Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Simulação de Sistemas Dinâmicos em Python

Rafael das Almas Ascensão

Orientador(es) | Miguel José Barão

Évora 2021

Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Simulação de Sistemas Dinâmicos em Python

Rafael das Almas Ascensão

Orientador(es) | Miguel José Barão

Évora 2021

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

Presidente	|	Paulo Miguel Quaresma (Universidade de Évora)
Vogais	|	Luís Rato (Universidade de Évora) (Arguente) Miguel José Barão (Universidade de Évora) (Orientador)

Évora 2021

Simulação de Sistemas Dinâmicos em Python

Resumo

O estudo e análise de sistemas dinâmicos é comum a várias áreas científicas, sendo a abordagem a este tipo de sistemas facilitada através do uso de simuladores. Apesar de existirem alternativas *open-source* disponíveis, estas apresentam algumas limitações o que por norma leva a que utilizadores optem por soluções *ad-hoc*. Por isso, tem-se verificado um interesse crescente nesta área.

Este trabalho de dissertação de mestrado pretende apresentar o processo de desenvolvimento de um simulador em que a especificação de sistemas dinâmicos e das suas interligações seja facilitada, permitindo o encapsulamento de blocos noutros blocos. Pretende-se também que o simulador possibilite a visualização gráfica dos sistemas e dos valores de *output*. Determinou-se que a implementação do simulador seria feita em Python, pelas suas características e pela popularidade da mesma na comunidade científica.

São ainda incluídas demonstrações do funcionamento do simulador com sistemas que permitam demonstrar de modo básico as suas funcionalidades.

Palavras-chave: Simulador, Sistemas Dinâmicos, Diagrama de Blocos, Controlo, Python.

Simulation of Dynamical Systems in Python

Abstract

Analysis and study of dynamical systems is common in several scientific fields, often aided by simulators used to simplify this task. Even though there are open-sorce options available, these may have limitations that drive users to develop their own *ad-hoc* solutions. Hence the growing interest in this area.

This dissertation work aims to present the development process of a simulator that facilitates the specification of dynamical systems through the specification of block-diagrams and their interconnections, including the ability to have encapsulation. The simulator should also allow graphical vizualization of the systems' blocks and connections, and their output values. It was decided that the simulator would be implemented using Python, due its characteristics and its adoption by the scientific community.

Some examples of the simulator running basic systems are also provided as a way of showing the functionality of the simulator in a simple manner.

Keywords: Simulator, Dynamical System, Block Diagram, Control, Python.

Agradecimentos

Este trabalho de dissertação de mestrado não seria possível sem o apoio de de algumas pessoas, às quais não posso deixar de agradecer.

Ao meu orientador, Professor Doutor Miguel José Simões Barão, pelo apoio ao longo deste processo, pela transmissão dos seus conhecimentos e pela paciência e pronta disponibilidade.

À Sara, por ser minha companheira neste percurso académico e na vida, por toda a motivação, pelas palavras de perseverança, pela presença em todos os momentos, bons e maus, e por ser a pedra basilar onde me apoio, sem o qual não seria possível aqui chegar. *Likewise!*.

Aos meus pais, pelo apoio incondicional, por acreditarem em mim, por todas as palavras de ânimo, e por estarem sempre comigo, mesmo ao meu lado, mesmo quando separados por um oceano de distância.

À minha irmã, pelas palavras de motivação, pela sua companhia digital e por ser cúmplice em todas as aventuras até nas mais descabidas, *Espeeera!*.

Ao Eugéne Suter e à Cindy Silva, pela sua amizade, por me terem acolhido sob o seu tecto e por terem estado presentes no meu percurso académico.

E a todos aqueles que direta ou indiretamente contribuíram para o percurso que me levou à conclusão desta etapa.

Conteúdo

Lista de Figuras

Lista de Códigos

Capítulo 1

Introdução

Desde que há registos que o Homem procura compreender a Natureza e o ambiente que o rodeia. Ao longo da História foram sendo desenvolvidos métodos cada vez mais rigorosos e complexos para esse efeito. Uma das grandes questões, transversal a todas as áreas da ciência, é a definição de teorias que unifiquem conceitos de uma ou várias áreas. Nesse sentido, é crucial a junção de ideias e conhecimento de diferentes áreas. São vários os exemplos de teorias que procuram satisfazer esse propósito, por exemplo, as Leis de Newton, a Teoria da Relatividade de Einstein, etc.

O estudo dos sistemas poderá ser facilitado através de várias abordagens, por exemplo, através da utilização de simuladores. É através de ferramentas como os simuladores que é possível não só compreender um sistema como também calcular o seu comportamento ou modelar os seus elementos de forma a perceber como essas interações afetam o sistema ou até o meio onde este se insere. O desenvolvimento de programas capazes de simular sistemas dinâmicos era, tradicionalmente, uma tarefa direcionada para soluções *ad-hoc* que dificilmente seriam adaptadas para outros problemas além daquele para o qual tinham sido desenhados.

Há alguns anos começaram a ser desenvolvidas ferramentas *open-source* que respondem a este problema de um modo mais geral, no entanto, como se poderá verificar ao longo deste trabalho, continua a verificar-se necessidade de desenvolvimento deste tipo de ferramentas para colmatar lacunas e ultrapassar limitações que continuam a existir.

1.1 Motivação

A ideia para o desenvolvimento deste simulador partiu, assim, da necessidade existente de soluções que permitam simular sistemas dinâmicos baseados no

conceito de diagrama de blocos com *inputs* e *outputs* que possam ser ligados entre os diversos blocos que compõem um sistema.

1.2 Objetivos

O principal objectivo deste trabalho de dissertação de mestrado é desenvolver uma plataforma de simulação de sistemas dinâmicos desenhados com base no conceito de diagrama de blocos, ou seja, pretende-se que esteja facilitada a especificação de sistemas dinâmicos através de blocos e das suas interligações. Para alcançar este objetivo, foram estabelecidas várias características que foram tidas em conta na concepção do simulador, nomeadamente

- Permitir simulação a partir do conceito básico de blocos, o que implica que o utilizador consiga criar blocos, definir *inputs* e *ouputs* para cada bloco e criar ligações entre os blocos.
- Permitir abstração do sistema em estudo, de forma a facilitar o desenho de sistemas complexos, levando a que seja necessário permitir o encapsulamento de blocos (blocos que contêm outros blocos) e a que seja possível determinar dependências entre blocos e os seus *outputs*, de forma a evitar que estas sejam fornecidos pelo utilizador.
- Permitir visualização gráfica das ligações entre blocos.
- Permitir visualização gráfica dos valores de *output*.

1.3 Estrutura

Esta dissertação de mestrado é composta por 6 capítulos, que permitem contextualizar este trabalho não só no âmbito da teoria em que o mesmo se baseia como acompanhar a progressão do desenvolvimento do simulador, desde a delineação do projeto até à implementação de um simulador com as características que foram idealizadas para o mesmo. Está ainda incluída neste trabalho uma reflexão acerca de perspetivas de desenvolvimento futuro.

O primeiro capítulo serve de introdução ao tema, contendo a motivação para o desenvolvimento deste trabalho de dissertação bem como os objetivos que foram definidos para o mesmo.

No capítulo 2, denominado de "Estado da Arte", são abordados os conceitos teóricos fundamentais de Teoria do Controlo que serviram de base à implementação do simulador e é também feita uma análise sumária de vários simuladores semelhantes que existem em Python.

No capítulo 3, "Proposta para Arquitetura do Simulador", é abordado o processo de delineação do simulador, sendo feita uma descrição da abordagem inicial ao problema com a aplicação dos conceitos que estão por detrás do simulador.

No capítulo 4 é apresentada a "Implementação em *Python*" sugerida no capítulo anterior, sendo apresentados o módulo que constitui o simulador e o módulo que fornece blocos pré-definidos ao utilizador com o objetivo de facilitar a utilização do simulador.

Com o capítulo 5, "Demonstração de Utilização", pretende-se demonstrar a utilização do simulador com recurso aos exemplos tradicionais de teoria do controlo.

Por fim, no capítulo 6 serão apresentadas as principais conclusões a que se chegou no decorrer deste projeto e é feita uma reflexão acerca da utilização do simulador e de possibilidades de implementação de funcionalidades que não foram consideradas para este trabalho de dissertação.

Capítulo 2

Estado da Arte

2.1 Revisão sobre Controlo

A teoria do controlo é uma área de investigação interdisciplinar onde múltiplos conceitos e métodos matemáticos se unem para criar uma nova área de matemática, sendo uma área de intersecção entre a engenharia e a matemática. [1, 2]

Esta área disponibiliza ferramentas que permitem o estudo metódico de sistemas, incluindo várias técnicas matemáticas para análise da estabilidade e para determinar bons esquemas de controlo que os optimizem. [3]

No entanto, antes de conseguirmos fazer qualquer tipo de análise, é necessário estabelecer alguns conceitos-chave que permitam expressar as características dos sistemas e das suas interações.

Um dos conceitos fundamentais é o próprio conceito de sistema que, de acordo com [4], é uma combinação de elementos cuja atuação resulta na transformação de sinais. Cada sistema poderá ser composto por um ou mais sinais de entrada e por um ou mais sinais de saída que se relacionam com os primeiros através de relações de transformação impostas pelo sistema.

A definição dos elementos que constituem um determinado sistema depende do problema que está a ser analisado, uma vez que, em certas situações, não é útil explorar todos os detalhes dos elementos internos do sistema, mostrando-se apenas relevante o conhecimento macroscópico do mesmo. Isto é, o modo como os *outputs* de cada sistema variam dado determinados *inputs*.

Uma vez que se caracterize o sistema de uma forma macroscópica, é possível representar elementos, ou conjuntos de elementos, através de um bloco. Esta técnica é conhecida como *caixa preta* e o seu objetivo é facilitar a análise geral do sistema e do modo como as interligações entre entre os elementos afetam o comportamento do sistema.

É com base nesta técnica que é possível alcançar uma das características mais poderosas para a definição de sistemas, o encapsulamento [5]. Esta característica permite construir sistemas compostos por sub-sistemas complexos sem que seja incluída a complexidade desses sub-sistemas, simplificando assim a representação e abordagem aos mesmos.

Esta visão simplificada do sistema é por norma representada graficamente com recurso aos diagramas de blocos.

2.1.1 Diagrama de Blocos

> A linguagem de blocos é uma ferramenta de aplicação universal. Útil não só na investigação e descrição de sistemas conhecidos, mas também na criação e desenvolvimento de novos sistemas. Simulações de diagramas de blocos evidenciam a essência dinâmica das coisas e a criatividade é definida de forma operacional e replicável.
>
> [5, p. xiii]

Os diagramas de blocos são ferramentas fundamentais pois facilitam, devido à sua representação visual, a compreensão de sistemas.

Cada sub-sistema é representado por um bloco que modela a relação entre os sinais de entrada e de saída. A relação entre múltiplos sub-sistemas é representada por ramos (setas) que ligam os *outputs* de um bloco aos *inputs* de outro bloco, como exemplificado na figura 2.1.

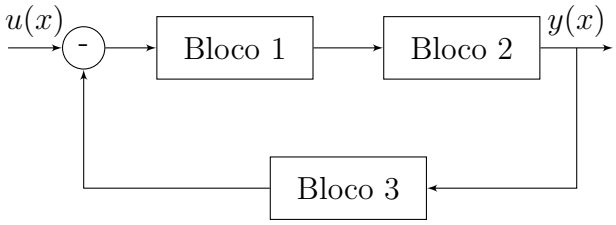


Figura 2.1: Exemplo de um diagrama de blocos

Os sistemas podem ser associados de diferentes formas, por exemplo:

Ligação em série Neste tipo de ligação o *output* de um sistema é conectado de forma sequencial ao *input* do sistema seguinte.

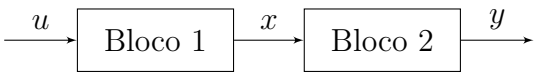


Figura 2.2: Diagrama de blocos em série

Ligação em paralelo Neste tipo de ligação, o mesmo sinal é introduzido em dois ou mais sistemas e o *output* é o resultado do somatório de todos os *outputs*.

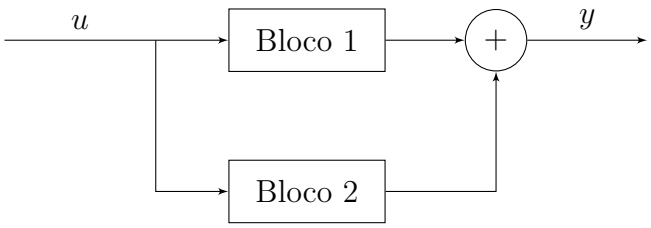


Figura 2.3: Diagrama de blocos em paralelo

Ligação com *feedback* Neste tipo de ligação o *output* do sistema é ligado ao *input* do mesmo.

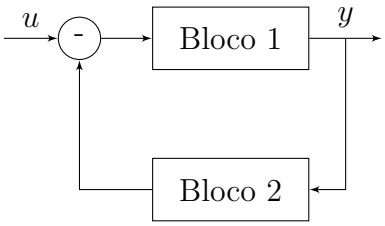


Figura 2.4: Diagrama de blocos com *feedback* negativo

2.1.2 Representação do Espaço de Estados

Outra forma de representar sistemas dinâmicos é pela própria descrição matemática do seu comportamento.

Esta representação pode ser feita descrevendo três aspectos do sistema: as suas variáveis de estado que juntamente com os *inputs* permitem caracterizar completamente o sistema; as equações diferenciais que descrevem como o

estado do sistema é alterado ao longo do tempo; e as equações de *output* que são funções do estado e do *input*, que definem o resultado da transformação do sistema. [4]

Modelo de estado

O modelo de estado é uma representação standard para a modelação de sistemas dinâmicos.

