tempo.

Relativamente ao valor do incremento de tempo, o programa utiliza, por defeito, um valor que abrange com sucesso a maioria das simulações possíveis (5E-6 segundos), contudo, pode ser necessário, bem como vantajoso, tal como referido, algum ajuste desse valor dependendo da simulação pretendida.

No quadro seguinte, são apresentados alguns valores referência de incrementos de tempo face ao tipo de simulação e discretização utilizados, por forma a que o utilizador tenha uma referência e consiga, mesmo em fases iniciais de utilização do programa, adaptar-se mais facilmente ao comportamento do método. Contudo, é necessário referir que os valores apresentados são meras referências obtidas experimentalmente com base nas simulações efetuadas ao longo deste trabalho.

Quadro 14: Incrementos de tempo referência.

Discretização (nº de segmentos por barra)	**Análises estáticas**	**Análises dinâmicas**
4	9E-5s	1E-7s*
8	6E-5s	

*nas simulações dinâmicas efetuadas (oscilatórias,) o comportamento da estrutura foi muito sensível ao incremento de tempo e, por isso, é aconselhável a utilização deste valor.

Das análises possíveis neste programa, em análises estáticas e análises elasto plásticas incrementais, a diminuição de discretização não afeta os resultados o suficiente para não serem válidos e, por isso, pode-se utilizar a discretização dependendo do que se pretende da simulação.

Se o objetivo for obter apenas diagramas de esforços, então uma menor discretização pode ser vantajosa, pois reduz consideravelmente o tempo de simulação e os diagramas de esforços, embora com menos valores, ao serem sempre de grau igual ou inferior a um, rapidamente se obtêm valores intermédios por proporcionalidade direta. Se o objetivo for avaliar o comportamento da estrutura em termos de deformada, então uma maior discretização é aconselhada.

Em termos conclusivos, é necessária alguma sensibilidade por parte do utilizador na conjugação dos parâmetros referidos. Por segurança, pode ser utilizado um valor de incremento de tempo pequeno, no entanto, aumentará os tempos de simulação.

De seguida, será avaliado outro fator importante no desempenho de programas computacionais deste tipo: o tempo de simulação. É sem dúvida dos fatores que mais influencia a opinião do utilizador relativamente ao desempenho de um programa.

O tempo de simulação, principalmente neste programa que utiliza um método iterativo, pode depender maioritariamente de três fatores: o valor de incremento de tempo (já referido), a qualidade do código e a capacidade do *hardware*. São referidos, de seguida, os últimos dois fatores.

Relativamente à influência do *hardware* utilizado, e por forma a expor o desempenho deste programa, apresenta-se informação referente aos tempos de simulação de dois computadores com capacidades distintas (Quadro 15), para uma mesma simulação.

Quadro 15: Especificações dos computadores utilizados.

Componente de *Hardware*	**Computador 1**	**Computador 2**
CPU	Intel Core 2 Duo T6500 2.1GHz	Intel Core i5-7200 2.5GHz
RAM	3 GB	8 GB
GPU	NVIDIA GeForce G103M	NVIDIA GeForce 930MX
Memória Gráfica	0,5 GB	2 GB

A estrutura simulada foi uma viga simplesmente apoiada (L=2m), com uma carga pontual a meio vão de 1kN. Nos quadros seguintes, apresentam-se os tempos de simulação dos três módulos do programa, bem como as percentagens comparativamente ao tempo total de simulação por forma a ter uma noção do consumo de tempo de cada módulo.

Quadro 16: Tempos de simulação do computador 1.

	Computador 1		
	Nº de chamadas	**Tempo total [s]**	**Proporção de tempo [%]**
Simulação	1	22,314	100,00
Motor de física	2520	21,243	95,20
Partículas	12600	12,816	57,43
Barras	12600	5,889	26,39
Motor gráfico	132	0,396	1,77*
Função de eventos	4136	1,453	6,51*

*Estas funções são utilizadas noutros modos além do de simulação, estando esse tempo aqui incluído.

Quadro 17: Tempos de simulação do computador 2.

	Computador 2		
	Nº de chamadas	**Tempo total [s]**	**Proporção de tempo [%]**
Simulação	1	3,865	100,00
Motor de física	2520	3,526	91,23
Partículas	12600	2,104	54,44
Barras	12600	1,031	26,68
Motor gráfico	129	0,071	1,84*
Função de eventos	5984	0,463	11,98*

*Estas funções são utilizadas noutros modos além do de simulação, estando esse tempo aqui incluído.

Com base nos tempos totais de simulação, é visível uma diferença significativa apenas pelo uso de um computador com maior capacidade de processamento. O computador dois teve um desempenho aproximadamente 6 vezes mais rápido, o que em simulações de maior complexidade ou com maior número de elementos é definitivamente relevante para o utilizador.

É importante referir que os tempos de simulação do motor gráfico (*Rendering Engine*) e da função de eventos (*Event Handler*) não são referentes apenas à simulação, pois ambos os módulos são utilizados noutros modos do programa e o tempo apresentado engloba toda a utilização dos módulos. Dessa forma, não é possível fazer uma comparação precisa sobre o tempo que os módulos gastam em simulação, mas os valores apresentados permitem ter uma noção bastante aproximada da proporção de tempo que cada módulo consome.

Outro fator de grande importância no desempenho de um programa é o código computacional. Programadores experientes têm incutida a preocupação de desempenho logo desde o início da programação, pois não importa só que o programa execute as funções pretendidas, mas também que o faça em determinados tempos ou em determinados passos, principalmente em programas interativos.

Neste trabalho existem melhoramentos possíveis que serão importantes na rapidez da simulação, principalmente na simulação de estruturas com um grande número de elementos. Através do uso de uma unidade de processamento gráfica (*Graphic Processing Unit* – GPU) ao invés de uma unidade de processamento central (*Central Processing Unit* – CPU) é possível diminuir o tempo total de simulação até 8,5 vezes (Liew, Mele et al. 2016).

Outra possível alteração é o uso de de programação vectorial. Esta consiste no uso de vetores e matrizes para agrupar conjuntos de elementos e realizar operações uma só vez ao invés de utilizar ciclos (ciclos "*for*", por exemplo, que são muito utilizados neste programa), nos quais os cálculos são repetidos elemento a elemento através de operações escalares. Este tipo de programação permite assim efetuar cálculos em grandes quantidades de elementos em menores tempos. A aplicação deste conceito neste trabalho é de todo vantajosa, pois não só irá diminuir os tempos de simulação, bem como facilitar a análise de estruturas mais complexas, com um maior número de elementos.

Neste trabalho, tendo em conta a dimensão das estruturas até então analisadas, o uso de um computador atual, com uma CPU e sem nenhuma alteração do programa, já consegue oferecer simulações 6 vezes mais rápidas. No entanto, para estruturas mais complexas as simulações ainda são um pouco demoradas para o exigido. Conciliando com uma optimização do código, através de programação vectorial por exemplo, a diminuição de tempos de simulação pode ser até 10 vezes, ou superior.

Desta forma, o desempenho actual do programa deve ser avaliado com alguma cautela, não devendo ser um fator determinante na validação das capacidades do método de partículas utilizado ou até mesmo do programa com ferramenta didática, pois, com alguns ajustes "certeiros", mostrou-se que se pode obter um aumento considerável de desempenho.

.

5 Conclusão e trabalho futuro

Neste capítulo, apresentam-se conclusões deste trabalho e algumas dificuldades enfrentadas. Para finalizar, são também referidos possíveis desenvolvimentos futuros para este projeto.

Na área de Engenharia Civil já existem vários programas computacionais focados na análise estrutural, contudo, a sua utilização é direcionada para o âmbito profissional. Em termos de ensino, o seu uso é incentivado em fases mais avançadas do curso, recorrendo-se a meios de ensino mais tradicionais na exposição de conceitos introdutórios de análise estrutural, como por exemplo a mecânica estrutural.

Este projeto é assim um avanço no desenvolvimento de ferramentas computacionais direcionadas ao ensino e no qual tanto os alunos como os professores possam dispor de um complemento no estudo/exposição de conceitos de mecânica estrutural através da visualização gráfica de fenómenos físicos ao longo do tempo.

Academicamente, a componente computacional desta dissertação foi sem dúvida o desafio mais difícil e ao mesmo tempo gratificante que enfrentei. O facto deste trabalho ser a continuação de um programa previamente iniciado, tornou esta experiência bastante desafiante. Não só foi necessário aprender a programar, mas também entender o programa que iria servir de base na qual iria trabalhar, e que teria de compatibilizar para obter um só programa no final. Esta oportunidade permitiu assim desenvolver autonomamente competências que até então não tinham sido exploradas, abrindo portas para um novo conjunto de oportunidades, quer a nível pessoal quer profissional.

Sendo objetivo desenvolver um programa que ofereça não só realismo físico dos fenómenos analisados, mas também a possibilidade de acompanhar visualmente o comportamento gráfico, foram adotados conceitos aplicados na indústria dos vídeo jogos. O funcionamento do programa é assim suportado por três módulos principais: módulo físico (*Physics Engine*), módulo gráfico (*Rendering Engine*) e função de eventos (*Event Handler*).

Cada um deste módulos foi criado e desenvolvido através da linguagem de programação *Python*, tendo o módulo gráfico utilizado também o *Pygame*. Pela extensa documentação existente para estas linguagens, o processo de aprendizagem é facilitado e permite mesmo a programadores menos experientes explorar novos conceitos e funcionalidades com maior criatividade.

Num programa computacional, é importante facilitar a experiência do utilizador de modo a que este possa ter uma postura interativa. O desenvolvimento de uma interface gráfica interativa resultou assim da minha experiência ao utilizar a versão anterior do programa, na qual senti algumas dificuldades em perceber o seu modo de funcionamento e funcionalidades. Com base nessa perspetiva, foi possível colmatar alguns desses aspetos, bem como desenvolver o meu conhecimento e entendimento sobre “motores gráficos” de programas computacionais.

Em termos de resultados, devido à fase em que este projeto se encontra, optou-se por manter o foco principal na extensão de casos de estudo e fenómenos possíveis de simular no programa. Tendo já sido, na versão anterior, avaliado o comportamento de estruturas articuladas, neste trabalho foi escolhido incluir a análise de estruturas reticuladas sujeitas a esforços axiais e de flexão. Com a

inclusão desta análise, consegue-se obter, no conjunto dos dois trabalhos, a grande maioria de estruturas académicas estudadas no curso de Engenharia Civil.