Variaveis de estado Conjunto mínimo de variaveis que permitem descrever o estado de um sistema.

Estado O estado pode ser representado por um vector com todas as variáveis de estado.

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{bmatrix} \tag{2.1}$$

Equação de Estado As equações de estado podem ser representadas por equações diferenciais de primeira ordem e descrevem a evolução do vector de estado ao longo do tempo.

Considerando x, ao longo do tempo t, e *input* u.

$$\begin{aligned} \dot{x}_1 &= f_1(x, u, t) \\ \dot{x}_2 &= f_2(x, u, t) \\ &\cdots \\ \dot{x}_n &= f_n(x, u, t) \end{aligned} \tag{2.2}$$

Que pode ser generalizado por:

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{2.3}$$

Onde A é uma matriz designada por matriz da dinâmica, e B é uma matriz com coeficientes que representam "peso" de cada *input*.

Equação de saída A equação de saída transforma o estado interno do sistema.

$$y(t) = Cx(t) + Du(t) \tag{2.4}$$

Onde C e D são matrizes com coeficientes de peso do estado e *input* no resultado.

2.2 Software existente

A linguagem *Python* é considerada umas das linguagens *standard* para a computação científica devido à existência de um ecossistema rico em soluções dedicadas à investigação científica [6, 7], como é o caso de *NumPy*, *SciPy*, *Matplotlib*, *etc.* que são ferramentas bem estabelecidas na comunidade.

Existem algumas ferramentas que abordam o problema de simulação de sistemas, usando os pacotes acima referidos, no entanto nenhuma delas é considerada um *de facto standard*.

De seguida, serão mencionados alguns dos principais pacotes de *Python* para este efeito.

2.2.1 Pynamical

Pynamical é uma ferramenta que procura facilitar a análise de sistemas discretos não-lineares, através de vários tipos de visualizações, *e.g.* diagramas de bifurcação, diagrama de fases bidimensionais e tridimensionais *etc.* [8]

Esta ferramenta permite a definição de modelos personalizados, contudo, essa tarefa é feita através da definição de uma única função que é posteriormente passada como argumento a uma função de simulação.

É necessário reduzir toda a modelação a uma função que descreve o comportamento global do sistema. Esta tarefa é difícil de concretizar em modelos complexos, sobretudo sem recorrer a estratégias que permitam abstração de subsistemas. Ou seja, A definição desta função é inteiramente da responsabilidade do utilizador, não existindo nenhum apoio à modelação do próprio sistema.

2.2.2 PyDSTool

PyDSTool é um pacote que oferece um conjunto de ferramentas para simulação e análise de sistemas dinâmicos, particularmente adotado pela comunidade das ciências biológicas. [9]

Os autores referem que esta ferramenta se destina a utilizadores avançados que estejam familiarizados com o uso de ferramentas sem interface gráfica ou familiarizados com programação *Python*. [10]

Embora disponibilize um vasto conjunto de ferramentas para análise de sistemas e alguns modelos de sistemas no âmbito das ciências biológicas, esta ferramenta possui uma elevada barreira à entrada nos casos onde é necessário criar modelos hierárquicos personalizados. Apesar de estes modelos poderem ser especificados através da criação de subclasses das classes base já existentes, é necessário conhecimento específico de como essas classes são usadas pelo pacote, o que leva a que esta ferramenta não seja atrativa a utilizadores menos experientes.

2.2.3 SimPy

SimPy é um pacote de simulação para sistemas discretos que usa simulação discreta baseada em processos e eventos. [11]

A situação a simular tem de ser descrita em termos de processos, recursos e eventos. Sendo que os processos são representações dos elementos do sistema, cujo comportamento pode ser definido tendo em conta os recursos que, por sua vez, são propriedades do ambiente de simulação e que podem estar disponíveis a todos os processos. Os processos também podem usar eventos gerados por outros processos ou pelo próprio ambiente (*e.g. timeouts*).

Em teoria, é possível fazer simulações de sistemas contínuos utilizando este pacote, no entanto, uma vez que esse não é foco principal desta ferramenta, não são disponibilizadas ferramentas que facilitem essa abordagem.

Para além disso, a modelação dos sistemas está diretamente associada às próprias definições das funções que definem os processos, eventos e recursos, não havendo assim suporte para tratar componentes do sistema como *caixas pretas*.

2.2.4 ManPy

ManPy é uma ferramenta que tem por base a ferramenta referida anteriormente, o *SimPy*, tendo como foco modelar processos industriais. [12, 13]

A principal vantagem desta solução em relação à anterior é a possibilidade de definir processos e as suas interações de forma abstrata recorrendo ao conceito de *caixas pretas* e ligações entre estas.

A existência desta ferramenta sublinha a importância da disponibilidade de mecanismos que permitam ao utilizador definir processos através do conceito de *caixas pretas*.

2.2.5 python-control

python-control é um pacote que contém ferramentas para a modelação e análise de sistemas de controlo com *feedback*. [14]

É um pacote completo, que disponibiliza várias ferramentas para a modelação e análise de sistemas, que permitem, por exemplo, álgebra de blocos, análise de resposta em frequência, estabilidade, *etc.*

Este pacote também oferece ao utilizador uma interface de compatibilidade que permite o uso de certos comandos disponíveis no *MATLAB*, o que é uma mais valia para utilizadores familiarizados com esse ambiente ou para utilizadores mais experientes em teoria do controlo.

Na versão atual, *0.8.3* (04-01-2020), um novo modulo para a criação de sistemas *input-output* foi adicionado que tenta solucionar os mesmos problemas que originaram os objetivos deste projeto.

2.2.6 SimuPy

SimuPy é, das ferramentas referidas, a que começou a ser desenvolvida mais recentemente (2017), tendo como objetivo a simulação de sistemas usando sistemas dinâmicos interligados. [15]

A principal característica desta ferramenta é a possibilidade de sistemas representados por blocos serem adicionados a um objeto chamado *BlockDiagram*, onde é possível estabelecer ligações entre as entradas e saídas dos seus componentes. No entanto, na versão atual, *1.0.0* (19-09-2017), a ferramenta não suporta o uso de *BlockDiagram* como elementos de outros sistemas. O autor refere que esta característica está planeada para uma uma futura versão.

Este objeto é também responsável pela simulação em si. Contudo para fins de simplificação da simulação, a ferramenta não permite o conceito de *passthrough*, ou seja, não permite que a função de *output* dependa do *input* no mesmo instante. É assumido que na expressão

$$y = Cx + Dy \tag{2.5}$$

a matriz D é nula, o que leva à simplificação da expressão para

$$y = Cx \tag{2.6}$$

de forma a evitar situações que gerem ciclos algébricos na resolução da simulação.

Outra das limitações atuais desta ferramenta é que esta não determina o *flow* dos sinais, ou seja, o utilizador é responsável por adicionar os sistemas numa ordem que satisfaça as interdependências dos blocos.

De todas as ferramentas mencionadas, esta é a ferramenta que mais se aproxima dos objetivos pretendidos, contudo ainda não os atinge completamente devido às limitações referidas.

2.3 Sumário

Tal como referido anteriormente na secção 1.2, um dos principais objetivos para este trabalho de dissertação é desenvolver um simulador que permita a modelação de sistemas a partir do conceito de diagrama de blocos.

Apesar das várias ferramentas acima mencionadas abordarem a simulação de sistemas, com exceção do *SimuPy* e do *python-control*, nenhuma delas faz uma abordagem genérica aos sistemas partindo do conceito de modelar os sistemas a partir de blocos.

Apesar do *SimuPy* estar bastante próximo dos objetivos pretendidos, também este apresenta algumas limitações que não cumprem exatamente os objetivos que pretendíamos alcançar, nomeadamente no que diz respeito à implementação dos conceitos de abstração, *i.e.* permitir criar blocos funcionais a partir da agregação de outros blocos, bem como a evitar a exposição de detalhes de implementação, *e.g.* a especificação da ordem em que os sistemas devem ser iterados para a correta simulação.

Embora o *python-control* na sua versão atual disponibilize ferramentas que vão de encontro aos objetivos que foram estabelecidos para este trabalho de dissertação, essa funcionalidade não estava disponível à data do início do desenvolvimento deste trabalho.

Apesar de existir uma grande diversidade de ferramentas de apoio à computação científica e da maioria dos projetos acima referidos ter um certo nível de maturidade, a resolução deste problema apenas começou a ser abordada no passado recente, pelo que considerou-se pertinente criar um protótipo que explorasse as ideias de abstração e encapsulamento no contexto de simulação de sistemas.

Capítulo 3

Proposta para Arquitetura do Simulador

Usando a representação de espaço de estados podemos representar um bloco, genericamente, pelo conjunto de expressões:

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned} \tag{3.1}$$

De forma a simplificar sistemas compostos por múltiplos sub-sistemas, pode ser criado um sistema unificado concatenando os sub-sistemas num sistema agregado, daqui em diante referido como *SuperBloco*.

Figura 3.1: Sistema composto por dois sub-sistemas

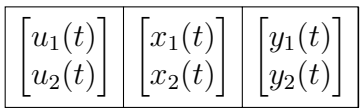


Figura 3.2: Sistema simplificado (*SuperBloco*)

Na figura 3.2 está representado o *SuperBloco* resultante da simplificação do sistema representado na figura 3.1. Dada esta concatenação, é da responsabilidade do *Superbloco* construir a função de atualização do seu estado e a sua função de *output*, concatenando os valores retornados pelas mesmas funções dos sub-sistemas que o compõem. Para isso, é também necessário estabelecer um mapeamento entre cada sub-sistema e a respetiva posição no sistema concatenado, isto é, no *SuperBloco*.

De notar que, apesar desta concatenação lógica, a atualização desta estrutura terá de ser feita em múltiplos passos devido às relações de interdependência entre os sub-sistemas do *SuperBloco*. Levando a que seja também necessário armazenar a ordem em que estas atualizações terão de ser efetuadas de forma a respeitar essas mesmas dependências.

Tomando como exemplo a figura 3.1, verifica-se que o *input* $u_2(t)$ é mapeado ao *output* $y_1(t)$; caso a definição de $y_2(t)$ dependa de $u_2(t)$, isto implica que $y_1(t)$ terá de ser avaliado antes de $y_2(t)$. Blocos com esta propriedade, ou seja, blocos que necessitam dos seus *inputs* para devolver os seus *outputs* no mesmo instante, serão referidos daqui em diante como blocos com *feedforward*.

Esta abordagem, onde vários sistemas são representados pela concatenação de vários sub-sistemas, foi escolhida com o objetivo principal de evitar problemas de continuidade causados pela integração independente de cada bloco.

Tendo em conta as considerações acima referidas, delineou-se um plano para o funcionamento do simulador, abordado neste capítulo, baseado em quatro passos principais: *Definição*, *Pre-processamento*, *Simulação* e *Resultados*.

1. *Definição* do sistema
   (a) Definição dos blocos
      i. Definição das suas variáveis e parâmetros
      ii. Definição da função de derivada do espaço de estado
      iii. Definição da função de *output*
   (b) Definição das suas conexões entre blocos
2. *Pre-processamento* do sistema a ser simulado:
   (a) Determinar a ordem em que os blocos podem ser simulados
   (b) Deteção de ciclos algébricos
   (c) Determinar propriedades de sistemas compostos por agregação de sub-sistemas
   (d) Registo de sinais que pretendemos analisar
3. *Simulação*
   (a) Cálculo e propagação de sinais entre blocos
   (b) Simulação do sistema pela integração das funções de estado

(c) Armazenamento dos sinais requeridos no ponto anterior (2d)

4. Apresentação de *resultados*

   (a) Apresentação dos sinais requeridos

   (b) Representação gráfica do sistema

3.1 Definição do Sistema

A definição do sistema é feita em duas partes. Em primeiro lugar, o utilizador cria instâncias de blocos básicos com determinados parâmetros, por exemplo, no caso em que pretenda criar um sistema que some os valores de duas ondas sinusoidais, deverá ser criado um bloco para cada sinusoide e um bloco soma, como exemplificado no código 3.1.

```
wave1 = Sinusoid('w1')
wave2 = Sinusoid('w2', frequency_hz=.7, amplitude=1)
sum_result = Sum('r', inputs=2)
```

Código 3.1: Definição de dois blocos

Em segundo lugar, o utilizador deverá criar um bloco do tipo *SuperBloco*, usando os blocos básicos referidos anteriormente e definir as ligações entre os mesmos, tal como exemplificado no código 3.2.

```
sys = SuperBlock(subsystems=[wave1, wave2, sum_result])
sys.connect('w1:out0', 'r:in0')
sys.connect('w2:out0', 'r:in1')
```

Código 3.2: Definição das ligações entre blocos

Para possibilitar o conceito de encapsulamento, de forma a permitir uma maior expressividade na modelação dos sistemas, é essencial que o uso do bloco `sys` no código 3.2 seja permitido na criação de novos sistemas do tipo *SuperBloco*.

3.2 Pré-processamento

Tendo em conta que o sistema é definido de forma arbitrária pelo utilizador, que poderá ser composto por múltiplos sub-sistemas aninhados, com ligações entre os sub-sistemas também definidas arbitrariamente, é necessário fazer algum processamento à estrutura criada pelo utilizador de forma a possibilitar a simulação.

A principal propriedade a ser calculada é a ordem em que os sub-sistemas necessitam de ser simulados de forma a satisfazerem as relações de dependência entre os seus elementos. É necessário ter em consideração que, em certos sistemas, alguns dos seus sub-sistemas poderão possuir características de *feedforward*, o que leva a que essas dependências sejam transitivas no mesmo instante t.

É também importante ter em conta que blocos do tipo *SuperBlocos* poderão ser utilizados como constituintes de outros blocos do tipo *SuperBloco*, sendo também necessário determinar se o próprio *SuperBloco* possui características de *feedforward* através da análise das suas ligações internas e da verificação das características de *feedforward* dos seus elementos.

Alem disso, uma vez que o utilizador cria blocos do tipo *SuperBloco* sem definir as funções de atualização do estado e de *output* explicitamente, o valor de retorno dessas funções terá de ser calculado a partir das definições das funções de atualização de estado e de *output* dos seus constituintes. Sendo assim necessário estabelecer um mapeamento entre o próprio *SuperBloco* e os seus constituintes.

3.2.1 *Feedforward*

Algumas ferramentas forçam a que, por definição, os sistemas não possuam a matriz D nas equações (3.1). No entanto, esta limitação provoca alguma diminuição da capacidade de expressar sistemas onde exista de facto esse comportamento.

Quando existe a possibilidade dos blocos terem *outputs* que dependem dos seus próprios *inputs*, as equações (3.1) poderão não ter solução caso existam ciclos algébricos.

Por exemplo, considerando o caso simplificado onde existe um sistema com um único estado, um único *input* e um único *output*, em que o *output* está ligado ao *input*, obtemos:

$$\begin{cases} y(t) = Cx(t) + Du(t) \\ y(t) = u(t) \end{cases} \Leftrightarrow y(t) = Cx(t) + Dy(t) \tag{3.2}$$

o que implica que, para calcular a saída do bloco ($y(t)$), precisemos de conhecer o próprio valor que pretendemos calcular.

O efeito observável da existência da matriz D não nula confere um comportamento onde os sinais de entrada, ou uma transformação destes, sejam reencaminhados como sinais de saída no mesmo instante t. Por isso, a estes blocos, atribuímos a nomenclatura de blocos com *feedforward*.

Em sistemas mais complexos estes ciclos podem não ser imediatos, ou seja, o sinal pode passar por múltiplos blocos antes de voltar ao bloco inicial e, caso todos esses blocos possuam *feedforward*, essa situação mantém-se devido à natureza de propagação de dependência.

A existência de blocos sem *feedforward* num ciclo de *feedback* quebra este ciclo algébrico, uma vez que o *output* desses blocos é dado pela expressão

$$y(t) = Cx(t) \tag{3.3}$$

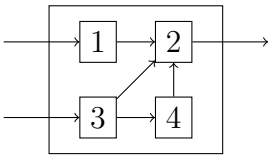
onde o *output* $y(t)$ pode ser imediatamente calculado a partir do estado do bloco sem necessidade de avaliação dos seus *inputs*.

Determinação de *feedforward* no *SuperBloco*

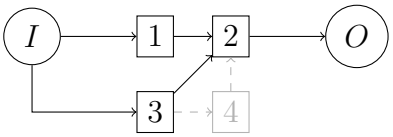
Devido ao facto da definição das funções de *output* ser fornecida pelo utilizador, cabe a este especificar se a função de *output* possui características de *feedforward*[1]. No entanto, analisando a lista de sub-sistemas e as suas interligações, em conjunto com a verificação da propriedade de *feedforward* de cada bloco, podemos inferir se o *SubperBloco* possui *feedforward*.

Isto é, considera-se que um *SuperBloco* possui *feedforward* se existir um caminho entre qualquer das suas entradas até qualquer das suas saídas composto exclusivamente por blocos com *feedforward*.

Podemos então construir um grafo a partir das ligações entre sub-sistemas do *SuperBloco* que possuam *feedforward*, ligando todos os *inputs* a um nó I, e ligando todos os *outputs* a outro nó O. Caso se verifique um caminho que ligue o nó de *input* do *SuperBloco* ao nó de *output*, considera-se que o *SuperBloco* tem *feedforward*.



(a) *Superbloco* com dois *inputs* e um *output*; Blocos 1, 2 e 3 considerados blocos com *feedforward*.



(b) Grafo considerado para cálculo de *feedforward*; Bloco 4 não é considerado por não ter *feedforward*.

Figura 3.3: Cálculo de *feedforward* no *SuperBloco*

[1]Esta limitação poderá ser minimizada no futuro através de suporte à definição simbólica de funções. Ver Seção 6.1 Limitações e Trabalho Futuro.

3.2.2 Ordem de Simulação

Considerando as expressões (3.1), verificamos que cada sistema obriga uma avaliação ordenada devido às dependências internas das expressões que as regem, como exemplificado na figura 3.4.

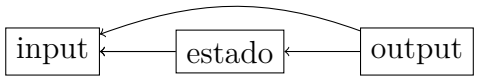


Figura 3.4: Dependência de avaliação de cada bloco

Podemos então estabelecer uma ordem de avaliação dos elementos que pretendemos calcular, considerando a:

- Avaliação dos *inputs* pela chamada das funções de *output*, uma vez que os valores de *inputs* são os resultados das funções de *output* a que o bloco está ligado.
- Avaliação da derivada do estado, cujo resultado apenas afeta o instante seguinte.

A avaliação dos *inputs* divide-se em dois casos principais, pois depende do facto do sistema que fornece o *input* ter ou não *feedforward*. Em blocos sem *feedforward*, por definição, o *output* não depende do *input*, assim sendo, a avaliação do *output* desses blocos pode ser feita de forma imediata. Por outro lado, a avaliação do *output* de um sistema com *feedforward* implica a avaliação dos sistemas que o precedem. Sendo assim necessário avaliá-los numa ordem que satisfaça essas relações de dependência.

Para isso, cria-se um grafo direcionado onde nós são sub-sistemas do bloco que possuem *feedforward* e onde arestas representam as suas interligações como dependências. Desta forma, a ordem pela qual os blocos com *feedforward* necessitam de ser avaliados é obtida pela ordem topológica desse mesmo grafo.

Assim, estabelece-se como ordem de simulação dos componentes de um sistema todos os elementos sem *feedforward* seguidos pela ordenação topológica de elementos com *feedforward*. De notar que os sistemas sem *feedforward* necessitam de atualizar os seus *inputs* depois desta avaliação, pois esses *inputs* podem ser dependentes de qualquer dos subsistemas já avaliados, caso existam ciclos de *feedback*.

Exemplo

Considere-se o sistema demonstrado na figura 3.5, onde os blocos B_1, B_2 e B_3 são blocos sem *feedforward* e os blocos F_1 e F_2 são blocos com *feedforward*.

Tendo em conta que a avaliação é definida pela lista de blocos sem *feedforward* seguida da ordenação topológica dos blocos com *feedforward*, uma ordem de avaliação válida para o caso mencionado na figura 3.5 é $[B_1, B_2, B_3, F_1, F_2]$.

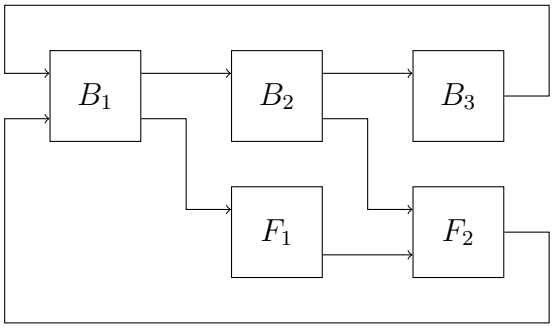


Figura 3.5: Sistema genérico

Os *inputs* e *outputs* dos blocos são então avaliados pela ordem definida na lista. Como todos os blocos sem *feedforward* estão no início da lista, estes são avaliados em primeiro lugar, conforme ilustrado na figura 3.6.

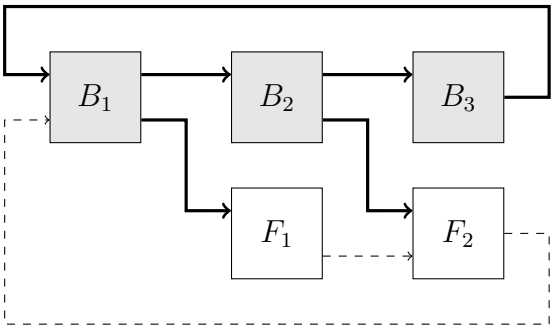


Figura 3.6: Avaliação dos blocos sem *feedforward*

De notar que estes blocos ainda poderão possuir *inputs* inválidos, como se pode verificar pelo *input* ainda a tracejado do bloco B_1 na figura 3.6.

Em seguida, são avaliados os blocos com *feedforward*, demonstrado na figura 3.7. Como estes blocos foram adicionados à lista segundo a sua ordem topológica, a avaliação é efetuada nessa ordem, respeitando assim as relações de inter-dependência.

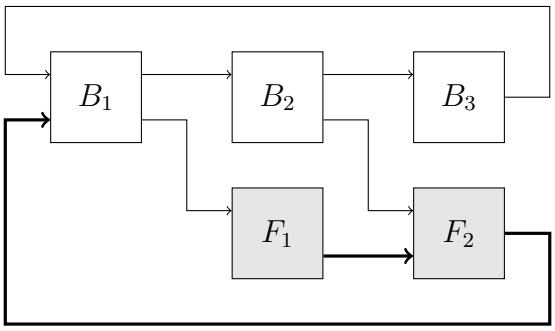


Figura 3.7: Avaliação dos blocos com *feedforward*

Por fim, é necessário iterar novamente sobre os blocos sem *feedforward* de forma a atualizar blocos que potencialmente possuam *inputs* ainda inválidos. Neste exemplo particular, esta atualização é necessária para a correta atualização do bloco B_1, contudo, os blocos B_2 e B_3 também são avaliados.

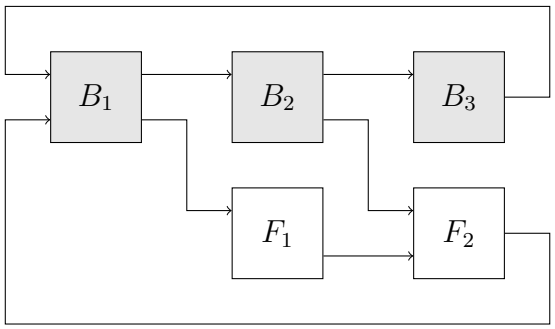


Figura 3.8: Finalização da avaliação

Uma vez concluídos estes passos, termina-se assim a avaliação de *inputs* e *outputs*.

3.3 Simulação

Com a informação obtida através dos processos anteriores, o sistema fica caracterizado de forma a que a simulação possa acontecer.

Seguindo a ordem definida anteriormente, a simulação é computada num ciclo de acordo com 3 passos.

O primeiro passo, propagação de *inputs* e *outputs*, consiste no processamento desses valores como descrito em 3.2.2.

No segundo passo, o estado do sistema é atualizado pela integração da função de derivada de estado.

E por fim, o terceiro passo consiste em armazenar em memória os valores dos sinais requeridos pelo utilizador, uma vez que quando o utilizador pretende simular um sistema, por norma, não está apenas interessado no resultado final, pretendendo na maior parte dos casos analisar valores de todos ou apenas alguns sinais de interesse, ao longo do tempo, e com determinada frequência. Apesar da especificação dos sinais ser feita como parte do preprocessamento, a obtenção dos seus valores é feita pela interrupção periódica ou introspeção dos métodos de integração. Tendo em conta que o processo para atingir este fim é dependente do método utilizado para a integração, um caso particular será discutido no capítulo seguinte.

De notar que devido à natureza de encapsulamento, onde os blocos poderão ser constituídos por outros blocos, será necessário simular cada um dos seus constituintes de igual forma, detalhe esse que será abordado com mais pormenor no capítulo seguinte.

3.4 Apresentação de Resultados

Considerou-se relevante a possibilidade de representar graficamente o sistema em si, de forma a facilitar a visualização dos seus elementos e conexões, que poderá ser apresentada em qualquer momento.

Depois da simulação do sistema, o utilizador poderá pedir que a informação calculada seja apresentada de forma gráfica, ou de forma numérica caso pretenda criar os gráficos noutro ambiente ou utilizar os resultados para outros fins.

O próximo capítulo abordará a criação de um pacote em *Python* com a implementação destas ideias.

Capítulo 4

Implementação em *Python*

A implementação deste pactote, daqui em diante também referido como *dynamicalpy*[1], foi realizada por dois módulos: `model` e `blocks`.

O módulo `model` é responsável pela funcionalidade fundamental ao funcionamento do simulador, enquanto o módulo `blocks` é responsável pela implementação de blocos pré-definidos, prontos a ser utilizados pelo utilizador.

4.1 Módulo `model.py`

Este módulo é composto por duas classes, `BaseBlock` e `SuperBlock`, que implementam as peças básicas do simulador, onde `BaseBlock` define as propriedades básicas necessárias para a representação de um bloco e o `SuperBlock` faz a gestão da topologia dos elementos e da simulação propriamente dita.

4.1.1 Classe `BaseBlock`

A classe `BaseBlock` não é destinada ao uso direto pelo utilizador, mas serve como base para a definição de outras classes através do conceito de herança, indicando quais os métodos que devem ser implementados pelos *blocos* e implementando algumas funcionalidades básicas comuns a todos os *blocos*.

O `BaseBlock` possui a seguinte assinatura:

```
BaseBlock(name=<Name>, inputs=<inputs>, outputs=<outputs>)
```

Código 4.1: Assinatura do `BaseBlock`

Onde `<Name>` é uma string que dá o nome ao *bloco*, `<inputs>` e `<outputs>` são uma especificação de nomes para portas, que podem tomar as seguintes formas:

[1] Disponível em `https://github.com/rafasc/dynamicalpy/releases/tag/0.0.0`

Numérico inteiro Especifica N portas, seguindo um padrão que por definição é `<prefixo><número>`, onde o `<prefixo>` toma valores de `in` e `out` para portas de *input* e *output* respetivamente.

E.g. `inputs=1 outputs=2` resultaria num bloco com portas de *input* `in0` e portas de *output* `out0` e `out1`.

String String com os nomes de cada porta separados por vírgulas.