O método aplicado, e no qual se baseou o módulo físico deste programa, foi um método de partículas finitas, cuja base resulta do constante equilíbrio de partículas (massas) que interagem entre si através de barras (elementos de ligação sem massa e com comportamento semelhante a molas). Estes métodos surgem em contraste aos Métodos de Elementos Finitos mais usualmente utilizados, pois através análises de partículas é permitido ao método abstrair-se do comportamento global da estrutura e obter soluções apenas pelo equilíbrio local nas partículas.

Conseguem-se assim, com Métodos de Partículas Finitas, obter bons resultados em problemas simples física e geometricamente lineares ou não lineares. Num programa onde se pretende oferecer um acompanhamento gráfico da simulação, estes métodos são vantajosos, pois ao serem iterativos permitem, quando combinados com um historial no tempo, oferecer ao utilizador um acompanhamento em tempo real do movimento da estrutura.

Neste trabalho, foram avaliadas estruturas reticuladas sujeitas a esforços axiais e de flexão em três análises diferentes: análise estática, análise elasto-plástica incremental e análise dinâmica.

Na análise estática, os resultados obtidos, tanto a nível de deslocamentos como de esforços, foram bastante bons comparativamente com os respetivos valores exatos e tendo em atenção o propósito do programa. Não existindo no programa um critério de paragem automático, o erro pode variar consoante o instante em que o utilizador pare a simulação. Contudo, quando graficamente é visível a convergência da solução (deixa de haver alteração gráfica de valores dos diagramas), os valores de erro são aceitáveis.

Na análise elasto-plástica o programa mostrou resultados interessantes no reconhecimento das rótulas plásticas e mecanismos de colapso. Nos exemplos estudados, o programa reconheceu o mecanismo associado à menor carga última e mostrou uma boa proximidade de valores (com erros máximos de aproximadamente 3%). Em estruturas mais complexas, ou com maior numero de rótulas plásticas, o programa mostrou aumentos de erros (ordem dos 8%) nos valores de carga última, no entanto, o mecanismo de colapso foi sempre corretamente reconhecido.

Em análises dinâmicas, o método mostrou capacidade de reconhecer o comportamento da estrutura, bem como em obter boas aproximações da frequência natural da estrutura, com erros sempre inferiores a 2%. Na obtenção da curva β_1, o programa mostrou algumas limitações em obter os deslocamentos para frequências perto da frequência natural da estrutura, no entanto, através do andamento geral da curva, é possível obter uma boa aproximação da resposta da estrutura.

Embora esta versão do programa já ofereça ao utilizador uma vasta possibilidade de estruturas 2D para analisar, bem como já começa a dar os seus primeiros passos em análises não lineares, ainda é um trabalho que está longe de concluído, pois as possibilidades de melhorias são inúmeras, tanto a nível de qualidade e desempenho do programa como de funcionalidades oferecidas ao utilizador. De seguida, serão indicados alguns melhoramentos possíveis, bem como desenvolvimentos futuros para este trabalho.

Trabalho futuro

Neste trabalho, existem muitos fenómenos e funcionalidades que não foram abordadas e as quais são não só interessantes de tratar, mas também uma mais valia para este projeto. Esta versão, embora já avalie comportamentos não lineares, tanto ao nível de plasticidade como de análises dinâmicas, tais análises são bastante simples e surgiram apenas como uma introdução aos respetivos temas e oportunidade para futuros desenvolvimentos. O principal foco e funcionalidade deste programa ainda é a análise estática de estruturas com comportamentos lineares.

Tendo em conta o ponto de situação no qual este trabalho se encontra, existem três principais abordagens a partir das quais é possível desenvolver o programa.

Primeiramente, e fora o foco físico do programa, penso que seria bastante vantajoso um melhoramento do código do programa. Ambas as versões deste projeto (a primeira versão (Lopes 2015) e a segunda versão correspondente a esta dissertação) foram desenvolvidas em paralelo com o processo de aprendizagem dos intervenientes, pois à medida que se foi ganhando conhecimento foram sendo ajustados os objetivos e funcionalidades, sempre na tentativa de melhorar o programa e chegar mais longe neste projeto. Tendo tal aprendizagem sido autónoma, existem vários aspetos que podem ser otimizados, tal como referido no capítulo 4.6, e sendo um programa interativo com foco didático, é bastante importante oferecer ao utilizador tempos de resposta, e de simulação, mais rápidos.

Relativamente aos fenómenos físicos, e uma vez que o programa já simula estruturas reticuladas gerais, um desenvolvimento bastante interessante seria introduzir a análise de instabilidade e encurvadura de colunas, pois não só completava a análise de estruturas 2D, bem como é um tema de grande importância, estudado em várias disciplinas do curso de Engenharia Civil.

Outra vertente bastante interessante é a continuação do desenvolvimento de análises dinâmicas com a introdução, por exemplo, de funcionalidades que reconheçam a frequência natural da estrutura, deformadas dos modos de vibração e respetivas frequências modais em estruturas 2D, de um e dois graus de liberdade. Com o que o programa já oferece, tal desenvolvimento iria englobar praticamente todos os conceitos de dinâmica de estruturas lecionados no curso de Engenharia Civil, tornando-se numa ferramenta de grande utilidade para alunos iniciantes, experientes, bem como para professores como ferramenta de exposição de conceitos.

Além da continuação do trabalho já desenvolvido neste programa, há também a possibilidade de introduzir a análise de outros tipos de estruturas, como por exemplo membranas, vigas com espessura ou até mesmo a introdução de uma terceira dimensão no programa por forma a analisar estruturas em 3D. Esta última opção requerá uma reformulação do “motor gráfico” do programa, uma vez que o agora utilizado apenas suporta visualização 2D.

Por fim, sendo objetivo que este se torne um programa interativo para utilização corrente, existem sempre melhoramentos a nível de interface gráfico e funcionalidades que podem ser feitos para tornar o programa mais “*user friendly*”.

Bibliografia

Alamatian, J. (2012). "A new formulation for fictitious mass of the Dynamic Relaxation method with kinetic damping."

Appleton, J. (2013). Estruturas de Betão, ORION.

Asprone, D., et al. (2014). "Modified Finite Particle Method: Applications to Elasticity and Plasticity Problems ".

Azevedo, J. e J. Proença (1991). Apontamentos de Dinâmica de Estruturas, IST.

Barnes, M. R. (1999). "Form Finding and Analysis of tension Structures by Dynamic Relaxation."

Bathe, K.-J. e S. Bolourchi (1979). "Large displacement analysis of three-dimensional beam structures ".

Expedition (2007) "Expedition workshed." 2017, from http://expeditionworkshed.org/.

Fowler, M. (2003). UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition).

Gregory, J. (2009). Game Engine Architecture.

Leitão, V. (2017). Apontamentos de Mecânica II, IST.

Liew, A., et al. (2016). "Vectorised graphics processing unit accelerated dynamic relaxation for bar and beam elements."

Lopes, P. (2015). Software development for assistance in the learning of structural analysis. Departamento de Engenharia Civil, Instituto Superior Técnico. Mestrado.

Martini, K. (2001). "Non-linear Structural Analysis as Real-Time Animation."

Pajand, M. R. e M. T. Hakkak (2006). "Nonlinear analysis of truss structures using dynamic relaxation."

Papadrakakis, M. (1981). "A Method for the Automatic Evaluation of the Dynamic Relaxation Parameters ".

Rezaiee-Pajanda, M. e H. Rezaeea (2014). "Fictitious Time Step for the Kinetic Dynamic Relaxation Method."

Rossum, G. v. (1991) "Python Software." from https://docs.python.org/2/faq/gui.html.

Shinners, P. (2000) "Pygame Documentation." from https://www.pygame.org/docs/.

Ting, E. C., et al. (2004). "Fundamentals of a Vector Form Intrinsic Finite Element: Part I. Basic Procedure and A Plane Frame Element."

Virtuoso, F. (2012). Apontamentos de Estruturas Metálicas, IST.

Wu, T.-Y. e E. C. Ting (2008). "Large deflection analysis of 3D membrane structures by a 4-node quadrilateral intrinsic element."

Wu, T.-Y., et al. (2009). "Dynamic elastic-plastic and large deflection analyses of frame structures using motion analysis of structures."

Wu, T. Y., et al. (2006). "Large deflection analysis of flexible planar frames."

Yu, Y. e Y.-z. Luo (2009a). "Finite particle method for kinematically indeterminate bar assemblies."

Yu, Y., et al. (2013). "Finite Particle Method for Progressive Failure Simulation of Truss Structures."

Anexo - algoritmo

Devido a limitação do número de páginas deste documento e à grande extensão do algoritmo do programa, este anexo apresenta apenas as funções principais, organizadas por módulo do programa. Tal exposição, serve assim apenas de complemento a um entendimento mais detalhado do funcionamento e organização do programa.

Ciclo principal – *Main loop*

```
game_mode = 'inicial_menu'
state = None
fpm_window = Windows.FPMWindows((800,600))
fpm_window.openWindow()
elements = {'environment': Environment.Environment(), 'particles': [], 'rods': [],
            'grids': []}
while True:
  if game_mode == 'inicial_menu':
    selected_element, game_mode = Modes.menuMode (elements, fpm_window, game_mode)
  elif game_mode == 'controls':
    game_mode = Modes.controlsMode (fpm_window, game_mode)
  elif game_mode == 'about':
    game_mode = Modes.aboutMode (fpm_window, game_mode)
  elif game_mode == 'message':
    selected_element, game_mode, state = Modes.messageMode (elements, game_mode, state)
  elif game_mode == 'simulation':
    game_mode = Modes.simulationMode(elements, fpm_window, game_mode)
  elif game_mode == 'pause':
    selected_element, game_mode = Modes.pauseMode(elements, fpm_window, game_mode)
  elif game_mode == 'properties':
    game_mode = Modes.propertiesMode(elements, fpm_window, game_mode, selected_element)
  elif game_mode == 'save':
    game_mode = Modes.saveMode(game_mode, elements, fpm_window)
  elif game_mode == 'load':
    game_mode, elements, state = Modes.loadMode(game_mode, elements, fpm_window)
  elif game_mode == 'force method':
    game_mode = Modes.forceMethodMode(game_mode, elements, fpm_window)
```

Modos do programa

```
def menuMode (elements, fpm_window, game_mode):
  selected_element = None
  fpm_window.openWindow()
  while game_mode == 'inicial_menu':
    selected_element, game_mode, elements = Events.checkEvents(game_mode, elements,
                                                                fpm_window,
                                                                selected_element)
    fpm_window.reset()
    if game_mode == 'inicial_menu':
      fpm_window.addWord(['Initial Menu', 1, (0,0,0),(2,2)])
      fpm_window.addbuttonMenu(['Simulation',300,200,200,50,(205,205,193),(238,238,224),
                               'Simular'])
      fpm_window.addWord(['Create Structure', 1, (0,0,0),(347,220)])
      fpm_window.addbuttonMenu(['Properties',300,270,200,50,(205,205,193),(238,238,224),
                               'Propriedades'])
      fpm_window.addWord(['Properties', 1, (0,0,0),(365,290)])
      fpm_window.addbuttonMenu(['Controls',300,340,200,50,(205,205,193),(238,238,224),
                               'Comandos'])
      fpm_window.addWord(['Controls', 1, (0,0,0),(372,360)])
      fpm_window.addbuttonMenu(['Load',300,410,200,50(205,205,193),(238,238,224),
                               'Carregar'])
      fpm_window.addWord(['Load File', 1, (0,0,0),(370,430)])
      fpm_window.addbuttonMenu(['About',660,540,100,40,(205,205,193),(238,238,224),
                               'Informar'])
      fpm_window.addWord(['About', 1, (0,0,0),(690,555)])
      fpm_window.update(game_mode)
  return selected_element, game_mode
```

```
def simulationMode(elements, fpm_window, game_mode):
  iteracao = 0
  while game_mode == 'simulation':
    for i in range(0,20):
      iteracao = iteracao + 1
      updateElementsPhysics(elements)
    game_mode = Events.checkEvents(game_mode, elements, fpm_window)
    updateElementsGraphics(elements)
    fpm_window.reset()
    fpm_window.addWord(['Esc - pause', 1, (0,0,0),(10,580)])
    if Environment.Environment.Structure_Type == 'truss':
      fpm_window.addWord([Structure Type: truss', 1, (0,0,0),(665,10)])
    else:
      fpm_window.addWord([Structure Type: beam', 1, (0,0,0),(665,10)])
    sendElementsToWindow(elements, fpm_window)
    fpm_window.update()
  print 'nº iteraçoes:' + str(iteracao)
  return game_mode

def pauseMode(elements, fpm_window, game_mode):
  selected_element = None
  fpm_window.openWindow()
  updateElementsGraphics(elements)
  while game_mode == 'pause':
    selected_element,game_mode,elements=Events.checkEvents(game_mode,elements,fpm_window,
                                                           selected_element)
    fpm_window.reset()
    if game_mode == 'pause':
      fpm_window.addbutton(['Menu',690,570,100,25,(205,205,193),(238,238,224),'Ir_Menu'])
      fpm_window.addWord(['Initial Menu', 1, (0,0,0),(705,577)])
      fpm_window.addbutton(['Save',620,570,60,25,(205,205,193),(238,238,224),'Guardar'])
      fpm_window.addWord(['Save', 1, (0,0,0),(635,577)])
      fpm_window.addWord(['Pause', 1, (255,0,0),(2,2)])
      fpm_window.addWord(['Esc - simulation ', 1, (0,0,0),(10,580)])
      if Environment.Environment.Structure_Type == 'truss':
        fpm_window.addWord(['Structure type: truss', 1, (0,0,0),(665,10)])
      else:
        fpm_window.addWord(['Structure type: beam', 1, (0,0,0),(665,10)])
      sendElementsToWindow(elements, fpm_window)
      fpm_window.update(game_mode)
  return selected_element, game_mode

def propertiesMode(elements, fpm_window, game_mode, selected_element):
  variables = selected_element.getChangeableVariables()
  input_window = Windows.InputWindow(selected_element.name)
  input_window.openWindow()
  selected_box = None
  boxes = createBoxes(input_window, variables)
  input_window.addWord(['Esc - back to pause', 1, (255,255,255),(600,576)])
  while game_mode == 'properties':
    game_mode,selected_box,input_key=Events.checkEvents(game_mode,elements,fpm_window,
                                                        boxes=boxes,
                                                        selected_box=selected_box,
                                                        allow_letters = False)
    if selected_box != None:
      selected_box.write(input_key)
    input_window.reset()
    sendBoxesToWindow(input_window, boxes)
    input_window.update()
  new_variables = {}
  for b in boxes:
    if b.displayed_word !='' and b.displayed_word != None and b.displayed_word != 'truss'\
    and b.displayed_word != 'beam':
      new_variables[b.title] = float(b.displayed_word)
    else:
      new_variables[b.title] = b.displayed_word
  selected_element.changeVariables(new_variables)
  if selected_element.type == 'rod':
    if selected_element.physics.N_segments > 1 and selected_element.graphycal == 'yes':
      selected_element.RodSubDivision(elements)
  Events.changeSelectedElement(elements, fpm_window)
  return game_mode

def controlsMode (fpm_window, game_mode):
  input_window = Windows.InputWindow('controls', 'controls')
  input_window.openWindow()
  write_controls_text(input_window)
```

```
  while game_mode == 'controls':
    game_mode = Events.checkEvents(game_mode, fpm_window)
    if game_mode == 'controls':
      input_window.update()
  return game_mode

def aboutMode (fpm_window, game_mode):
  input_window = Windows.InputWindow('about', 'about')
  input_window.openWindow()
  write_about_text(input_window)
  while game_mode == 'about':
    game_mode = Events.checkEvents(game_mode, fpm_window)
    if game_mode == 'about':
      input_window.update(game_mode)
  return game_mode

def messageMode (elements, game_mode, state):
  selected_element = None
  fpm_window = Windows.FPMWindows((500,200))
  fpm_window.openWindow()
  while game_mode == 'message':
    selected_element,game_mode,elements = Events.checkEvents(game_mode,elements,
                                                              fpm_window,
                                                              selected_element)
    fpm_window.reset()
    if game_mode == 'message':
      if state == 'File not Found':
        fpm_window.addWord(['File not Found', 1, (0,0,0),(200,70)])
        fpm_window.addMessageButton(['Simulation',105,150,140,25,(205,205,193),(238,238,
                                    224),'Simular'])
        fpm_window.addWord(['Create Structure', 1, (0,0,0),(123,157)])
        fpm_window.addMessageButton(['Load',255,150,140,25,(205,205,193),(238,238,224),
                                    'Carregar'])
        fpm_window.addWord(['Load File', 1, (0,0,0),(295,157)])
      else:
        fpm_window.addWord(['Save changes?', 1, (0,0,0),(200,70)])
        fpm_window.addMessageButton(['Quit',125,150,100,25,(205,205,193),(238,238,224),
                                    'Sair'])
      fpm_window.addWord(['Quit', 1, (0,0,0),(160,157)])
      fpm_window.addMessageButton(['Save',275,150,100,25,(205,205,193),(238,238,224),
                                  'Guardar'])
      fpm_window.addWord(['Save', 1, (0,0,0),(310,157)])
      fpm_window.update(game_mode)
  state = None
  return selected_element, game_mode, state

def saveMode(game_mode, elements, fpm_window):
  variables = {'New File Name:': ''}
  input_window = Windows.InputWindow('Save File')
  input_window.openWindow()
  selected_box = None
  boxes = createBoxes(input_window, variables)
  dir_path = os.path.realpath(os.path.dirname(__file__))
  pickle_dir = os.path.join(dir_path, 'Saved Files')
  input_window.addWord(['Esc - back to pause', 1, (255,255,255),(600,576)])
  while game_mode == 'save':
    game_mode,selected_box,input_key = Events.checkEvents(game_mode,elements,fpm_window,
                                                          boxes=boxes,
                                                          selected_box=selected_box,
                                                          allow_letters= True)
    if selected_box != None:
      selected_box.write(input_key)
    input_window.reset()
    sendBoxesToWindow(input_window, boxes)
    input_window.update()
  file_dir = os.path.join(pickle_dir, boxes[0].displayed_word)
  zoom = elements['environment'].zoom
  analysis = elements['environment'].Structure_Type
  damping = elements['environment'].damping
  if boxes[0].displayed_word != '' and file_dir != None:
    with open(file_dir+'.dat', 'wb') as f:
      pickle.dump([elements,zoom,analysis,damping,fpm_window.grid_in_use],f,protocol=2)
      f.close()
  return game_mode
```

```
def loadMode(game_mode, elements, fpm_window):
  variables = {'Open File': ''}
  input_window = Windows.InputWindow('Load File:')
  input_window.openWindow()
  selected_box = None
  state = None
  boxes = createBoxes(input_window, variables)
  dir_path = os.path.realpath(os.path.dirname(__file__))
  pickle_dir = os.path.join(dir_path, 'Saved Files')
  input_window.addWord(['Esc - back to pause', 1, (255,255,255),(600,576)])
  while game_mode == 'load':
    game_mode,selected_box,input_key=Events.checkEvents(game_mode,elements,fpm_window,
                                                        boxes = boxes,
                                                        selected_box=selected_box,
                                                        allow_letters = True)
    if selected_box != None:
      selected_box.write(input_key)
    input_window.reset()
    sendBoxesToWindow(input_window, boxes)
    input_window.update()
  file_name = os.path.join(pickle_dir, boxes[0].displayed_word)
  if not os.path.exists(file_name+'.dat'):
    game_mode = 'message'
    state = 'File not Found'
  elif boxes[0].displayed_word != '' and file_name != None:
    with open(file_name+'.dat', 'rb') as f:
      elements, zoom, analysis, damping, fpm_window.grid_in_use = pickle.load(f)
      f.close()
    Environment.Environment.zoom = zoom
    Environment.Environment.Structure_Type = analysis
    Environment.Environment.damping = damping
  return game_mode, elements, state

def updateElementsPhysics(elements):
  if Environment.Environment.Structure_Type == 'truss':
    for i, p in enumerate(elements['particles']):
      p.updatePhysics_truss(elements['particles'][i+1:])
    for s in elements['rods']:
      s.updatePhysics_truss()
  if Environment.Environment.Structure_Type == 'beam':
    for i, p in enumerate(elements['particles']):
      p.updatePhysics_beam(elements['particles'][i+1:])
    for s in elements['rods']:
      s.updatePhysics_beam()
  elements['environment'].updatePhysics(elements['particles'])

def updateElementsGraphics(elements):
  for i, p in enumerate(elements['particles']):
    p.updateGraphics()
  for s in elements['rods']:
    s.updateGraphics()

def sendElementsToWindow(elements, fpm_window):
  if Environment.Environment.Structure_Type == 'truss':
    for p in elements['particles']:
#'vector' recebe a informação da posição da partícula, dos vetores de reações e forças
aplicadas, das restrições, e texto, organizada de modo a serem enviados para as funções
do Pygame, as quais desenham no ecrã.
      vector = p.propertiesToReadableVector()
      for v in vector:
        if v != []:
          if v == vector[0]:
            fpm_window.addCircle(v)
          if v == vector[1]:
            a = [v[0],v[2],v[3]]
            fpm_window.addArrow(a)
            fpm_window.addWord(v[1])
          if v == vector[2]:
            a = [v[0],v[2],v[3]]
            fpm_window.addArrow(a)
            fpm_window.addWord(v[1])
          if v == vector[3]:
            fpm_window.addConstraint(v)
          if v == vector[4]:
            a = [v[0],v[2],v[3]]
            fpm_window.addArrow(a)
            fpm_window.addWord(v[1])
```

```
          if v == vector[5]:
            a = [v[0],v[2],v[3]]
            fpm_window.addArrow(a)
            fpm_window.addWord(v[1])
    for s in elements['rods']:
# 'vector' recebe a informação da posição das barras e dos diagramas de esforços organizada
de modo a ser enviada para as funções do Pygame, as quais desenham no ecrã.
      vector = s.propertiesToReadableVector()
      if len(vector[0]) == 3:
        fpm_window.addLine(vector[0][0])
        fpm_window.addPolygon(vector[0][1])
        fpm_window.addWord(vector[0][2])
      elif len(vector[0]) == 1:
        fpm_window.addLine(vector[0][0])
      elif len(vector[0]) == 0:
        pass
  for g in elements['grids']:
    fpm_window.addGrid(g.propertiesToReadableVector())
  if Environment.Environment.Structure_Type == 'beam':
    for p in elements['particles']:
      vector = p.propertiesToReadableVector()
      for v in vector:
        if v != []:
          if v == vector[0]:
            fpm_window.addCircle(v)
          if v == vector[1]:
            a = [v[0],v[2], v[3]]
            fpm_window.addArrow(a)
            fpm_window.addWord(v[1])
          if v == vector[2]:
            a = [v[0],v[2], v[3]]
            fpm_window.addArrow(a)
            fpm_window.addWord(v[1])
          if v == vector[3]:
            fpm_window.addConstraint(v)
          if v == vector[4]:
            a = [v[0],v[2],v[3]]
            fpm_window.addArrow(a)
            fpm_window.addWord(v[1])
          if v == vector[5]:
            a = [v[0],v[2],v[3]]
            fpm_window.addArrow(a)
            fpm_window.addWord(v[1])
          if v == vector[6]:
            a = [v[0],v[2], v[3]]
            fpm_window.addArc(a)
            fpm_window.addWord(v[1])
          if v == vector[7]:
            a = [v[0],v[2], v[3]]
            fpm_window.addArc(a)
            fpm_window.addWord(v[1])
    for s in elements['rods']:
      vector = s.propertiesToReadableVector()
      if s.graphycal == 'yes':
        if len(vector[0]) == 3:
          fpm_window.addLine(vector[0][0])
          fpm_window.addPolygon(vector[0][1])
          fpm_window.addWord(vector[0][2])
        elif len(vector[0]) == 1:
          fpm_window.addLine(vector[0][0])
          fpm_window.addHingeCircle(vector[4][0])
          fpm_window.addHingeCircle(vector[4][1])
      if s.graphycal == 'yes':
        if len(vector[1]) == 4:
          fpm_window.addPolygon(vector[1][1])
          fpm_window.addWord(vector[1][2])
          fpm_window.addWord(vector[1][3])
        elif len(vector[1]) == 0:
          pass
        if len(vector[2]) == 4:
          fpm_window.addPolygon(vector[2][1])
          if vector[2][2][3][0] <= vector[2][3][3][0]:
            fpm_window.addWord(vector[2][2])
          else:
            fpm_window.addWord(vector[2][3])
        elif len(vector[2]) == 0:
          pass
```

```
      if len(vector[3]) == 4:
        fpm_window.addPolygon(vector[3][1])
        if vector[3][2][3][0] <= vector[3][3][3][0]:
          fpm_window.addWord(vector[3][2])
        else:
          fpm_window.addWord(vector[3][3])
      elif len(vector[3]) == 0:
        pass
```