E.g. `inputs="velocidade,posicao"` define dois *inputs* com os nomes `velocidade` e `posicao`.

Iterável de strings Lista ou tuplo de strings a serem usadas como nomes de cada porta.

E.g. `outputs=["velocidade", "posicao", "rotacao"]` define três *outputs* com os nomes `velocidade`, `posicao` e `rotacao`.

Cada bloco também terá de implementar os seguintes métodos:

`def state_deriv(self, t, x, u):`
: Função deve retornar um vector com a derivada de estado. Sendo t um escalar com o valor do instante no tempo, e x e u os vectores de estado e *input*, respetivamente.

`def output_eq(self, t, x, u):`
: Função deve retornar um vector com os valores de *output* do bloco. Sendo t um escalar com o valor do instante no tempo, e x e u os vectores de estado e *input*, respetivamente.

`def set_initial_sate(self, state):`
: Função deve inicializar o vector `self.initial_state` com os valores do estado inicial, que será utilizado como estado inicial do bloco.

Cada `BaseBlock` também possui as seguintes propriedades:

`self.order`
: Número de variáveis que o vector de estado possui.

`self.input_names, self.output_names`
: São listas que armazenam os nomes de cada porta de *input/output*.

  Cada *input/output* é identificado pela posição numérica na lista correspondente.

4.1.2 Classe SuperBlock

A classe SuperBlock é responsável por criar *blocos* que sejam compostos pelo agrupamento de outros *blocos* e criar o ambiente necessário para que esses blocos possam ser simulados, sendo importante que a classe SuperBlock seja também uma *sub-classe* de BaseBlock permitindo assim a criação de objetos SuperBlock com base em outros objetos da classe BaseBlock ou SuperBlock.

O SuperBlock possui a seguinte assinatura:

```
SuperBlock(name=<Name>,
           inputs=<inputs>,
           outputs=<outputs>,
           subsystems=[lista de subsistemas])
```

Código 4.2: Assinatura do SuperBlock

onde subsystems é uma lista de blocos que compõem o *SuperBloco*.

O utilizador define e interage com as instâncias de SuperBlock através dos seguintes métodos:

`SuperBlock.connect(self, outport, inport):`

Conecta o *output* de um bloco membro do SuperBlock ao *input* de outro membro do SuperBlock. através da utilização de uma especificação do tipo '<nomedobloco>:<nomedaporta>'.

`SuperBlock.run(self, time, start=0, resolution=100, record=None):`

Corre a simulação durante um período de tempo especificado por time, tendo início pelo valor definido por start. record é uma especificação que consiste numa lista de sinais dos quais se pretende guardar valores no decorrer da simulação, sendo resolution o numero de amostras por segundo.

`SuperBlock.plot(self, plot=None):`

Apresenta o gráfico dos valores dos sinais especificados por plot.

Apenas sinais que tenham tido uma entrada em record no método run(), referido anteriormente, estão disponíveis.

`SuperBlock.values(self, record=None):`

Semelhante a plot(), no entanto retorna um array com os valores especificados por record.

Apenas sinais que tenham tido uma entrada em record no método run(), referido anteriormente, estão disponíveis.

`SuperBlock.draw_diagram(self):`
: Mostra uma visualização dos elementos do `SuperBlock` e das suas interligações.

`SuperBlock.print_links(self, raw=False):`
: Imprime uma tabela que mostra as ligações entre elementos internos ao `SuperBlock`. No caso em que `raw=True` mostra também os índices que representam cada elemento no *super bloco*.

Funções e parâmetros internos

Estas funções fazem parte integrante do `SuperBloco`, e são essenciais ao funcionamento e execução da simulação, no entanto não são destinadas a uso direto pelo utilizador.

`SuperBlock.links`
: Um dicionario onde estão representadas as ligações entre os componentes do `SuperBlock`.

  Contém não só a informação das ligações internas entre membros do `SuperBlock`, como o mapeamento entre *inputs* e *outputs* do próprio `SuperBlock` aos *inputs* e *outputs* dos seus elementos.

`SuperBlock.update_internal_structures(self):`
: Função responsável pelo pré-processamento referido no capítulo anterior.

`SuperBlock.slices`
: Uma lista de *slices* que faz o mapeamento do estado de cada componente do `SuperBlock` para o estado do `SuperBlock` em si. Este mapeamento é efetuado pela função anterior.

`SuperBlock._recalculate_inputs_outputs(self, t, x, u)`
: Função que calcula os *inputs*/*outputs* de cada bloco.

`SuperBlock.uu, SuperBlock.yy`
: Vectores para armazenamento de valores transientes no processo do cálculo de *inputs* e *output* do *super bloco*. Os valores destes vectores são calculados pela função anterior.

`SuperBlock.state_deriv(self, t, x , u)`
: Função que retorna a derivada de estado, pela chamada das funções de derivada de cada elemento do `SuperBlock`.

Funcionamento

Para simular a evolução do estado do sistema ao longo do tempo faremos uso do pacote `scipy`, mais precisamente do método `scipy.integrate.ode`, para resolução de equações diferencias do tipo $y'(t) = f(t, y)$.

O método `scipy.integrate.ode` tem a seguinte assinatura:

```
scipy.integrate.ode(f, jac=None)
```

Código 4.3: Assinatura da função `scipy.integrate.ode`

onde `f` é um *callable* com assinatura `f(t, y, *f_args)`.

Este método é usado na execução da simulação seguindo o pseudo-código 4.4.

```
1  # pré-processamento
2  self._update_internal_structures()
3
4  # inicialização
5  r = scipy.integrate.ode(self.state_deriv)
6  r.set_initial_value(self.initial_state, start)
7  t1 = time
8  dt = 1/resolution
9
10 # execução
11 while r.successful() and r.t < time:
12     values = r.integrate(r.t+dt)
13     # armazenamento dos valores de output
14     # requeridos pelo utilizador em 'record='
```

Pseudo-código 4.4: `SuperBlock.run()`

Como primeiro passo, a função `_update_internal_structures()` é chamada, fazendo o pré-processamento discutido no capítulo anterior. Neste pré-processamento a lista `self.subsystems` é re-ordenada de forma a respeitar as dependências, tendo em conta o cálculo de *feedforward*, e é determinado o mapeamento entre o estado do `SuperBlock` e o estado de cada um dos seus elementos através de uma lista de *slices*, nomeadamente `SuperBlock.slices`.

A lista `SuperBlock.slices` contém o mesmo número de elementos que a lista `Superblock.subsystems` e, para cada elemento desta última, um objeto *slice* correspondente existe em `SuperBlock.slices`, representando o número de variáveis de estado de cada elemento e a sua posição no estado do `SuperBlock`. Blocos sem variáveis de estado produzem uma *slice* "vazia", ou seja, uma *slice* cujo valor de `slice.start` e `slice.stop` é o mesmo (i.e. uma *slice* do tipo `slice(n,n,None)`).

Uma vez que a chamada desta função poderá causar uma reorganização da lista `Superblock.subsystems`, é também necessário atualizar os índices no mapeamento de blocos, representado por `SuperBlock.links`, uma vez que as

ligações nesta lista são armazenadas sob a forma de chave-valor $k \rightarrow v$,

$$(B_i, m) \rightarrow (B_j, n) \tag{4.1}$$

onde B_i e B_j são índices dos blocos na lista `SuperBlock.subsystems`, m refere-se ao *input* número-m do bloco referido por B_i e n ao *output* número-n do bloco referido por B_j.

No caso particular em que B_i ou B_j tomam o valor de `None`, m e n representam *outputs* e *inputs* do próprio `SuperBlock`, respetivamente. Note-se que neste caso existe uma aparente troca entre *output* e *input*, isto deve-se ao facto das portas de *output* do `SuperBlock` se comportarem como portas de *input* que recebem o valor de um dos *outputs* internos e o reencaminham como *output* do próprio `SuperBlock`. O mesmo acontece em relação ao *input* do `SuperBlock`, onde as suas portas de *input* funcionam como portas de *output* que fornecem os valores de *input* do `SuperBlock` aos *inputs* dos seus blocos internos.

Uma vez estabelecidas as relações de *input-output*, esta informação é utilizada pela função `SuperBlock._recalculate_inputs_outputs()`, para o cálculo de *inputs* e *outputs* no `SuperBlock`. Este cálculo ocorre em dois passos, em primeiro lugar são calculados todos os *inputs* e *outputs* de todos os blocos, obedecendo à ordem definida em `SuperBlock.subsystems`, sendo os resultados temporariamente armazenados em `SuperBlock.uu` e `SuperBlock.yy`. No entanto, isto leva a que os blocos sem *feedforward* tenham os seus *inputs* calculados com dados potencialmente desatualizados. Por isso, como segundo passo, é feita outra iteração sobre os elementos, agora apenas aos elementos sem *feedforward*, calculando, desta vez, apenas os seus *inputs* com informação atualizada decorrente do passo anterior.

A função `SuperBlock._recalculate_inputs_outputs()` é então chamada como primeiro passo da função `SuperBlock.state_deriv()` (ver código 4.5), que é definida *"recursivamente"*.

```
def state_deriv(self, t, x, u):
  self._recalculate_inputs_outputs(t, x, u)
  for i, s in enumerate(self.slices):
    deriv[s] = self.subsystems[i]
               .state_deriv(t, x[s], self.uu[i])
  return deriv
```

Código 4.5: Função derivada de estado de um `SuperBlock`

Neste caso o termo *recursivamente* é utilizado com um sentido lato, dado que a função em si não é verdadeiramente recursiva, mas sim capaz de fazer a travessia sistemática de todos os componentes do sistema, pela chamada do método com o mesmo nome de cada um dos seus constituintes, fazendo com

que SuperBlock.state_deriv() possa ser utilizada como argumento do método scipy.integrate.ode, simulando a totalidade do sistema de forma unificada como se se tratasse de um único sistema. É através desta técnica que se consegue o conceito de *encapsulamento*.

Uma vez calculados, os valores dos *outputs* requeridos pelo utilizador através da especificação designada em record= são armazenados para posterior utilização em SuperBlock.plot() para a visualização gráfica, ou devolvidos por SuperBlock.values(). Este processo é representado no pseudo-código 4.4 pelos comentários das linhas 13 e 14.

4.2 Módulo `blocks.py`

Para facilitar a utilização deste pacote, o módulo `blocks` fornece a definição de alguns blocos básicos, podendo o utilizador apenas importar os blocos que pretende utilizar, ou importar todos os blocos pré-definidos da seguinte forma:

```
# import do bloco Sinusoid
from dynamicalpy.blocks import Sinusoid
# import de todos os blocos pré-definidos
from dynamicalpy.blocks import *
```

Código 4.6: Importação de blocos pré-definidos

Nesta seção serão apresentados os blocos pre-definidos pelo pacote e uma breve descrição dos parâmetros adicionais em relação ao parâmetros definidos pelo `BaseBlock`.

ContinuousTimeLinearSystem

É um bloco para a definição de um bloco tendo como base as matrizes A, B, C e D, como referido na Subseção 2.1.2 Representação do Espaço de Estados, ou através da sua função de transferência.

A sua assinatura é

```
ContinuousTimeLinearSystem(name=None,
                           state_space=None,
                           transfer_function=None,
                           inputs=None, outputs=None,
                           initial_state=None)
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

state_space um tuplo com as 4 matrizes, (A, B, C, D), que definem o sistema. As matrizes C e D, caso omitidas, assumem por omissão a matriz identidade e a matriz nula, respectivamente.

initial_state um vector com os valores do estado inicial.

transfer_function tuplo na forma (N, D), onde N e D são os coeficientes do numerador e denominador da função de transferência, respectivamente.

ContinuousTimeSystem

Este bloco é semelhante ao anterior, no entanto, em vez de usar matrizes usa funções especificadas pelo utilizador.

A sua assinatura é

```
ContinuousTimeSystem(name=None,
                     state_eq=None,
                     order=0,
                     inputs=0,
                     outputs=0,
                     output_eq=(lambda t, x, u: x),
                     initial_state=0,
                     feedforward=False))
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

state_eq função da derivada de estado.

output_eq equação de *output* do bloco.

initial_state estado inicial.

feedforward boolean que especifica se o bloco apresenta características de *feedforward*.

Sawtooth

Este bloco produz uma "onda serra".

A sua assinatura é

```
class Sawtooth(name=None,
               low=0,
               high=1,
               frequency_hz=1,
               phase=0)
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

low o valor mínimo do sinal.

high o valor máximo do sinal.

frequency_hz frequência do sinal em Hertz.

phase fase do sinal.

Triangle

Este bloco produz uma onda triangular.

A sua assinatura é

```
Triangle(name=None,
         frequency_hz=1,
         amplitude=1,
         offset=0,
         phase=0)
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

frequency_hz frequência do sinal em Hertz

amplitude amplitude do sinal

offset desfasamento do sinal no eixo dos yy.

phase fase do sinal.

Square

Este bloco produz uma onda quadrada.

A sua assinatura é

```
class Square(name=None,
             frequency_hz=1,
             low=0,
             high=1,
             dutycycle=0.5,
             phase=0)
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

frequency_hz frequência do sinal em Hertz.

low valor mínimo do sinal.

high valor máximo do sinal.

duty_cycle fração do período em que o sinal mantém o seu valor máximo.

phase fase do sinal.

Sinusoid

Este bloco produz uma onda sinusoidal.

A sua assinatura é

```
class Sinusoid(name=None,
               frequency_hz=1,
               amplitude=1,
               offset=0,
               phase=0)
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

frequency_hz define a frequência do sinal do bloco em Hertz

amplitude amplitude do sinal do bloco

offset desfasamento do sinal no eixo dos yy

phase fase do sinal do bloco

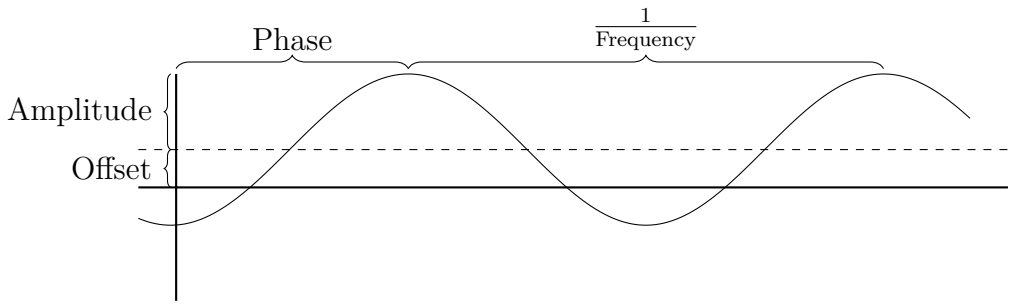


Figura 4.1: Propriedades do bloco *Sinusoid*

Constant

Este bloco produz um sinal constante, útil para especificação de valores de referência.

A sua assinatura é

```
class Constant(name=None, value=1)
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

value valor do sinal constante.

Step

Este bloco produz um sinal "degrau".

```
class Step(name=None,
           step_time=1,
           initial_value=0,
           final_value=1)
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

initial_value valor inicial.

final_value valor final.

step_time instante em que o sinal altera o seu valor de `initial_value` para `final_value`.

Sum

Este bloco produz o somatório de todos os sinais de entrada.

A sua assinatura é

```
class Sum(name=None,
          signs='++')
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

signs especificação que consiste numa sequencia de caracteres `+` e `-`.

O número de entradas é inferido pelo número de caracteres desta especificação e o valor do somatório é obtido aplicando os sinais especificados aos sinais de entrada.

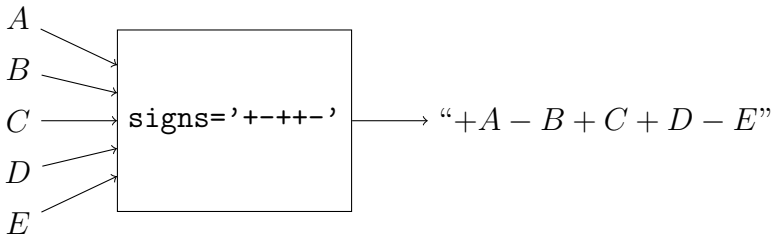


Figura 4.2: Exemplo bloco `Sum`

SumConst

Este bloco é derivado do bloco anterior (`Sum`) e permite a especificação de uma constante a ser adicionada ao sinal de saída.

A sua assinatura é

```
class SumConst(name=None,
               signs='+',
               value=0)
```

com os seguintes parâmetros adicionais em relação ao `Sum`:

value valor da constante a adicionar ao sinal de saída.

Product

Este bloco multiplica todos os valores dos sinais de entrada.

A sua assinatura é

```
class Product(name=None,
              inputs=2)
```

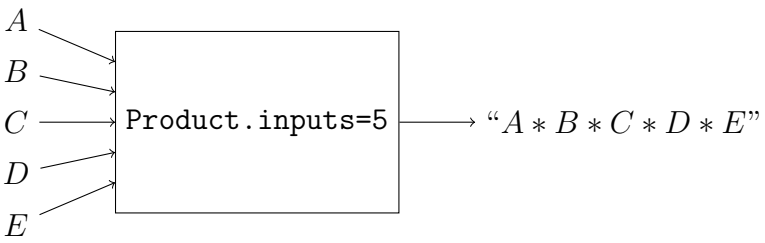


Figura 4.3: Exemplo bloco `Product`

Function

Este bloco define um sistema sem estado, cuja equação de *output* é definida pelo utilizador.

A sua assinatura é

```
class Function(name=None
               function,
               inputs=1)
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

function função definida pelo utilizador que deverá receber tantos argumentos como existem inputs, e retornar um único valor.

Saturation

Este bloco impõe valores máximo e mínimos que um sinal pode tomar.

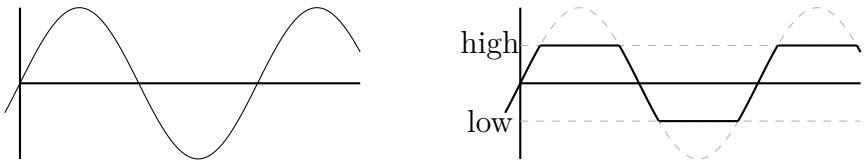
A sua assinatura é

```
class Saturation(name=None,
                 inputs=1,
                 outputs=1,
                 low=-1,
                 high=1)
```

com os seguintes parâmetros adicionais em relação ao `BaseBlock`:

low valor mínimo do sinal.

high valor máximo do sinal.



(a) Sinal de entrada (b) Sinal de saída

Figura 4.4: Exemplo bloco `Saturation`

Capítulo 5

Demonstração de Utilização

Neste capítulo apresenta-se o uso da ferramenta desenvolvida, mostrando alguns exemplos com objetivo de familiarizar o utilizador ao uso do pacote.

Assim, este capítulo está dividido em duas partes, na primeira demonstram-se as funcionalidades da ferramenta com recurso a exemplos mais básicos, enquanto que na segunda parte se procura mostrar exemplos do uso da ferramenta através da implementação de sistemas considerados clássicos.

Em todos os casos, o fluxo de trabalho segue aproximadamente a seguinte sequência:

- Configuração do sistema
  - Definir blocos elementares (`BaseBlock()` ou através dos blocos oferecidos por `dynamicalpy.blocks`)
  - Agrupar os blocos em super blocos (`SuperBlock()`)
  - Definir as ligações dos elementos do super bloco (`connect()`)
- Inspeção do sistema
  - `print_links()`
  - `draw_diagram()`
- Execução
  - `run()`
- Análise dos resultados
  - `plot()`
  - `values()`

5.1 Exemplos Básicos

5.1.1 Soma de Dois Sinais

Neste exemplo mostra-se a utilização da ferramenta num caso onde o *output* pretendido é a soma de dois blocos.

```
from dynamicalpy.model import *
from dynamicalpy.blocks import *

#1 definição dos blocos
wave1 = Sinusoid('w1')
wave2 = Sinusoid('w2', frequency_hz=.7, amplitude=1)
wave3 = Sum('w3', '++')

#2 aggregação dos blocos em SuperBloco
signal = SuperBlock('signal', outputs=1,
                    subsystems=[wave1, wave2, wave3])

#3 ligações entre blocos
signal.connect('w1:out0', 'w3:in0')
signal.connect('w2:out0', 'w3:in1')
signal.connect('w3:out0', ':out0')

#4 visualização do sistema em estudo
signal.draw_diagram()
signal.print_links()

#5 execução da simulação por 10 segundos
signal.run(10, record=['w3:out0', 'w1:out0',
                       'w2:out0', ':out0'])

#6 visualização gráfica dos valores
signal.plot('w1:out0')
signal.plot(['w2:out0'])
signal.plot([':out0'])
signal.plot(['w3:out0', 'w1:out0', 'w2:out0'])

#7 requisição dos valores
print(signal.values(record=['w3:out0', 'w2:out0']))
```

Código 5.1: Exemplo soma de dois sinais

Considerando o código 5.1, após o `import` dos módulos `model` e `blocks`, são definidos todos os blocos que constituem o sistema em estudo, como se pode verificar após o comentário `#1`.

De seguida, é criado o *super bloco* que agrupa todos os elementos anteriores, conforme se vê após o comentário `#2`.

O passo seguinte é definir as ligações entre os elementos do *Super Bloco* indicados pelo comentário `#3`.

O utilizador pode, opcionalmente, inspecionar o sistema criado (comentário `#4`). A ferramenta dispõe de dois métodos para esse efeito, um baseado numa descrição textual das ligações do `SuperBlock`, como se pode verificar no código 5.2, e outro proporciona uma representação gráfica da mesma informação, apresentado pela figura 5.1. De notar que nesta figura o próprio `SuperBlock` aparece representado como um bloco. Isto é feito de forma a demonstrar o mapeamento entre *inputs* e *outputs* do próprio `SuperBlock` aos *inputs* e *outputs* dos seus elementos.

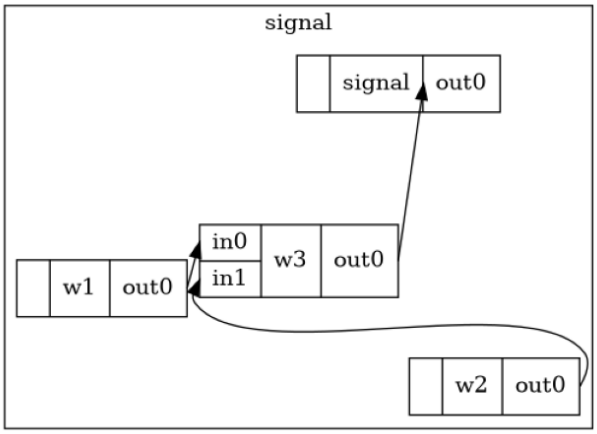


Figura 5.1: *Output* de `draw_diagram()`

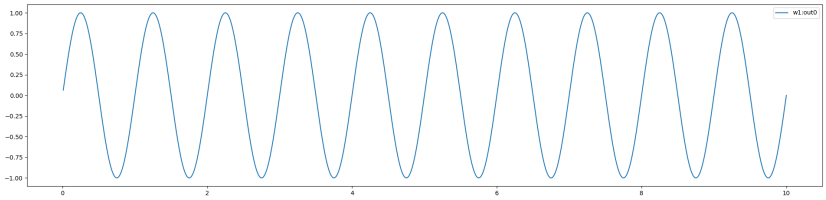
```
---- links of superblock "signal" ----
      w1:out0      ->        w3:in0
      w2:out0      ->        w3:in1
      w3:out0      ->    signal:out0
--------------------------------------
```

Código 5.2: Output de `print_links()`

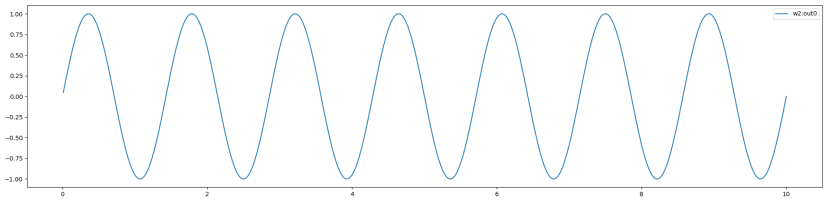
O próximo passo consiste na chamada da função `run()`, na qual o utilizador especifica os parâmetros de simulação, onde neste caso foi especificada uma duração de simulação de 10s. O utilizador define também os sinais a armazenar que serão utilizados nos passos seguintes. Passo ilustrado pelo comentário `#5`.

Finalmente, após a conclusão da simulação, o utilizador visualiza de forma gráfica os valores armazenados no ponto anterior através da chamada da função `SuperBlock.plot()` (ver comentário `#6`), cujos resultados se podem verificar na figura 5.2.

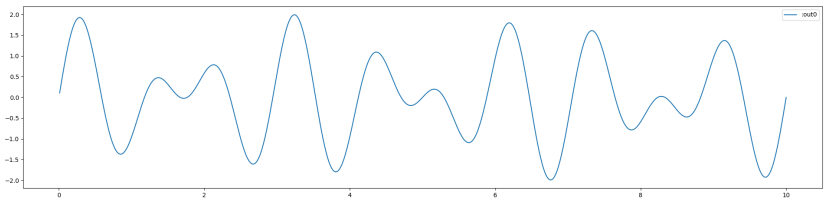
Neste caso são também requeridos os valores de forma explicita, usando a função `SuperBlock.values()`, exemplificado no comentário `#7`, cuja representação é impressa na consola, demonstrado pelo código 5.3.



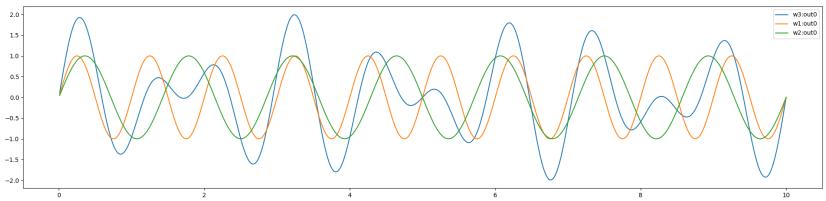
(a) Sinusóide 1



(b) Sinusóide 2



(c) Soma das Sinusóides



(d) Valores anteriores no mesmo gráfico

Figura 5.2: Gráficos de `signal.plot()`

```
[[None None]
[0.10675863784717828 0.0439681183178649]
[0.21318443011504742 0.08785119655074317]
...
[-0.21318443011683896 -0.08785119655148262]
[-0.10675863784898132 -0.043968118318605684]
[-1.8018369205901545e-12 -7.406789507093105e-13]]
```

Código 5.3: Output de `values()`

5.1.2 Blocos Encapsulados

Neste exemplo demonstra-se o uso do conceito de *encapsulamento*, com o objetivo de simplificação de um sistema.