Elementos

```
class Particles:
  def __init__(self, original_pos, original_node_rot, category, particle_number = '' ):
    self.name = "particle " + str(particle_number)
    self.delta_pos = [0,0]
    self.external_force = [0,0]
    if Environment.Environment.Structure_Type == 'beam':
      self.radius = 3
    else:
      self.radius = 4
    self.type = 'particle'
    self.category = category
    self.node_rot = original_node_rot
    self.external_moment = 0
    self.appliedForce = [{'existence': 'existe', 'scalar':[0], 'angle':[0]}, [0,0], [0,0]]
    self.appliedMoment = [{'existence': 'existe', 'scalar': 0},0]
    self.physics= PhysicsParticles.Physical_Particle(original_pos,self.radius,category)
    self.graphical=Geometry.Graphical_Particle(original_pos,self.radius,
                                               self.physics.constraint,
                                               original_node_rot, category)
    self.excel_delta_t = []
    self.excel_pos_x = []
    self.excel_pos_y = []
    self.excel_rot = []
    self.excel_F_exterior = []
    self.excel_F_interior = []
    self.excel_E_cinetica = []
    self.excel_M_exterior = []
    self.excel_M_interior = []
    self.excel_E_cinetica_angular = []

  def updatePhysics_truss(self, other_particles):
    self.delta_pos, self.external_force, self.appliedForce =\
                self.physics.updatePhysicalParticle_Truss(other_particles)

  def updatePhysics_beam(self, other_particles):
    self.delta_pos, self.node_rot, self.external_force, self.external_moment,\
    self.appliedForce, self.appliedMoment, hinge =
           self.physics.updatePhysicalParticle_Beam(other_particles)
    self.graphical.hinge = hinge

  def updateGraphics(self):
    if Environment.Environment.excel_export == 1:
      if Environment.Environment.Structure_Type == 'beam':
        self.excel_delta_t.append(self.physics.force_x)
        self.excel_F_interior.append(self.physics.Internal_forces['scalar'][0])
      else:
        self.excel_delta_t.append(self.physics.inc)
        self.excel_F_interior.append(self.physics.internal_forces['scalar'][0])
        self.excel_pos_y.append(self.physics.delta_pos[1]/100)
        self.excel_pos_x.append(self.physics.delta_pos[0]/100)
        self.excel_rot.append(self.physics.delta_node_rot)
        self.excel_E_cinetica.append(self.physics.kinetic_next)
        self.excel_M_interior.append(self.physics.internal_moments/100)
        self.excel_E_cinetica_angular.append(self.physics.angular_kinetic_next)
        self.excel_F_exterior.append([self.physics.external_forces['angle'][0],
        self.physics.external_forces['scalar'][0]])
        self.excel_M_exterior.append(self.physics.external_moments)
    self.graphical.updateGraphicalParticle(self.delta_pos, self.external_force,
                                           self.node_rot,self.external_moment,
                                           self.appliedForce, self.appliedMoment)
```

```
class Rods:
  def __init__ (self, p1, p2, rod_number = '', discontinuity = False):
    self.name = "rod " + str(rod_number)
    self.particles = [p1, p2]
    self.discontinuity = discontinuity
    self.hooke_force = 0
    self.rod_plasticity = False
    self.type = 'rod'
    self.graphycal = 'yes'
    self.physical = 'yes'
    self.M_interior = [0,0]
    self.V_interior = [(0,0),(0,0)]
    self.physics=PhysicsRods.Physical_Rod(self.particles,discontinuity = discontinuity)
    if self.discontinuity == False:
      self.graphical=Geometry.Graphical_Rod(p1.graphical.pos,p2.graphical.pos,
                                            self.particles,rod_number,
                                            self.physics.original_rod_angle)
    self.excel_deltat = []
    self.excel_N = []
    self.excel_V = []
    self.excel_M1 = []
    self.excel_M2 = []
    self.excel_deltaL = []
    self.excel_tetaR = []

  def updatePhysics_truss(self):
    if self.physical == 'yes':
      if self.discontinuity == False:
        self.hooke_force,self.rod_plasticity=self.physics.updatePhysicalRod_Truss()
      else:
        self.physics.updatePhysicalRod_Truss()

  def updatePhysics_beam(self):
    if self.physical == 'yes':
      if self.discontinuity == False:
        self.hooke_force,self.rod_plasticity,self.M_interior,self.V_interior,hinge
            =self.physics.updatePhysicalRod_Beam()
        self.graphical.hinge = hinge
      else:
        self.physics.updatePhysicalRod_Beam()

  def updateGraphics(self):
    if Environment.Environment.excel_export == 1:
      self.excel_deltat.append(self.physics.inc)
      self.excel_N.append(self.physics.interiorN[0][1])
      self.excel_V.append(self.physics.Resultant_V_rod[0][1])
      self.excel_M1.append(self.physics.interiorM[0])
      self.excel_M2.append(self.physics.interiorM[1])
      self.excel_deltaL.append(self.physics.delta_length/100)
      self.excel_tetaR.append(self.physics.rod_angle)
    if self.discontinuity == False:
      self.graphical.updateGraphicalRod(self.particles[0].graphical.pos,
                                        self.particles[1].graphical.pos,
                                        self.rod_plasticity,self.hooke_force,
                                        self.M_interior,self.V_interior)

  def RodSubDivision(self, elements):
    pos_1 = self.physics.attached_particles[0].physics.pos
    pos_2 = self.physics.attached_particles[1].physics.pos
    delta_x = pos_1[0] - pos_2[0]
    if delta_x < 0:
      delta_x = delta_x * -1
    delta_y = pos_1[1] - pos_2[1]
    if delta_y < 0:
      delta_y = delta_y * -1
    inc_x = delta_x / self.physics.N_segments
    inc_y = delta_y / self.physics.N_segments
    division_particles = []
    for p in range(1, int(self.physics.N_segments)):
      if pos_2[0] > pos_1[0]:
        pos_x = pos_1[0] + p*inc_x
      else:
        pos_x = pos_1[0] - p*inc_x
      if pos_2[1] > pos_1[1]:
        pos_y = pos_1[1] + p*inc_y
      else:
        pos_y = pos_1[1] - p*inc_y
```

```
      pos_p = [pos_x, pos_y]
      division_particles.append(pos_p)
    rod_particles = [self.physics.attached_particles[0]]
    for p in division_particles:
      a = Modes.createParticle(p, elements, 'second')
      rod_particles.append(a)
    rod_particles.append(self.physics.attached_particles[1])
    for i, particle in enumerate(rod_particles,1):
      for other_particle in rod_particles[i:i+1]:
        rod = Modes.createRod(particle, other_particle, elements)
        rod.physics.EI = self.physics.EI
        rod.physics.EA = self.physics.EA
        rod.physics.rod_density = self.physics.rod_density
        rod.physics.radius_of_gyration = self.physics.radius_of_gyration
        rod.physics.bending_strength = self.physics.bending_strength
        rod.physics.strength = self.physics.strength
    self.graphycal = 'no'
    self.physical = 'no'
    self.physics.physical = 'no'
    for s in elements['rods']:
      for p in s.physics.attached_particles:
        p.physics.rotacional_inercia = p.physics.calculateRotacionalInercia()
        p.physics.mass = p.physics.calculateMass()

class Environment:
  Structure_Type = 'beam'
  damping = 1
  viscous_damping = 0
  zoom = 1
  excel_export = 0

  def __init__ (self):
    self.name = 'environment'
    self.acceleration = [0, 0]
    self.force = [0, 0]
    self.type = 'environment'
    self.angular_acceleration = 0
    self.moment = 0

  def updatePhysics(self, particles):
    for p in particles:
      p.physics.imposedAccelToForce(self.acceleration)
      p.physics.addExternalForce(self.force)
      if Environment.Structure_Type == 'beam':
        p.physics.imposedAngularAccelToMoment(self.angular_acceleration)
        p.physics.addExternalMoment(self.moment)

class Grid:
  def __init__(self, grid_number = ''):
    self.name = "Grid" + str(grid_number)
    self.type = 'grid'
    self.x_spacing = 50
    self.y_spacing = 50
    self.colour = (200,200,200)
```