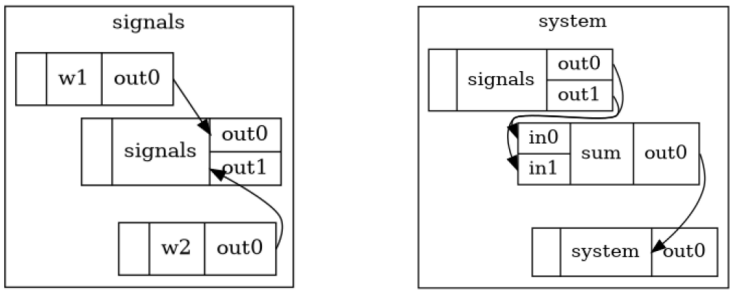
O exemplo utilizado é semelhante ao exemplo anterior, onde duas ondas são somadas, no entanto, os blocos que produzem os sinais são agrupados num único bloco.

```
from dynamicalpy.model import *
from dynamicalpy.blocks import *

#1 definição dos blocos
wave1 = Sinusoid('w1')
wave2 = Sinusoid('w2', frequency_hz=.7, amplitude=1)

#2 configuração do superbloco dos sinasi e ligações
signals = SuperBlock('signals', outputs=2,
                     subsystems=[wave1, wave2])
signals.connect('w1:out0', ':out0')
signals.connect('w2:out0', ':out1')

#3 configuração do superbloco para simulação
result = Sum('sum', '++')
system = SuperBlock('system', outputs=1,
                    subsystems=[signals,result])
system.connect('signals:out0', 'sum:in0')
system.connect('signals:out1', 'sum:in1')
system.connect('sum:out0', 'system:out0')

# #4 visualização do sistema em estudo
system.draw_diagram()
signals.draw_diagram()

#5 execução da simulação por 10 segundos
system.run(10, record=[':out0',
                       'signals:out0',
                       'signals:out1'])

#6 visualização gráfica dos valores
system.plot('signals:out0')
system.plot(['signals:out1'])
system.plot([':out0'])
system.plot([':out0', 'signals:out0', 'signals:out1'])
```

Código 5.4: Exemplo soma de dois sinais com *encapsulamento*

Como se pode verificar, o código 5.4 é semelhante ao apresentado no exemplo 5.1, sendo as principais diferenças a configuração dos super blocos de forma encapsulada, e a chamada da função `draw_diagram()` para todos os

blocos do tipo `SuperBlock`, cujo resultado se pode consultar na figura 5.3.

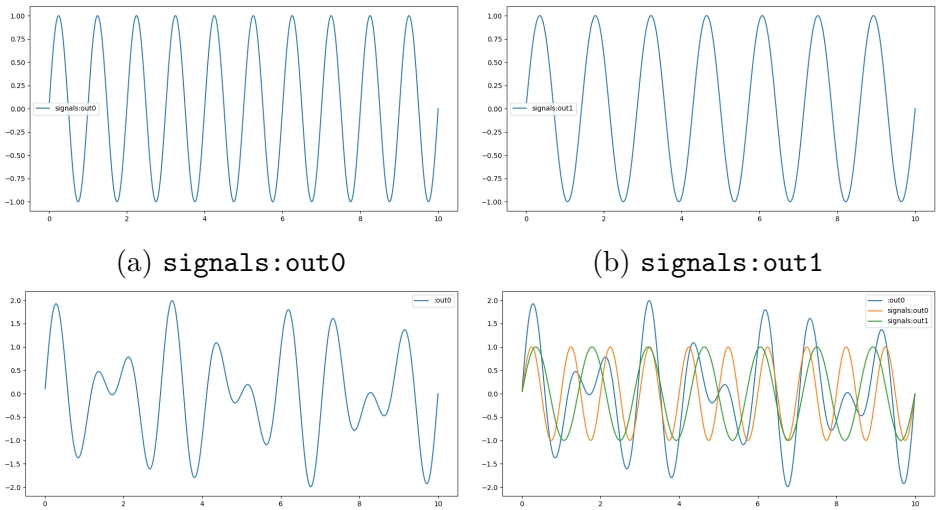


(a) Diagrama do bloco `signals` (b) Diagrama do bloco `system`

Figura 5.3: *Output* das chamadas a `draw_diagram()`

Na figura 5.3b é apresentado o sistema simplificado, enquanto que na figura 5.3a é apresentado o conteúdo do *Super Bloco* `signals`, membro do *Super Bloco* `system`.

Também podemos verificar que existe uma equivalência de resultados entre as figuras 5.4 e 5.2.



(a) `signals:out0` (b) `signals:out1`

(c) `system:out0` (d) Valores anteriores no mesmo gráfico

Figura 5.4: Gráficos de `system.plot()`

5.2 Exemplos Clássicos

5.2.1 Massa Suspensa

Considere-se o cenário onde um objeto de massa m é suspenso por uma mola como representando pela figura 5.5.

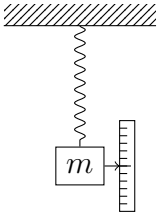


Figura 5.5: Massa suspensa por mola

Pela lei de Hooke, a força necessária para mover uma mola da sua posição de equilíbrio é proporcional ao deslocamento, $F = kx$. E de acordo com a segunda lei de Newton temos que $F = ma$.

Considerando que a mola exerce a força no sentido contrário ao movimento, relativamente à posição de equilíbrio, por substituição temos:

$$\begin{aligned} -kx &= m\frac{d^2x}{dt^2} \\ \frac{d^2x}{dt^2} &= \frac{-k}{m}x \end{aligned} \tag{5.1}$$

Assim, para caracterizar o sistema podemos usar um vector de estado com [posição (x), velocidade ($\dot{x}$)] e, pela aplicação da expressão de espaço de estados,

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{5.2}$$

obtemos

$$\begin{aligned} \dot{x}(t) &= Ax(t) \\ \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ \frac{-k}{m} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \end{aligned} \tag{5.3}$$

uma vez que, neste exemplo em particular, não são considerados *inputs* externos ao sistema, resultando na omissão da parte $Bu(t)$.

A implementação deste sistema é demonstrada pelo código 5.5, com parâmetros $k = 0.5$, $m = 1$ e estado inicial de uma unidade abaixo do ponto de equilíbrio e velocidade zero.

```
from dynamicalpy.model import SuperBlock
from dynamicalpy.blocks import ContinuousTimeLinearSystem
import numpy as np

k, m = 0.5, 1

A = np.array([[0,    1],
              [-k/m, 0]])
B = np.array([[]])

spring = ContinuousTimeLinearSystem(name='spring',
                                    state_space=(A,B))
spring.set_initial_state([1,0])

system = SuperBlock(name='system', subsystems=[spring])

record=['spring:out0', 'spring:out1']
system.run(30, record=record)
system.plot(record)
```

Código 5.5: Massa suspensa por mola

A figura 5.6 mostra o resultado desta simulação onde se pode verificar o movimento harmónico do sistema. Também se pode verificar a relação entre a posição (`spring:out0`) e a velocidade (`spring:out1`), verificando-se que quando uma atinge o seu valor máximo a outra é zero.

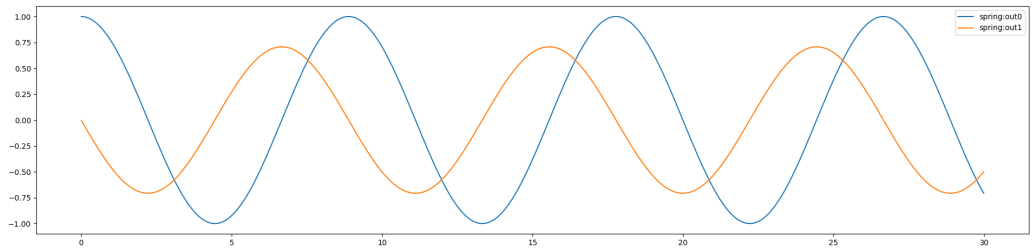


Figura 5.6: Relação entre posição e velocidade da massa suspensa

5.2.2 Massa Suspensa Sujeita a Força Externa

Este próximo exemplo será uma continuação do exemplo anterior, no entanto, considera-se a existência de *inputs* externos ao sistema e uma componente de amortecimento.

Assim sendo, e assumindo que a força de amortecimento é proporcional e contrária ao movimento, o sistema é descrito pelas seguintes expressões (5.4),

$$\begin{aligned} -kx - \beta\dot{x} + F_{\text{externa}} &= m\ddot{x} \\ -kx - \beta\frac{dx}{dt} + F_{\text{externa}} &= m\frac{d^2x}{dt^2} \\ \frac{d^2x}{dt^2} &= -\frac{kx}{m} - \frac{\beta\frac{dx}{dt}}{m} + \frac{F_{\text{externa}}}{m} \end{aligned} \tag{5.4}$$

resultando no modelo descrito por (5.5),

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{-k}{m} & \frac{-\beta}{m} \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} F_{\text{externa}} \tag{5.5}$$

que quando transcrito para o uso neste pacote é ilustrado pelo código 5.6.

```
from dynamicalpy.model import SuperBlock
from dynamicalpy.blocks import Square
from dynamicalpy.blocks import ContinuousTimeLinearSystem
import numpy as np

m, k, β = 1, 0.5, 0.10

A = np.array([[0,1],
              [-k/m, -β/m]])
B = np.array([[0],[1/m]])

spring = ContinuousTimeLinearSystem(name='spring',
                                    state_space=(A,B))
spring.set_initial_state([0.5, 0])

f_externa = Square(name='force', frequency_hz=3/120,
                   dutycycle=0.05, phase=-5, high=1.5)

system = SuperBlock(name='system',
                    outputs=('force', 'pos'),
                    subsystems=[spring,f_externa])

system.connect('force:out0', 'spring:in0')
system.connect('spring:out0', 'system:pos')
system.connect('force:out0', 'system:force')
system.draw_diagram()
```

```
record=[':pos',':force']
system.run(120, record=record)
system.plot(plot=record)
```

Código 5.6: Massa suspensa por mola com *input* externo

A figura 5.7 mostra o diagrama interno do sistema e na figura 5.8 mostra-se o resultado desta simulação.

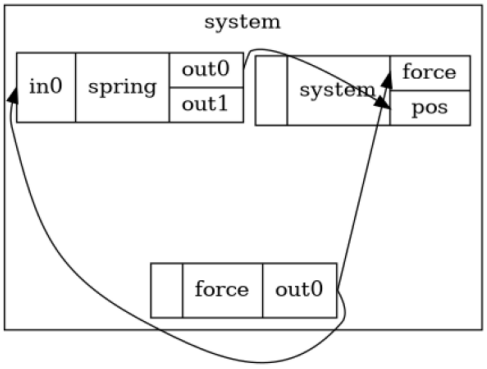


Figura 5.7: Diagrama de massa suspensa com *input* externo

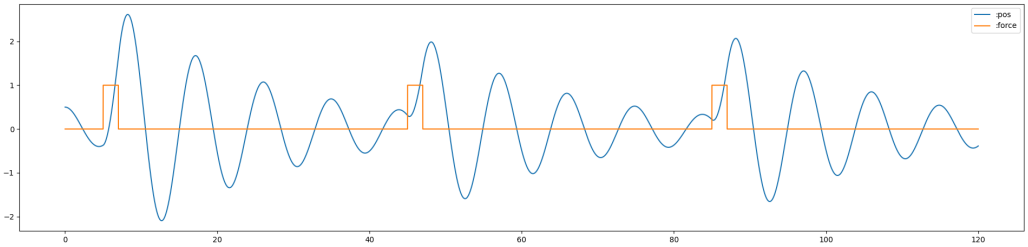


Figura 5.8: Resultado da simulação do código 5.6

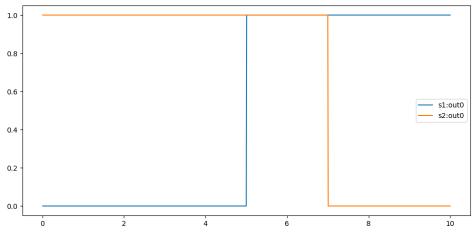
Criação de Impulso com Recurso a Blocos Existentes

No exemplo anterior utilizou-se uma onda retangular como *input* externo do sistema. No entanto, a natureza periódica deste afeta o sistema de forma cíclica, impossibilitando a análise dos efeitos a longo prazo de apenas um "pulso".

Nesta situação existem duas vias de solução. O utilizador pode criar um bloco personalizado fazendo uma subclasse de `BaseBlock`, definindo as expressões que regem o seu comportamento, porém, e em certos casos, é consideravelmente mais prático criar sinais pela composição de sinais existentes.

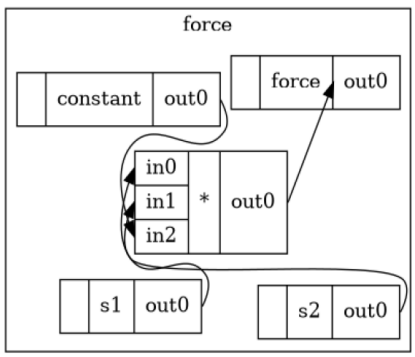
Assim, com este exemplo, pretende-se demonstrar a flexibilidade que o pacote proporciona ao suportar o conceito de *encapsulamento*.

Definição do *input* No caso particular da criação de um impulso retangular, o sinal pode ser construído a partir do produto de dois blocos "degrau", neste caso já fornecidos por `dynamicalpy.model`.

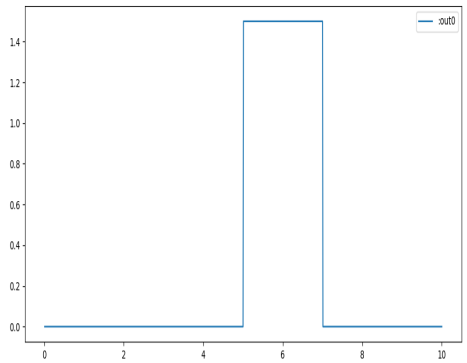


De forma a controlar a amplitude deste sinal, é adicionada uma porta de *input* adicional, ligada a um bloco constante, que funciona como um ganho.

O resultado de `draw_diagram()` é apresentado pela figura 5.9a e o resultado da sua simulação é mostrado na figura 5.9b.