Módulo físico – *Physics Engine*

```
class Physical_Particle:
  coef_restitution = 1
  time_step = 0.0000001
  beam_time_step = 0.000005

  def __init__(self, original_pos, radius, category):
    self.constraint = [0,0,0,0]
    self.connected_rods = []
    self.mass = self.calculateMass()
    self.radius = radius
    self.user_applied_accels = {'angle':[0], 'scalar':[0]}
    self.user_applied_forces = {'angle':[0], 'scalar':[0]}
    self.forces = {'angle':[0], 'scalar':[0]}
    self.internal_forces = {'angle':[0], 'scalar':[0]}
    self.external_forces={'angle':[self.user_applied_forces['angle'][0]],
                          'scalar':[self.user_applied_forces['scalar'][0]]}
    self.vels = {'angle':[0], 'scalar':[0]}
    self.accels = {'angle':[0], 'scalar':[0]}
```

```
    if category == 'first':
      self.original_pos=(original_pos[0]*Environment.Environment.zoom,original_pos[1]*
                         Environment.Environment.zoom)
    else:
      self.original_pos = original_pos
    self.pos = list(self.original_pos)
    self.delta_pos=[self.pos[0]-self.original_pos[0],self.pos[1]-self.original_pos[1]]
    self.rotacional_inercia = self.calculateRotacionalInercia()
    self.internal_N_forces = {'angle':[0], 'scalar':[0]}
    self.internal_V_forces = {'angle':[0], 'scalar':[0]}
    self.pos_anterior = list(self.original_pos)
    self.force_type = 0
    self.force_frequency = 1
    self.force_x = 0
    self.accel_type = 0
    self.accel_frequency = 1
    self.user_applied_angular_accels = 0
    self.particuleDampCoef = 250
    self.user_applied_moments = 0
    self.r = 0
    self.moments = 0
    self.internal_moments = 0
    self.external_moments = self.user_applied_moments*100
    self.angular_vels = 0
    self.incremental_angular_vel = 0
    self.angular_accels = 0
    self.node_rot = 0
    self.delta_node_rot = 0
    self.articulate = 0
    self.damping_coeficient = 0
    self.damping_force = {'angle':[0], 'scalar':[0]}
    self.inc = 0
    self.kinetic_next = 0
    self.Internal_forces = {'angle':[0], 'scalar':[0]}
    self.angular_kinetic_next = 0

  def updatePhysicalParticle_Beam(self, other_particles):
    self.particleCollision(other_particles)
    self.force_x = self.force_x + Physical_Particle.beam_time_step
    if self.force_type == 0 and self.accel_type != 0:
      self.external_forces={'angle':[self.user_applied_forces['angle'][0]],
                            'scalar':[self.user_applied_forces['scalar'][0]]}
      self.external_forces['scalar'][0]=(self.user_applied_accels['scalar'][0]/self.mass)
                                         *math.cos(self.accel_frequency*2*math.pi*
                                         self.force_x)
      self.external_forces = supfc.sumVectorsDict(self.external_forces)
    elif self.force_type != 0 and self.accel_type == 0:
      self.external_forces = {'angle':[self.user_applied_forces['angle'][0]],
                              'scalar':[self.user_applied_forces['scalar'][0]]}
      self.external_forces['scalar'][0]=self.user_applied_forces['scalar'][0]*\
                                        math.cos(self.force_frequency*2*math.pi*\
                                        self.force_x)
      self.external_forces = supfc.sumVectorsDict(self.external_forces)
    elif self.force_type != 0 and self.accel_type != 0:
      self.external_forces = {'angle':[self.user_applied_forces['angle'][0]],
                              'scalar':[self.user_applied_forces['scalar'][0]]}
      F=self.user_applied_forces['scalar'][0]*math.cos(self.force_frequency*2*math.pi*\
        self.force_x)
      A=(self.user_applied_accels['scalar'][0]/self.mass)*math.cos(self.accel_frequency*\
        2*math.pi*self.force_x)
      self.external_forces['scalar'][0] = A + F
      self.external_forces = supfc.sumVectorsDict(self.external_forces)
    if self.external_forces['scalar'] == [0.0]:
      applied_force = self.user_applied_forces
      applied_force['existence'] = 'nao existe'
    else:
      applied_force = self.external_forces
      applied_force['existence'] = 'existe'
    if self.external_moments == 0.0:
      applied_moment = {'scalar':self.user_applied_moments}
      applied_moment['existence'] = 'nao existe'
    else:
      applied_moment = {'scalar':self.user_applied_moments}
      applied_moment['existence'] = 'existe'
    self.external_moments = self.user_applied_moments*100
    self.user_applied_accels = supfc.sumVectorsDict(self.user_applied_accels)
    self.imposedAccelToForce([self.user_applied_accels['angle'][0],
```

```
                            self.user_applied_accels['scalar'][0]])
    self.imposedAngularAccelToMoment(self.user_applied_angular_accels)
    self.Internal_forces = self.addForces([self.internal_N_forces,
                                           self.internal_V_forces])
    if Environment.Environment.viscous_damping != 0:
      self.damping_force={'angle':[supfc.normalizeAngle(self.vels['angle'][0]+math.pi)],
                          'scalar':[self.damping_coeficient*self.vels['scalar'][0]]}
      self.Internal_forces = self.addForces([self.Internal_forces, self.damping_force])
    self.forces = self.addForces([self.Internal_forces, self.external_forces])
    reaction_N_T, reactionM, reactionW = self.calculateConstraintReaction_Beam()
    self.forces = self.addForces([self.forces, reaction_N_T])
    self.accels = self.forceToAccel(self.forces)
    if self.constraint[2] != 0:
      self.moments = self.internal_moments + self.external_moments
      self.moments = self.addMoments([self.moments, reactionM])
    else:
      self.moments = self.external_moments + self.internal_moments
      self.angular_accels = self.momentsToAngularAccel(self.moments)
    self.move_Taylor()
    self.anterior_vels = self.vels
    self.vels = self.accelerate_Truss()
    self.anterior_angular_vels = self.angular_vels
    if self.constraint[2] != 0:
      self.angular_vels = self.addAngularVels([self.angular_vels, reactionW])
      self.angular_vels = self.accelerate_Beam()
    if Environment.Environment.damping != 0:
      self.kineticDamping()
    real_external_force=supfc.addVectors([self.external_forces['angle'][0],
                                          self.external_forces['scalar'][0]],
                                          reaction_N_T['angle'][0],
                                          reaction_N_T['scalar'][0]])
    real_external_moment=self.external_moments+reactionM
    reaction_t,reaction_n=supfc.decomposeVector((reaction_N_T['angle'][0],
                                                 reaction_N_T['scalar'][0]),
                                                 -(self.constraint[3])*math.pi/180)
    applied_force = [applied_force, reaction_n, reaction_t]
    applied_moment = [applied_moment, -0.01*reactionM]
    if self.articulate == 1:
      hinge = 'yes'
    else:
      hinge = 'no'
    self.reset()
    return self.delta_pos, self.node_rot, real_external_force, real_external_moment,\
           applied_force, applied_moment, hinge

  def kineticDamping(self):
    vel_anterior = self.anterior_vels['scalar'][0]
    vel_next = self.vels['scalar'][0]
    self.kinetic_anterior = 0.5 * self.mass * pow(vel_anterior,2)
    self.kinetic_next = 0.5 * self.mass * pow(vel_next,2)
    if self.kinetic_next < self.kinetic_anterior:
      self.vels['angle'] = self.accels['angle']
      self.vels['scalar'][0]=0.5*Physical_Particle.beam_time_step*\
                              self.accels['scalar'][0]
      self.position_resetToPeak()
    angular_vel_anterior = self.anterior_angular_vels
    angular_vel_next = self.angular_vels
    self.angular_kinetic_anterior=0.5*self.rotacional_inercia*\
                                  pow(angular_vel_anterior,2)
    self.angular_kinetic_next=0.5*self.rotacional_inercia*pow(angular_vel_next,2)
    if self.angular_kinetic_next < self.angular_kinetic_anterior:
      self.angular_vels = 0.5 * Physical_Particle.beam_time_step * self.angular_accels
      self.rotation_resetToPeak()

  def position_resetToPeak(self):
    Time = Physical_Particle.beam_time_step
    self.pos[0] = self.pos[0]-\
                  1.5*Time*(self.vels['scalar'][0]*math.cos(self.vels['angle'][0]))+\
                  0.5*pow(Time,2)*\
                  (self.accels['scalar'][0]*math.cos(self.accels['angle'][0]))
    self.pos[1] = self.pos[1] -
                  (1.5)*Time*self.vels['scalar'][0]*math.sin(self.vels['angle'][0])+\
                  (0.5)*pow(Time,2)*\
                  self.accels['scalar'][0]*math.sin(self.accels['angle'][0])
```

```
def rotation_resetToPeak(self):
  Time = Physical_Particle.beam_time_step
  self.node_rot=self.node_rot-(1.5)*Time*self.angular_vels+\
                 0.5*Time*self.angular_accels*pow(Time,2)

def move_Taylor(self):
  delta_x=((self.vels['scalar'][0]*math.cos(self.vels['angle'][0])*\
            Physical_Particle.beam_time_step)+self.accels['scalar'][0]*\
            math.cos(self.accels['angle'][0])*\
            pow(Physical_Particle.beam_time_step,2)/2))*100
  delta_y=((self.vels['scalar'][0]*math.sin(self.vels['angle'][0])*\
            Physical_Particle.beam_time_step)+(self.accels['scalar'][0]*\
            math.sin(self.accels['angle'][0])*\
            pow(Physical_Particle.beam_time_step,2)/2))*100
  self.pos[0] = self.pos[0] + delta_x
  self.pos[1] = self.pos[1] + delta_y
  self.delta_pos=[self.pos[0]-self.original_pos[0],self.pos[1]-self.original_pos[1]]
  self.node_rot += (self.angular_vels*Physical_Particle.beam_time_step)+\
                     (self.angular_accels*pow(Physical_Particle.beam_time_step,2)/2)
  if self.node_rot < 1.0e-13:
    self.delta_node_rot = 0
  else:
    self.delta_node_rot = self.node_rot

def calculateMass(self):
  if len(self.connected_rods) != 0:
    mass = 0
    for s in self.connected_rods:
    if s.physics.physical == 'yes':
      if Environment.Environment.Structure_Type == 'truss':
        mass += (s.physics.density) * (s.physics.rest_length)/2
      else:
        mass += (s.physics.rod_density) * (s.physics.rest_length/100)/2
  else:
    mass = 1
  return mass

def calculateRotacionalInercia(self):
  if len(self.connected_rods) != 0:
    inercia = 0
    for s in self.connected_rods:
      if s.physics.physical == 'yes':
        inercia += (s.physics.beam_mass/2) * (s.physics.radius_of_gyration**2)
  else:
    inercia = 1
  return inercia

def forceToAccel(self, force):
  accels = {'angle': [], 'scalar':[]}
  accels['angle'].append(force['angle'][0])
  accels['scalar'].append((force['scalar'][0]*1000)/self.mass)
  accels = supfc.sumVectorsDict(accels)
  return accels

def momentsToAngularAccel(self, moment):
  Moment = moment * 1000 * 0.01
  angular_accel = Moment/(self.rotacional_inercia)
  return angular_accel

def accelerate_Beam(self):
  self.incremental_angular_vel=self.angular_accels * Physical_Particle.beam_time_step
  new_angular_vel = self.angular_vels + self.incremental_angular_vel
  return new_angular_vel

def accelerate_Truss(self):
  incremental_vel  = {'angle':self.accels['angle'],
                      'scalar':[self.accels['scalar'][0]*Physical_Particle.time_step]}
  new_vel = supfc.sumVectorsDict(incremental_vel, self.vels)
  return new_vel

def calculateConstraintReaction_Beam(self):
  constraint_angle_radians = (self.constraint[3]) * math.pi / 180
  reaction_N_T =  {'angle':[0], 'scalar':[0]}
  resultant = supfc.sumVectorsDict(self.forces)
  resultant_t,resultant_n=supfc.decomposeVector((resultant['angle'][0],\
                                                 resultant['scalar'][0]),\
                                                 -1*constraint_angle_radians)
```

```
    vel_t,vel_n=supfc.decomposeVector((self.vels['angle'][0],self.vels['scalar'][0]),\
                                      -1*constraint_angle_radians)
    if self.constraint[0] != 0:
      reaction_N_T['angle'].append(supfc.normalizeAngle(resultant_n[0] + math.pi))
      reaction_N_T['scalar'].append(resultant_n[1])
      self.vels['angle'].append(vel_n[0] + math.pi)
      self.vels['scalar'].append(vel_n[1])
    if self.constraint[1] != 0:
      reaction_N_T['angle'].append(resultant_t[0] + math.pi)
      reaction_N_T['scalar'].append(resultant_t[1])
      self.vels['angle'].append(vel_t[0] + math.pi)
      self.vels['scalar'].append(vel_t[1])
    reaction_N_T = supfc.sumVectorsDict(reaction_N_T)
    reactionM = 0
    reactionW = 0
    if self.constraint[2] != 0:
      reactionM = -1 * (self.internal_moments + self.external_moments)
      reactionW = -1 * self.angular_vels
    return reaction_N_T, reactionM, reactionW

class Physical_Rod:

  def __init__(self, particles, discontinuity = False):
    self.discontinuity = discontinuity
    self.attached_particles = particles
    self.pos1 = self.attached_particles[0].physics.pos
    self.pos2 = self.attached_particles[1].physics.pos
    self.angle_p1_to_p2, self.angle_p2_to_p1 = self.calculateAngles()
    self.rest_length = self.calculateLength()
    self.current_deformation = self.calculateDeformation()
    self.delta_length = self.current_deformation * self.rest_length
    self.stiffness = 164640
    self.damp_coef = 250
    self.density = 0.01
    self.strength = (-0,0)
    self.plasticity = False
    self.delta_elastic_deformation = self.calculateDeltaElasticDeformation()
    self.plastic_deformation = 0
    self.elastic_limit_deformation=[list(self.delta_elastic_deformation)[0]+\
                                    self.plastic_deformation,
                                    (self.delta_elastic_deformation)[1]+\
                                    self.plastic_deformation]
    self.rodHookeForce()
    self.rodDamping()
    self.rodResultantAxialForce()
    self.physical = 'yes'
    self.pos1_anterior = self.attached_particles[0].physics.pos_anterior
    self.pos2_anterior = self.attached_particles[1].physics.pos_anterior
    self.current_lenght = self.rest_length
    self.original_rod_angle = self.calculateRodAngle()
    self.rod_angle = self.original_rod_angle
    self.p1_node_rot = self.attached_particles[0].physics.node_rot
    self.p2_node_rot = self.attached_particles[1].physics.node_rot
    self.EI = 359.1
    self.EA = 216720
    self.dampCoef = 50.0
    self.rod_density = 8.1
    self.radius_of_gyration = 0.0407
    self.beam_mass = self.rest_length * (self.rod_density*0.01)
    self.rod_deformation = [0,0,0]
    self.N_plasticity = False
    self.limit_elastic_axial_deformation = self.LimitAxialElasticDeformation()
    self.M_plasticity = [False, False]
    self.bending_strength = 0
    self.plastic_r = [0,0]
    self.hinge_in_left_end = 0
    self.hinge_in_right_end = 0
    self.interiorM = (0,0)
    self.Resultant_V_rod = [[0,0],[0,0]]
    self.interiorN = [[0,0],[0,0]]
    self.N_segments = 1
    self.rodInternalStresses()
    self.rodToParticles()
    self.inc = 0
```

```
def updatePhysicalRod_Beam(self):
  self.inc = self.inc + PhysicsParticles.Physical_Particle.time_step
  self.updateRodPosition_Beam()
  if self.discontinuity == False:
    hinge = self.rodDeformation()
    self.NplasticityCheck()
    Hinge = [hinge, self.pos1, self.pos2, self.angle_p1_to_p2, self.angle_p2_to_p1]
    self.rodInternalStresses()
    self.rodToParticles()
    return self.hooke_force,self.N_plasticity,self.interiorM,self.Resultant_V_rod,
           Hinge
  else:
    self.rodToParticles()

def updateRodPosition_Beam(self):
  self.pos1 = self.attached_particles[0].physics.pos
  self.pos2 = self.attached_particles[1].physics.pos
  self.p1_node_rot = self.attached_particles[0].physics.node_rot
  self.p2_node_rot = self.attached_particles[1].physics.node_rot

def rodDeformation(self):
  self.current_deformation = self.calculateDeformation()
  self.delta_length = self.current_deformation * self.rest_length
  self.current_lenght = self.rest_length + self.delta_length
  self.angle_p1_to_p2, self.angle_p2_to_p1 = self.calculateAngles()
  self.rod_angle = self.calculateRodAngle()
  hinge = self.calculate_r()
  self.rod_deformation=[self.delta_length,self.attached_particles[0].physics.r,
                        self.attached_particles[1].physics.r]
  return hinge

def calculateRodAngle (self):
  self.angle_p1_to_p2, self.angle_p2_to_p1 = self.calculateAngles()
  if self.attached_particles[0].physics.pos[0] > \
    self.attached_particles[1].physics.pos[0]:
    rod_angle = self.angle_p2_to_p1
  elif self.attached_particles[0].physics.pos[0] < \
    self.attached_particles[1].physics.pos[0]:
    rod_angle = self.angle_p1_to_p2
  else:
    if self.attached_particles[0].physics.pos[1] > \
      self.attached_particles[1].physics.pos[1]:
      rod_angle = self.angle_p2_to_p1
    else:
      rod_angle = self.angle_p1_to_p2
  return rod_angle

def calculate_r(self):
  if (self.rod_angle * self.original_rod_angle) < 0:
    self.original_rod_angle = supfc.normalizeAngle(self.original_rod_angle - math.pi)
  if (self.hinge_in_left_end == 1 or self.hinge_in_right_end == 1) or \
      (self.attached_particles[0].physics.articulate == 1 or \
      self.attached_particles[1].physics.articulate == 1):
    hinge = self.articulateRod()
  else:
    self.attached_particles[0].physics.r=self.p1_node_rot-(self.rod_angle -
                                         self.original_rod_angle)
    self.attached_particles[1].physics.r=self.p2_node_rot - (self.rod_angle -
                                       self.original_rod_angle)
    hinge = 'no hinge'
  return hinge

def NplasticityCheck(self):
  if self.current_deformation > self.limit_elastic_axial_deformation[1] and\
     self.strength[1] != 0:
    self.N_plasticity = True
    self.plastic_deformation=self.current_deformation-\
                             self.limit_elastic_axial_deformation[1]
    self.rod_deformation[0]=self.limit_elastic_axial_deformation[1]*self.rest_length
  elif self.current_deformation < self.limit_elastic_axial_deformation[0] and\
         self.strength[0] != 0:
    self.N_plasticity = True
    self.plastic_deformation=self.current_deformation-\
                             self.limit_elastic_axial_deformation[0]
    self.rod_deformation[0]=self.limit_elastic_axial_deformation[0* self.rest_length
```

```
def MplasticityCheck (self):
  if self.bending_strength != 0:
    if abs(self.M_interior[0]) > abs(self.bending_strength*100):
      self.M_plasticity[0] = True
      if (self.M_interior[0] * (self.bending_strength*100)) > 0:
        self.M_interior[0] = self.bending_strength*100
      else:
        if self.bending_strength > 0 and self.M_interior[0] < 0:
          self.M_interior[0] = -1*(self.bending_strength*100)
        else:
          self.M_interior[0] = self.bending_strength*100
      self.attached_particles[0].graphical.M_plastic_hinge.append(True)
    else:
      self.M_plasticity[0] = False
      self.M_interior[0] = self.M_interior[0]
      self.attached_particles[0].graphical.M_plastic_hinge.append(False)
    if abs(self.M_interior[1]) > abs(self.bending_strength*100):
      self.M_plasticity[1] = True
      if (self.M_interior[1] * (self.bending_strength*100)) > 0:
        self.M_interior[1] = self.bending_strength*100
      else:
        if self.bending_strength > 0 and self.M_interior[1] < 0:
          self.M_interior[1] = -1*(self.bending_strength*100)
        else:
          self.M_interior[1] = self.bending_strength*100
      self.attached_particles[1].graphical.M_plastic_hinge.append(True)
    else:
      self.M_plasticity[1] = False
      self.M_interior[1] = self.M_interior[1]
      self.attached_particles[1].graphical.M_plastic_hinge.append(False)

def rodResultantAxialForce(self):
  self.axial_force = self.damping_force + self.hooke_force

def rodInternalStresses(self):
  self.internalIndependentStresses()
  self.dependentStresses()

def internalIndependentStresses(self):
  if self.N_plasticity == False:
    self.hooke_force=self.rod_deformation[0* self.E/ self.current_lenght
  else:
    L = self.rod_deformation[0] + self.rest_length
    self.hooke_force = self.rod_deformation[0] * self.EA / L
  self.N_direction()
  self.M_interior = [self.rod_deformation[1]*(40000*self.EI/self.current_lenght)+\
                     self.rod_deformation[2]*(20000*self.EI/self.current_lenght),
                     self.rod_deformation[1]*(20000*self.EI/self.current_lenght)+\
                     self.rod_deformation[2]*(40000*self.EI/self.current_lenght)]
  self.MplasticityCheck()
  self.Acerto_sinalM()
  self.independentStresses = [self.interiorN, self.interiorM]

def N_direction (self):
  if self.delta_length < 0:
    self.N_interior_p1 = [self.angle_p2_to_p1, -1 * self.hooke_force]
    self.N_interior_p2 = [self.angle_p1_to_p2, -1 * self.hooke_force]
  else:
    self.N_interior_p1 = [self.angle_p1_to_p2, self.hooke_force]
    self.N_interior_p2 = [self.angle_p2_to_p1, self.hooke_force]
  self.interiorN = [self.N_interior_p1, self.N_interior_p2]

def Acerto_sinalM (self):
  self.interiorM = [self.M_interior[0]*0.01, self.M_interior[1]*0.01]
  if self.attached_particles[0].physics.original_pos[0]==
       self.attached_particles[1].physics.original_pos[0]:
    if self.attached_particles[0].physics.pos[1]>
       self.attached_particles[1].physics.pos[1]:
      self.interiorM[0] = -1 * self.interiorM[0]
    else:
      self.interiorM[1] = -1 * self.interiorM[1]
  else:
    if self.attached_particles[0].physics.original_pos[0] >
       self.attached_particles[1].physics.original_pos[0]:
      self.interiorM[0] = -1 * self.interiorM[0]
    else:
      self.interiorM[1] = -1 * self.interiorM[1]
```

```
  def dependentStresses (self):
    self.total_moment = self.M_interior[0] + self.M_interior[1]
    V = self.total_moment/self.current_lenght
    if self.attached_particles[0].physics.pos[0] > \
      self.attached_particles[1].physics.pos[0]:
      if self.total_moment >= 0:
        self.V_interior_p2 = [supfc.normalizeAngle(self.rod_angle - math.pi/2),V]
        self.V_interior_p1 = [supfc.normalizeAngle(self.rod_angle + math.pi/2),V]
        self.V_interior_b2 = [supfc.normalizeAngle(self.rod_angle + math.pi/2),-1*V]
        self.V_interior_b1 = [supfc.normalizeAngle(self.rod_angle - math.pi/2),-1*V]
      else:
        self.V_interior_p2 = [supfc.normalizeAngle(self.rod_angle + math.pi/2),-1*V]
        self.V_interior_p1 = [supfc.normalizeAngle(self.rod_angle - math.pi/2),-1*V]
        self.V_interior_b2 = [supfc.normalizeAngle(self.rod_angle - math.pi/2),-1*V]
        self.V_interior_b1 = [supfc.normalizeAngle(self.rod_angle + math.pi/2),-1*V]
    else:
      if self.total_moment >= 0:
        self.V_interior_p1 = [supfc.normalizeAngle(self.rod_angle - math.pi/2),V]
        self.V_interior_p2 = [supfc.normalizeAngle(self.rod_angle + math.pi/2),V]
        self.V_interior_b2 = [supfc.normalizeAngle(self.rod_angle + math.pi/2),-1*V]
        self.V_interior_b1 = [supfc.normalizeAngle(self.rod_angle - math.pi/2),-1*V]
      else:
        self.V_interior_p1 = [supfc.normalizeAngle(self.rod_angle + math.pi/2),-1*V]
        self.V_interior_p2 = [supfc.normalizeAngle(self.rod_angle - math.pi/2),-1*V]
        self.V_interior_b2 = [supfc.normalizeAngle(self.rod_angle + math.pi/2),-1*V]
        self.V_interior_b1 = [supfc.normalizeAngle(self.rod_angle - math.pi/2),-1*V]
    self.Resultant_V_particle = [self.V_interior_p1, self.V_interior_p2]
    self.Resultant_V_rod = [self.V_interior_b1, self.V_interior_b2]

  def rodToParticles (self):
    self.forceToParticles_beam()
    self.momentsToParticles()
```