(a) Diagrama do bloco personalizado



(b) Output do bloco personalizado

```
from dynamicalpy.model import SuperBlock
from dynamicalpy.blocks import Step, Constant, Product
from dynamicalpy.blocks import ContinuousTimeLinearSystem
import numpy as np

amplitude, start, end = 1.5, 5, 7
step1 = Step('s1', start, 0, 1)
step2 = Step('s2', end, 1, 0)
c = Constant('constant', value=amplitude)
p = Product('*', inputs=3)
f_externa = SuperBlock('force', outputs=1,
                    subsystems=[step1,step2,p,c])

f_externa.connect('s1:out0', '*:in0')
f_externa.connect('s2:out0', '*:in1')
f_externa.connect('constant:out0', '*:in2')
f_externa.connect('*:out0', ':out0')

f_externa.draw_diagram()
record=[':out0','s1:out0', 's2:out0']
f_externa.run(10, record=record)
f_externa.plot(['s1:out0','s2:out0'])
f_externa.plot(':out0')

# Simulação conforme o exemplo anterior
m, k, β = 1, 0.5, 0.10
A = np.array([[0,1],
              [-k/m, -β/m]])
B = np.array([[0],[1/m]])
spring = ContinuousTimeLinearSystem(name='spring',
                                    state_space=(A,B))
spring.set_initial_state([0.5, 0])

# Força externa definida anteriormente
system = SuperBlock(name='system',
                    outputs=('force', 'pos'),
                    subsystems=[spring,f_externa])

system.connect('force:out0', 'spring:in0')
system.connect('spring:out0', 'system:pos')
system.connect('force:out0', 'system:force')
system.draw_diagram()

record=[':pos',':force']
system.run(120, record=record)
system.plot(plot=record)
```

Código 5.7: Massa suspensa por mola com *input* externo

O código 5.7 mostra a completa implementação do sistema, desde a criação do *input* personalizado à aplicação deste impulso no mesmo sistema do exemplo anterior.

A figura 5.10 apresenta o resultado da aplicação deste *input* personalizado ao sistema, onde se pode verificar quer a oscilação da mola, quer o seu amortecimento ao longo do tempo.

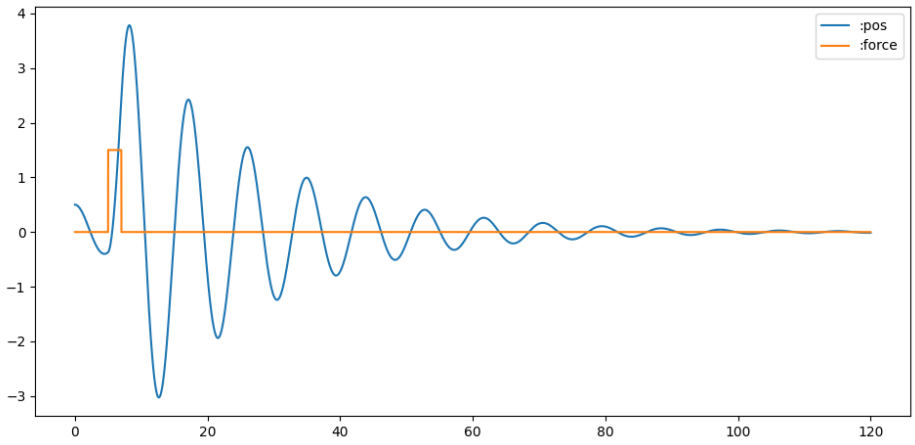


Figura 5.10: Resultado da aplicação do *input* personalizado

Como podemos verificar, a criação de sinais pela composição de elementos existentes é relativamente simples e rápida. O facto de ser possibilitada a simulação de *super blocos*, facilita a definição dos mesmos, uma vez que eles próprios podem ser simulados de forma independente do sistema em que estão inseridos.

Caso esses blocos se mostrem úteis a longo prazo, o utilizador poderá criar um módulo de blocos, análogo a `dynamical.blocks`, onde poderá definir os seus próprios blocos. Blocos que provem ser úteis o suficiente para uso genérico poderão ser adicionados aos blocos providenciados por defeito, enriquecendo assim o pacote.

5.2.3 Modelo para doenças infectocontagiosas (*SIRD*)

O nome deste modelo resulta das quatro variáveis de estado que são utilizadas na modelo de propagação de uma doença infectocontagiosa.

Onde S é o número de pessoas suscetíveis a doença, I o número de pessoas infectadas com a doença, R o número de pessoas que recuperaram da doença e D o número de pessoas que faleceram como consequência da doença.

Cada individuo pode evoluir neste modelo de acordo com a máquina de estados representada na figura 5.11 e cujo comportamento é descrito pelo conjunto de equações diferenciais em (5.6).

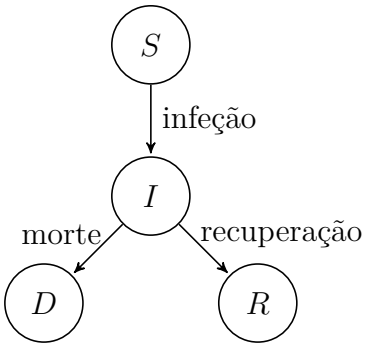


Figura 5.11: Máquina de estados do modelo SIRD

$$\begin{aligned} \frac{dS}{dt} &= -\frac{\beta IS}{N} \\ \frac{dI}{dt} &= \frac{\beta IS}{N} - \gamma I - \mu I \\ \frac{dR}{dt} &= \gamma I \\ \frac{dD}{dt} &= \mu I \end{aligned} \tag{5.6}$$

Este modelo também necessita da especificação de alguns parâmetros que caracterizam a doença em análise e a população por ela afetada. Nomeadamente, N representa o número total de indivíduos, e β, γ, e μ são as taxas de infeção, recuperação e mortalidade, respectivamente.

Para além da implementação do modelo *SIRD*, que consiste essencialmente na transcrição do conjunto de equações diferenciais (5.6) para código *Python* (ver linhas 13-19 do código 5.8), tarefa que é consideravelmente facilitada pelo facto da linguagem permitir o uso de caracteres especiais, pretende-se também demonstrar o uso de portas com nomes personalizados. (ver linhas 21, 22 e 26).

A figura 5.12 mostra os resultados da execução do código 5.8, onde se pode analisar a evolução de cada um dos grupos de pessoas que constituem o modelo.

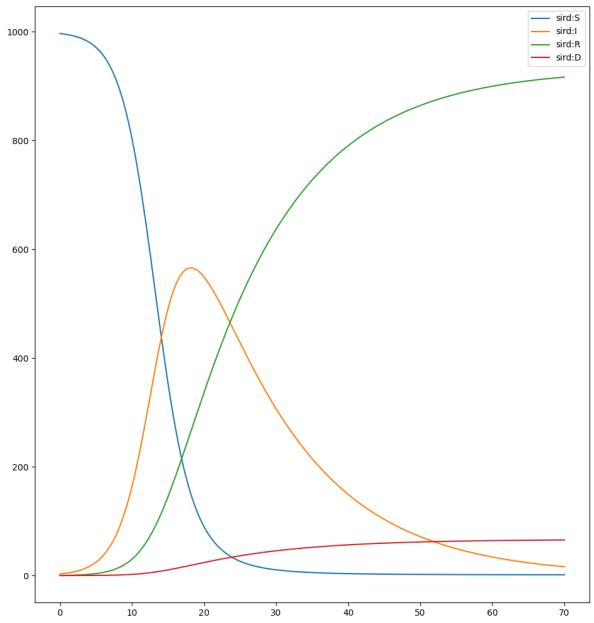


Figura 5.12: Resultados da simulação do modelo SIRD

O exemplo dado no código 5.8 pretende também evidenciar a possibilidade de criar ligações de forma programática, embora neste caso particular, ilustrado pela figura 5.13, estas ligações não tenham grande utilidade prática devido ao facto do *super bloco* ter acesso direto a todos os *outputs* dos blocos que o compõem.

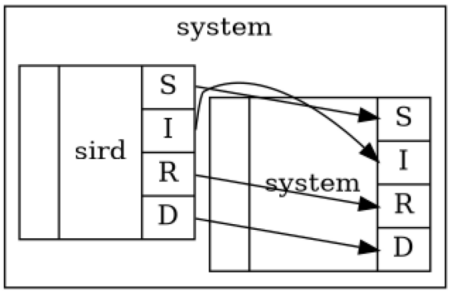


Figura 5.13: Diagrama do modelo SIRD

```
from dynamicalpy.model import SuperBlock
from dynamicalpy.blocks import ContinuousTimeSystem
import numpy as np

β = 0.5
γ = 0.07
μ = 0.005
initial_state = [997, 3, 0, 0]
N = sum(initial_state)

print(f"Modelo␣SIRD␣com␣R0␣de␣{β␣/␣γ}")

def dsird(t,x,u):
    S, I, R, D = x
    dS = - (β * I * S) / N
    dI = (β * I * S) / N - I * γ - I * μ
    dR = γ * I
    dD = μ * I
    return np.array([dS, dI, dR, dD])

ports=('S','I','R','D')
sird = ContinuousTimeSystem(name='sird', outputs=ports,
                            state_eq=dsird, order=4, )
sird.set_initial_state(initial_state)

system = SuperBlock(name='system', outputs=ports,
                    subsystems=[sird])

record=[f"sird:{p}" for p in ports]
for port in ports:
    system.connect(f"sird:{port}", f"system:{port}")

system.run(70, record=record)
system.plot(record)
system.draw_diagram()
```

Código 5.8: Implementação do modelo SIRD

5.2.4 Controlo de Pêndulo Invertido

Considere-se um pêndulo montado num carro que se movimenta apenas num eixo, como ilustrado na figura 5.14.

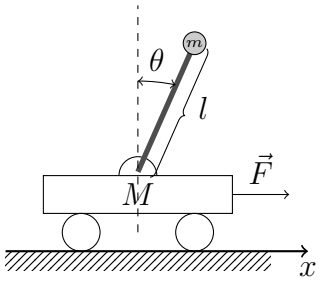


Figura 5.14: Pêndulo invertido num carro

Um possível modelo para este cenário é o proposto por [16], descrito pelo vetor de estado $[x, \dot{x}, \theta, \dot{\theta}]$ e pelas equações de estado (5.7).

$$\begin{aligned}
\dot{x}_1 &= \dot{x} = x_2 \\
\dot{x}_2 &= \ddot{x} = \frac{-mg \sin x_3 \cos x_3 + mlx_4^2 \sin x_3 + f_\theta m x_4 \cos x_3 + F}{M + (1 - \cos^2 x_3)m} \\
\dot{x}_3 &= \dot{\theta} = x_4 \\
\dot{x}_4 &= \frac{(M+m)(g \sin x_3 - f_\theta x_4) - (lmx_4^2 \sin x_3 + F) \cos x_3}{l(M + (1 - \cos^2 x_3)m)}
\end{aligned} \tag{5.7}$$

Como objetivo estabeleceu-se alcançar o ponto de equilíbrio na origem, para isso foram criados três controladores proporcionais, de forma a corrigir os erros no ângulo do pêndulo e na posição e velocidade do carro.

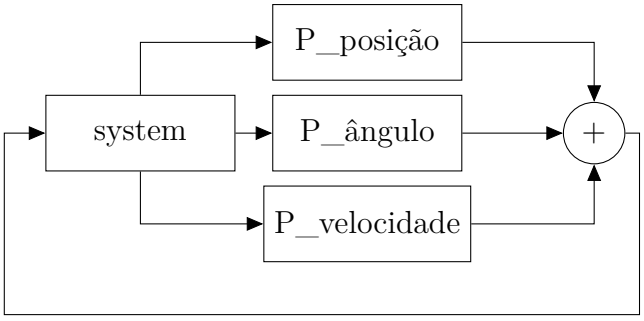


Figura 5.15: Controlo do pêndulo invertido

Seguindo o esquema de controlo ilustrado pela figura 5.15, simulou-se o sistema pela implementação do código 5.9.