Função de eventos - *Event Handler*

```
def checkEvents(game_mode, elements = None, fpm_window = None, selected_element = None,
                boxes = None, selected_box = None, allow_letters = False, part = None,
                sub_part = None, estado = None):
  if game_mode == 'inicial_menu':
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        game_mode = 'message'
        return selected_element, game_mode, elements
      elif event.type == pygame.MOUSEBUTTONDOWN:
        mouse_pos = pygame.mouse.get_pos()
        mouse_click = pygame.mouse.get_pressed()
        for b in fpm_window.buttons_menu:
        if b[1]+b[3] > mouse_pos[0] > b[1] and b[2]+b[4] > mouse_pos[1] > b[2]:
          if mouse_click[0] == 1 and b[7] == 'Simular':
            game_mode = 'pause'
            return selected_element, game_mode, elements
          elif mouse_click[0] == 1 and b[7] == 'Propriedades':
            game_mode = 'properties'
            if selected_element == None:
              selected_element = changeSelectedElement(elements, fpm_window)
            return selected_element, game_mode, elements
          elif mouse_click[0] == 1 and b[7] == 'Comandos':
            game_mode = 'controls'
            return selected_element, game_mode, elements
          elif mouse_click[0] == 1 and b[7] == 'Carregar':
            game_mode = 'load'
            return selected_element, game_mode, elements
          elif mouse_click[0] == 1 and b[7] == 'Informar':
            game_mode = 'about'
            return selected_element, game_mode, elements
    return selected_element, game_mode, elements
  if game_mode == 'controls':
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        game_mode = 'message'
        return game_mode
      elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_ESCAPE:
          game_mode = 'inicial_menu'
          return game_mode
```

```
    return game_mode
  if game_mode == 'about':
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        game_mode = 'message'
        return game_mode
      elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_ESCAPE:
          game_mode = 'inicial_menu'
          return game_mode
    return game_mode
  if game_mode == 'message':
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        game_mode = 'inicial_menu'
        return selected_element, game_mode, elements
      elif event.type == pygame.MOUSEBUTTONDOWN:
        mouse_pos = pygame.mouse.get_pos()
        mouse_click = pygame.mouse.get_pressed()
        for b in fpm_window.message_buttons:
          if b[1]+b[3] > mouse_pos[0] > b[1] and b[2]+b[4] > mouse_pos[1] > b[2]:
            if mouse_click[0] == 1 and b[7] == 'Sair':
              quitGame()
            elif mouse_click[0] == 1 and b[7] == 'Guardar':
              game_mode = 'save'
              return selected_element, game_mode, elements
            elif mouse_click[0] == 1 and b[7] == 'Simular':
              game_mode = 'pause'
            elif mouse_click[0] == 1 and b[7] == 'Carregar':
              game_mode = 'load'
              return selected_element, game_mode, elements
    return selected_element, game_mode, elements
  if game_mode == 'simulation':
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        game_mode = 'message'
        return game_mode
      elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_ESCAPE:
          game_mode = 'pause'
          return game_mode
    return game_mode
  if game_mode == 'pause':
    for event in pygame.event.get():
      if event.type == pygame.QUIT:
        game_mode = 'message'
        return selected_element, game_mode, elements
      elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_h:
          selected_element = changeGrid(fpm_window, elements)
        elif event.key == pygame.K_z:
          game_mode = "inicial_menu"
          return selected_element, game_mode, elements
        elif event.key == pygame.K_g:
          addGrid(fpm_window, elements)
        elif event.key == pygame.K_s:
          game_mode = "save"
          return selected_element, game_mode, elements
        elif event.key == pygame.K_d:
          game_mode = "load"
          return selected_element, game_mode, elements
        elif event.key == pygame.K_k:
          if Environment.Environment.Structure_Type == 'truss':
            game_mode = 'force method'
            return selected_element, game_mode, elements
        elif event.key == pygame.K_ESCAPE:
          if selected_element == elements['environment']:
            game_mode = 'simulation'
            return selected_element, game_mode, elements
          elif selected_element == None:
            game_mode = 'simulation'
            return selected_element, game_mode, elements
          else:
            selected_element = changeSelectedElement(elements, fpm_window)
        elif event.key == pygame.K_p:
          addParticle(elements, fpm_window)
        elif event.key == pygame.K_l:
```

```
        addRod(elements, fpm_window)
      elif event.key == pygame.K_i:
        if Environment.Environment.Structure_Type == 'truss':
          switchDiagrams(elements, selected_element)
      elif event.key == pygame.K_f:
        Events.switchForces(elements, selected_element)
      elif event.key == pygame.K_r:
        if Environment.Environment.Structure_Type == 'beam':
          Events.switchReactions(elements, selected_element)
      elif event.key == pygame.K_m:
        if Environment.Environment.Structure_Type == 'beam':
          Events.switchMDiagrams(elements, selected_element)
      elif event.key == pygame.K_v:
        if Environment.Environment.Structure_Type == 'beam':
          Events.switchVDiagrams(elements, selected_element)
      elif event.key == pygame.K_n:
        if Environment.Environment.Structure_Type == 'beam':
          Events.switchNDiagrams(elements, selected_element)
      elif event.key == pygame.K_e:
        if Environment.Environment.excel_export == 1:
          Events.exportExcel(elements, selected_element)
      elif event.key == pygame.K_RETURN:
        game_mode = 'properties'
        if selected_element == None:
          selected_element = changeSelectedElement(elements, fpm_window)
          return selected_element, game_mode, elements
        else:
          return selected_element, game_mode, elements
      elif event.key == pygame.K_DELETE:
        selected_element = deleteElement(elements, selected_element, fpm_window)
    elif event.type == pygame.MOUSEBUTTONDOWN:
      mousepos = pygame.mouse.get_pos()
      selected_element = changeSelectedElement(elements, fpm_window, mousepos)
      mouse_click = pygame.mouse.get_pressed()
      for b in fpm_window.buttons:
        if b[1]+b[3] > mousepos[0] > b[1] and b[2]+b[4] > mousepos[1] > b[2]:
          if mouse_click[0] == 1 and b[7] == 'Ir_Menu':
            game_mode = 'inicial_menu'
            return selected_element, game_mode, elements
          if mouse_click[0] == 1 and b[7] == 'Guardar':
            game_mode = 'save'
            return selected_element, game_mode, elements
  return selected_element, game_mode, elements
if game_mode == 'properties' or game_mode == 'save' or game_mode == 'load':
  input_key = None
  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      game_mode = 'message'
      return game_mode, selected_box, input_key
    elif selected_box == None:
      if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_ESCAPE:
          game_mode = 'pause'
          return game_mode, selected_box, input_key
      elif event.type == pygame.MOUSEBUTTONDOWN:
        mousepos = pygame.mouse.get_pos()
        selected_box = selectBox(boxes, mousepos)
    else:
      if event.type == pygame.KEYDOWN:
        if (event.key == pygame.K_BACKSPACE or event.key == pygame.K_ESCAPE
            or (event.key >= 45 and event.key <= 57)
            or ((event.key >= 97 and event.key <= 122)
            or event.key == 32)):
          if allow_letters == False and selected_box.title != 'a) Structure Type
                                (beam / truss)':
            if event.key <= 57:
              if event.key != 47 and event.key != 32:
                input_key = event.key
              elif event.key == 47:
                input_key = ord('-')
          else:
            if game_mode == 'properties' and selected_box.title == 'a) Structure Type
                            (beam / truss)':
              if event.key == 8 or event.key == 98 or event.key == 116
                  or event.key == 101 or event.key == 114 or event.key == 97
                  or event.key == 117 or event.key == 109 or event.key == 115:
                input_key = event.key
```

```
            elif game_mode == 'save' or game_mode == 'load':
              if event.key != 47:
                input_key = event.key
              else:
                input_key = ord('-')
        if event.key == pygame.K_RETURN:
          selected_box = selectBox(boxes, None)
    return game_mode, selected_box, input_key

  def quitGame():
    pygame.quit()
    sys.exit()

  def exportExcel (elements, selected_element):
    if selected_element.type == 'particle':
      for e in elements['particles']:
        if selected_element == e:
          header = [u'delta t', u'delta pos x', u'delta pos y', u'rotação', u'F
                   exterior', u'F interior', u'E cinetica', u'M exterior', u'M interior',
                   u'E cinetica angular',]
          delta_t = []
          pos_x = []
          pos_y = []
          rotation = []
          F_exterior = []
          F_interior = []
          E_cinetica = []
          M_exterior = []
          M_interior = []
          E_cinetica_angular = []
          for a in e.excel_delta_t:
            delta_t.append(a)
          for b in e.excel_pos_x:
            pos_x.append(b)
          for c in e.excel_pos_y:
            pos_y.append(c)
          for d in e.excel_rot:
            rotation.append(d)
          if e.physics.force_type != 0 or e.physics.accel_type != 0:
            angle = e.excel_F_exterior[0][0]
            for f in e.excel_F_exterior:
              if f[0] != angle:
                F_exterior.append(-1*f[1])
              else:
                F_exterior.append(f[1])
          else:
            for f in e.excel_F_exterior:
              F_exterior.append(f[1])
          for g in e.excel_F_interior:
            F_interior.append(g)
          for h in e.excel_E_cinetica:
            E_cinetica.append(h)
          for i in e.excel_M_exterior:
            M_exterior.append(i)
          for j in e.excel_M_interior:
            M_interior.append(j)
          for k in e.excel_E_cinetica_angular:
            E_cinetica_angular.append(k)
          data = []
          for i in range(0, len(pos_x)):
            data.append([delta_t[i], pos_x[i], pos_y[i], rotation[i], F_exterior[i],
                         F_interior[i],E_cinetica[i], M_exterior[i], M_interior[i],
                         E_cinetica_angular[i]])
          name = 'Particle'
  if selected_element.type == 'rod':
      for e in elements['rods']:
        if selected_element == e:
          header = [u'delta t', u'N', u'V', u'M1', u'M2', u'delta_L', u'rod angle',]
          delta_t = []
          N = []
          V = []
          M1 = []
          M2 = []
          deltaL = []
          rodAngle = []
          for a in e.excel_deltat:
            delta_t.append(a)
```

```
        for b in e.excel_N:
          N.append(b)
        for c in e.excel_V:
          V.append(c)
        for d in e.excel_M1:
          M1.append(d)
        for f in e.excel_M2:
          M2.append(f)
        for g in e.excel_deltaL:
          deltaL.append(g)
        for h in e.excel_tetaR:
          rodAngle.append(h)
        data = []
        for i in range(0, len(delta_t)):
          data.append([delta_t[i], N[i], V[i], M1[i], M2[i], deltaL[i], rodAngle[i]])
        name = 'Rod'
    wb = Workbook()
    ws = wb.active
    t = datetime.datetime
    instant = t.now().strftime("%Y.%m.%d - %Hh%Mm%Ss")
    dest_filename = str(name) + ' Results - ' + str(instant)
    ws.append(header)
    for row in data:
      ws.append(row)
    from openpyxl.chart import ScatterChart, Reference, Series
    chart_pos1 = -10
    chart_pos2 = -10
    chart_pos3 = -10
    for i in range(2,len(header)+1):
      chart = ScatterChart()
      xvalues = Reference(ws, min_col=1, min_row=2, max_row=len(delta_t))
      yvalues = Reference(ws, min_col=i, min_row=1, max_row=len(delta_t))
      series = Series(yvalues, xvalues, title_from_data=True)
      chart.series.append(series)
      if i <= 4:
        chart_pos1 += 15
        ws.add_chart(chart, "L" + str(chart_pos1))
      elif 4 < i <= 7:
        chart_pos2 += 15
        ws.add_chart(chart, "U" + str(chart_pos2))
      else:
        chart_pos3 += 15
        ws.add_chart(chart, "AD" + str(chart_pos3))
    wb.save(dest_filename + '.xlsx')
```

Módulo gráfico – *Rendering Engine*

```
class Graphical_Particle:
  scale = 1
  deformed_scale = 1

  def __init__(self, original_pos, radius, constraint, original_rot, category):
    self.radius = radius
    self.constraint = constraint
    if category == 'second':
      self.original_pos = (original_pos[0] / Environment.Environment.zoom,
                           original_pos[1] / Environment.Environment.zoom)
    else:
      self.original_pos = original_pos
    self.pos = list(self.original_pos)
    self.delta_pos = [0,0]
    self.colour = (0,0,0)
    self.diagram_colour = (0,100,255)
    self.show_diagram = False
    self.show_Forces = True
    self.decompoded_F = 0
    self.decompoded_R = 0
    self.rot = original_rot
    self.show_Reactions = False
    self.hinge = 'no'
    self.M_plastic_hinge = [False]
    self.constraint_colour = (0,0,0)
    if Environment.Environment.Structure_Type == 'truss':
      appliedForce = [{'existence': 'existe', 'angle':[0], 'scalar':[0]},[0,0],[0,0]]
      self.updateDiagram_Truss([0,0], appliedForce)
```

```
    if Environment.Environment.Structure_Type == 'beam':
      appliedForce = [{'existence': 'existe', 'angle':[0], 'scalar':[0]},[0,0],[0,0]]
      appliedMoment = [{'existence': 'existe', 'scalar': 0},0]
      self.updateDiagram_Truss([0,0], appliedForce)
      self.updateDiagram_Beam([0,0], appliedForce, 0, appliedMoment)

  def updateGraphicalParticle(self, delta_pos=None,force=None,rot=None,moment=None,
                              appliedForce = [{'existence': 'existe', 'scalar':[0],
                              'angle':[0]},[0,0],[0,0]],appliedMoment = [{'existence':
                              'existe', 'scalar': 0},0]):
    if Environment.Environment.Structure_Type == 'truss':
      if delta_pos != None:
        self.delta_pos = delta_pos
      self.pos=[self.original_pos[0]+(self.delta_pos[0]/Environment.Environment.zoom)*
                Graphical_Particle.scale,self.original_pos[1]+
                (self.delta_pos[1] / Environment.Environment.zoom)*
                Graphical_Particle.scale]
      self.updateDiagram_Truss(force, appliedForce)
    if Environment.Environment.Structure_Type == 'beam':
      self.plasticHingeColour()
      if delta_pos != None:
        self.delta_pos = delta_pos
      self.pos = [self.original_pos[0]+\
                (self.delta_pos[0] / Environment.Environment.zoom)*\
                Graphical_Particle.scale, self.original_pos[1]+\
                (self.delta_pos[1] / Environment.Environment.zoom)*\
                Graphical_Particle.scale]
      self.updateDiagram_Truss(force, appliedForce)
      if rot != None:
        self.rot = rot * Graphical_Particle.scale
      self.updateDiagram_Beam(force, appliedForce, moment, appliedMoment)

class Graphical_Rod:
  scale = 1
  N_diagramScale = 1
  M_diagramScale = 1
  V_diagramScale = 1

  def __init__ (self, pos1, pos2, particles, name, angle):
    self.particles = particles
    self.pos1 = pos1
    self.pos2 = pos2
    self.colour = (0,0,0)
    self.plasticity = False
    self.diagram_colour = (0,0,0)
    self.show_diagram = False
    self.original_angle = angle
    self.original_pos1 = pos1
    self.original_pos2 = pos2
    self.show_M_diagram = False
    self.show_V_diagram = False
    self.show_N_diagram = False
    self.M_diagram_colour = (0,0,0)
    self.hinge = ['no hinge', pos1, pos2, 0, 0]
    if Environment.Environment.Structure_Type == 'truss':
      self.updateDiagram_Truss(0)
    if Environment.Environment.Structure_Type == 'beam':
      self.updateDiagram_Beam([0,0], [(0,0),(0,0)], 0)

  def updateGraphicalRod(self, new_pos1, new_pos2, rod_plasticity = None, axial_force =
                         None, moment = [], V = []):
    if rod_plasticity != None:
      self.plasticity = rod_plasticity
    self.changecolour()
    self.pos1 = new_pos1
    self.pos2 = new_pos2
    if Environment.Environment.Structure_Type == 'truss':
      self.updateDiagram_Truss(axial_force)
    if Environment.Environment.Structure_Type == 'beam':
      self.hinge[1] = self.particles[0].graphical.pos
      self.hinge[2] = self.particles[1].graphical.pos
      self.updateDiagram_Beam(moment, V, axial_force)
```