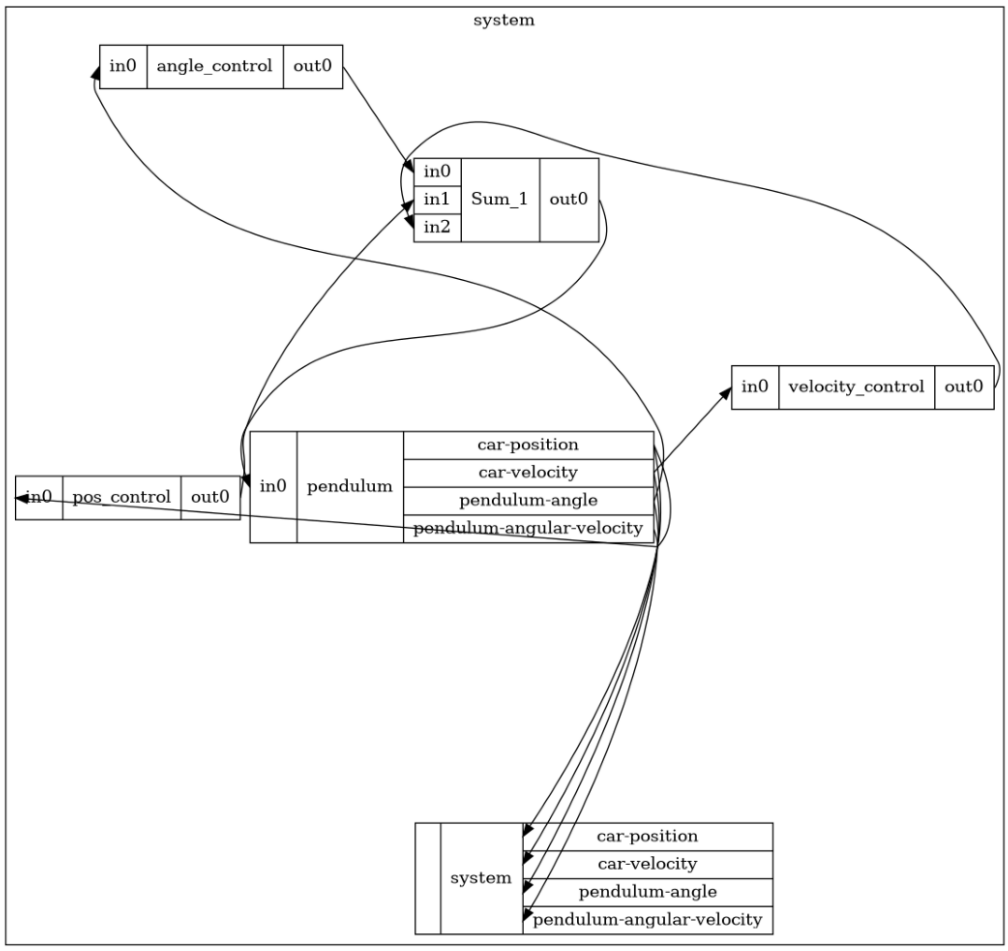
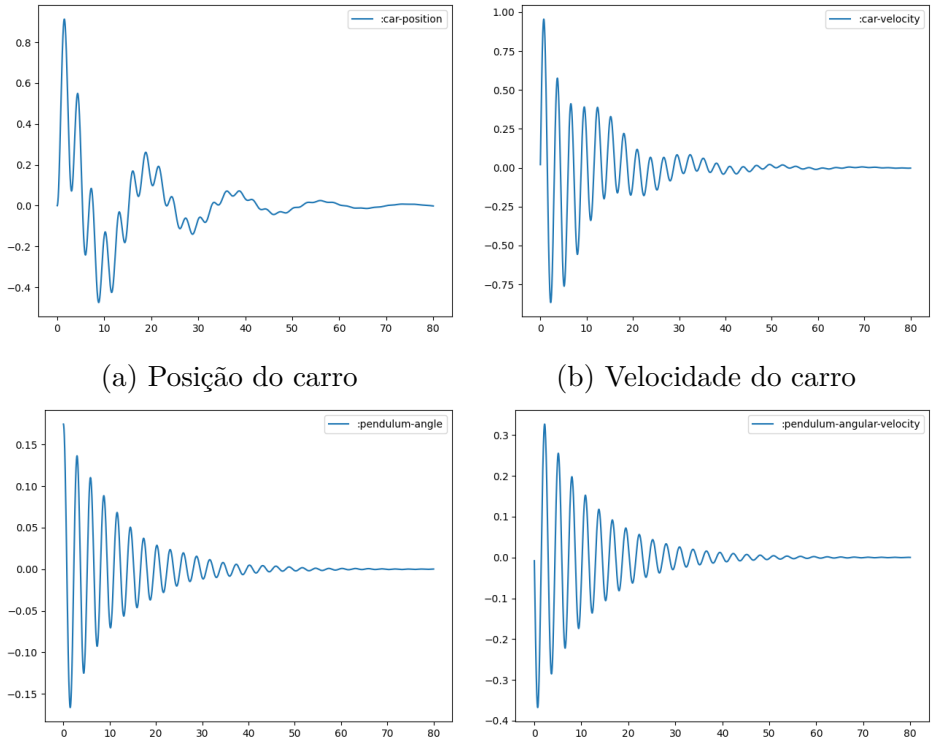


Figura 5.16: `draw_diagram()` do pêndulo invertido

A figura 5.16 mostra o diagrama do sistema, equivalente ao esquema da figura 5.15, contudo mostra também os limites do uso do *graphviz* para a visualização desta informação.



(a) Posição do carro

(b) Velocidade do carro

(c) Ângulo do pêndulo

(d) Velocidade angular do pêndulo

Figura 5.17: Controlo de pêndulo invertido com controlador proporcional

Nos gráficos da figura 5.17, verificamos que o sistema estabiliza ao fim de algum tempo, apesar de ter sido usado um esquema simples de controlo, consistindo em apenas três controladores proporcionais.

```
from dynamicalpy.model import SuperBlock
from dynamicalpy.blocks import *

import numpy as np
from scipy import constants

m = .23 # massa pendulo
M = 2.4 # massa do carro
l = .35 # tamanho do pendulo
g = constants.g # constante gravitacional
f = .1  # coeficiente de fricção entre o pendulo e o carro
F = 0   # Forca aplicada no carro

def dpendulum(t,x,u):
    x1, x2, x3,x4 = x
    F = u[0]
    dx1 = x2
    dx2 = ((-m*g*np.sin(x3)*np.cos(x3) +
             m*l*x4**2*np.sin(x3) +
             f * m * x4 * np.cos(x3) + F ) /
            (M + (1-np.cos(x3)**2)*m))

    dx3 = x4
    dx4 = (((M+m)*(g*np.sin(x3)-f*x4) -
             (l*m*x4**2*np.sin(x3) + F) * np.cos(x3)) /
             (l*(M+(1-np.cos(x3)**2)*m)))
    return np.array([dx1, dx2, dx3, dx4])

outputs= ('car-position', 'car-velocity',
          'pendulum-angle', 'pendulum-angular-velocity')
pend = ContinuousTimeSystem(name='pendulum',
                            outputs=outputs,
                            inputs=1,
                            state_eq=dpendulum,
                            order=4)
pend.set_initial_state([0,0,np.radians(10),0])


# Proportional Controler
def PController(name, P):
    c = Constant(value=P)
    p = Product(inputs=2)
    controller = SuperBlock(name=name, inputs=1, outputs=1,
                  subsystems=[c,p])
    controller.connect(':in0', f"{p.name}:in0")
    controller.connect(f"{c.name}:out0", f"{p.name}:in1")
    controller.connect(f"{p.name}:out0", ":out0")
    return controller

```

```
50 P_angle = PController('angle_control', 30)
51 P_position = PController('pos_control', 0.05)
52 P_velocity = PController('velocity_control',0.05)
53 s = Sum(signs='+++')
54 system = SuperBlock(name='system', outputs=outputs,
55                     subsystems=[pend, s,
56                                 P_angle,
57                                 P_position,
58                                 P_velocity])
59
60 system.connect('pendulum:pendulum-angle',
61                f'{P_angle.name}:in0')
62 system.connect('pendulum:car-position',
63                f'{P_position.name}:in0')
64 system.connect('pendulum:car-velocity',
65                f'{P_velocity.name}:in0')
66
67 system.connect(f'{P_angle.name}:out0',
68                f'{s.name}:in0')
69 system.connect(f'{P_position.name}:out0',
70                f'{s.name}:in1')
71 system.connect(f'{P_velocity.name}:out0',
72                f'{s.name}:in2')
73
74 system.connect(f'{s.name}:out0', 'pendulum:in0')
75
76 for port in outputs:
77     system.connect(f'pendulum:{port}', f'system:{port}')
78
79 record=[f':{port}' for port in outputs]
80 system.run(80, record=record)
81 for port in record:
82     system.plot(port)
83 system.draw_diagram()
```

Código 5.9: Pêndulo invertido

No código 5.9 também se podem verificar algumas técnicas que podem ser utilizadas para modelação de sistemas com este pacote, como por exemplo, a criação *ad-hoc* de determinados subcomponentes de forma repetitiva, neste caso para os três controladores proporcionais, demonstrado nas linhas 39-48.

Capítulo 6

Conclusão

Este trabalho de dissertação de mestrado teve como objetivo principal o desenvolvimento de uma plataforma de simulação de sistemas dinâmicos com base no conceito de diagrama de blocos. Para esse efeito foram definidos quatro sub-objetivos relativos às características que a referida plataforma deveria ter, nomeadamente, permitir a simulação a partir do conceito de blocos, permitir a existência sistemas encapsulados, permitir a visualização gráfica dos sistemas e permitir a visualização gráfica dos valores de *output* dos sistemas.

De uma forma geral, os objetivos foram alcançados e foi criado um pacote que consegue modelar sistemas dinâmicos, com as funcionalidades acima descritas.

6.1 Limitações e Trabalho Futuro

Um dos aspetos do pacote que poderá sofrer melhorias no futuro é a avaliação do *feedforward* que, neste estado do projeto, é avaliado bloco a bloco, uma vez que as funções de cada bloco são definidas pelo utilizador.

Caso se adicionasse suporte à definição de expressões de forma simbólica, este conceito poderia ter em conta, especificamente, quais os pares de portas que possuem características de *feedforward*.

No estado atual do pacote, a partir do momento em que um dos *outputs* apresenta *feedforward*, o bloco é marcado como *feedforward*, ou seja, do ponto de vista externo, o utilizador é forçado a assumir que todos os outputs possuem *feedforward*, limitando por vezes a capacidade expressiva do bloco.

No bloco ilustrado na figura 6.1, o segundo *output* tem feedforward proveniente do primeiro *input*. No entanto, esta relação não se verifica entre o segundo *input* e o primeiro *output*. A ausência da possibilidade de definição

de *feedforward* de forma especifica pode impossibilitar o uso deste bloco em certos ciclos, mesmo não existindo tecnicamente ciclos algébricos.

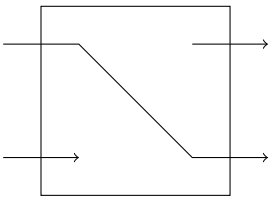


Figura 6.1: Exemplo de *feedforward* parcial

Ou seja, em determinadas situações a avaliação parcial deste bloco, especificamente o primeiro *output*, poderá ser um ponto de partida, da mesma forma que o segundo *input* poderá ser o ponto em que um ciclo algébrico é quebrado. Contudo isto acrescenta um maior nível de complexidade, não sendo evidente a relação de custo-benefício desta funcionalidade.

Outro aspeto a ter em conta futuramente é o modo como a simulação é calculada, em particular, a propagação de *inputs* e *outputs*. Na versão atual, esta propagação é calculada de modo recursivo, no entanto, especialmente em sistemas com níveis profundos de encapsulamento, poderão existir vantagens em efetuar um pré-processamento de forma a que a atualização do sistema seja efetuada de modo sequencial, evitando o custo de múltiplas chamadas às funções de cálculo de *output*, mesmo quando estas possam implementar conceitos de *cache* para evitar cálculos desnecessários.

No estado atual do simulador, a função de atualização de estado dos blocos pode ser chamada múltiplas vezes, dependendo do algoritmo usado no integrador, o que impossibilita a definição de blocos com memória. Devido às suas implicações, esta é a limitação que terá de ser solucionada com mais brevidade, no entanto a sua implementação poderá implicar uma reavaliação de como a toda a simulação em si é calculada.

Também em relação à simulação, o facto do sistema ser agregado num único bloco e ser simulado de forma unificada, impossibilita o uso de múltiplos algoritmos de integração para diferentes componentes do sistema. No entanto este é um aspeto interno de como o sistema é simulado, pelo que a sua alteração não deverá impactar significativamente o utilizador.

Por fim, seria interessante considerar outras ferramentas quer para a visualização dos sistemas elaborados programaticamente, quer para a definição de sistemas de forma gráfica. Atualmente o projeto usa o *graphviz* para a visualização esquemática dos sistemas, no entanto este apresenta limitações que dificultam a obtenção de boas representações sem que sejam feitas afinações manuais. Relativamente à definição de sistemas de forma gráfica,

terão de ser investigadas ou desenvolvidas soluções que, em última instância, poderão ser solução para ambos estes pontos.

Bibliografia

[1] N. Andrei, "Modern control theory," *Studies in Informatics and Control*, vol. 15, no. 1, p. 51, 2006.

[2] E. Fernández Cara and E. Zuazua Iriondo, "Control theory: History, mathematical achievements and perspectives," *Boletín de la Sociedad Española de Matemática Aplicada, 26, 79-140.*, 2003.

[3] M. Kihl, A. Robertsson, M. Andersson, and B. Wittenmark, "Control-theoretic analysis of admission control mechanisms for web server systems," *World Wide Web*, vol. 11, no. 1, pp. 93–116, 2008.

[4] I. M. Lourtie, *Sinais e sistemas*. 2002.

[5] N. M. Karayanakis, *Advanced system modelling and simulation with block diagram languages*. CRC Press, 1995.

[6] T. E. Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 10–20, 2007.

[7] M. Kj, "Aivazis m. python for scientists and engineers," *Comput Sci Eng*, vol. 13, pp. 9–12, 2011.

[8] G. Boeing, "Pynamical: Model and visualize discrete nonlinear dynamical systems, chaos, and fractals," *Journal of Open Source Education*, vol. 1, no. 1, p. 15, 2018.

[9] R. Clewley, W. Sherwood, M. LaMar, and J. Guckenheimer, "Pydstool, a software environment for dynamical systems modeling (2007)," *URL http://pydstool. sourceforge. net*, 2007.

[10] R. Clewley, W. Sherwood, M. LaMar, and J. Guckenheimer, "Pydstool, a software environment for dynamical systems modeling (2007)," *URL https://pydstool. github. io/PyDSTool/ProjectOverview. html*, 2019.

[11] S. Team, "Simpy: Discrete-event simulation for python," *URL: https:// pypi. org/project/simpy*, 2020.

[12] G. Dagkakis, C. Heavey, S. Robin, and J. Perrin, "Manpy: An open-source layer of des manufacturing objects implemented in simpy," in *2013 8th EUROSIM Congress on Modelling and Simulation*, pp. 357–363, IEEE, 2013.

[13] G. Dagkakis, I. Papagiannopoulos, and C. Heavey, "Manpy: an open-source software tool for building discrete event simulation models of manufacturing systems," *Software: Practice and Experience*, vol. 46, no. 7, pp. 955–981, 2016.

[14] R. M. Murray, "Control systems library for python." http://github.com/python-control/python-control, 2020.

[15] B. W. Margolis, "Simupy: A python framework for modeling and simulating dynamical systems.," *J. Open Source Softw.*, vol. 2, no. 17, p. 396, 2017.

[16] "Control of an inverted penduluim - laboration in automatic control," 2012.