Controlador de Semáforos

Relatório Técnico Final - Laboratório de Aquisição e Controlo

Pedro Sousa (up201704307)

Leonardo Hügens (up201705764)

1 Introdução

Neste relatório está documentado o processo de criação de um controlador de semáforos. Para isso, utilizamos o LabView como ferramenta principal, também utilizando um pouco de Python na automatização.

Foi feita a escolha de um local real da cidade do Porto, a respetiva simulação, e o desenvolvimento de um algoritmo que, dadas certas informações sobre o estado do tráfego em tempo real, decide a configuração de cada um dos semáforos que a ela melhor se adequa. A aplicação real destes controladores teria como consequência uma melhor gestão global do tráfego, nomeadamente a diminuição da duração dos congestionamentos diários que ocorrem a certas horas em várias zonas de grandes cidades como o Porto.

2 Especificações iniciais

Em primeiro lugar, começamos por discutir que situação iríamos estudar, com foco em situações reais do área metropolitana do Porto. Começamos por pensar na interceção da Rua Júlio Dínis com a Praça de Mouzinho de Albuquerque pois apresenta uma interação interessante entre o semáforo dos peões e os semáforos de entrada e saída da rotunda. No entanto, decidimos optar por um cruzamento pois abria possibilidades para novas interações. O cruzamento em questão é o cruzamento entre a Rua Latino Coelho e a Rua Alegria (Figura 1).

Escolhemos o LabView para a implementação do controlador, já que nos dá a capacidade de intervir manualmente durante a execução do código, podendo assim simular facilmente bastantes estados do tráfego logo na fase inicial do desenvolvimento, muito antes de precisarmos realmente de automatizar esse processo. Desta maneira, será prática a avaliação contínua da performance do controlador.

Por outro lado, o pacote de interação com um interpretador de Python que o LabView tem é um bom apoio na eventualidade de precisarmos de implementar algo mais complexo como as funções que farão parte da automatização da simulação do controlador.

Figura 1: Cruzamento entre a Rua da Alegria e a Rua Latino Coelho, Porto.

3 Planeamento

Para este projeto, implementamos um processo de evolução natural de trabalho. Isto é, começamos com um programa simples (α_0.1) e fomos inserindo novas características e funcionalidades. Apresentamos agora um breve resumo das várias fases do programa.

- α_0.1: Sistema com semáforo principal e semáforo para peões perfeitamente sincronizados. Apresenta um botão de emergência que, caso esteja ativo, o semáforo para peões é desativado e o semáforo principal fica em amarelo intermitente.

- α_0.2: Reforma à primeira versão do programa. Apresenta as mesmas características do α_0.1, porém, de forma mais compacta, assim, quando fossemos estudar sistemas mais complexos teríamos uma maneira simples de fazer semáforos usando uma subVI de α_0.2.

- α_0.3: Com o insucesso de α_0.2, fazemos a nossa segunda reforma. Processo utiliza uma ideia semelhante à versão 0.2 mas agora fizemos um semáforo à volta de "case structures". Apresenta dois semáforos principais, conectados como se fossem dois semáforos de um cruzamento, e os respetivos semáforos para peões. Apresenta também um mecanismo de acionameto antecipado por controlo de velocidade dos veículos e um mecanismo que, caso haja bastante tráfego na via, o semáforo utiliza essa informação de forma a haver uma desiguldade "positiva"[1] entre o tempo em que está em vermelho e o tempo em que está no verde + amarelo, e assim diminuir naturalmente o tráfego.

- α_0.4: Automatização da simulação através de um script de python com funções que descrevem os rácios de

[1] Com desigualdade "positiva" queremos dizer que o semáforo vai estar mais tempo no verde+amarelo do que no vermelho

entrada ao longo do tempo e outras que com base nos valores destes rácios geram amostras estocásticas da velocidade instânea dos carros.

- α_0.5: Otimização da execução do código Python através da utilização de apenas um script que toma como argumento um array todos os valores necessários e retorna os pretendidos também num array, em oposição a ter um script por valor pretendido.
- α_0.6:

4 Desenvolvimento

Nesta parte do relatório iremos explicar em detalhe as fases do nosso projeto sumariadas anteriormente.

α-0.1

Em primeiro lugar, criamos o semáforo principal, para isso utilizamos booleanos do tipo botão, alterando a cor para o respetivo elemento do semáforo e fizemos uma flat sequence onde por cada frame apenas uma luz estava ligada. Depois criamos mais dois botões referentes ao semáforo para peões e alteramos a sua condição consoante o estado em que o semáforo principal estava (se o semáforo estava vermelho o semáforo para peões ficava verde, (se o semáforo estava verde o semáforo para peões ficava vermelho). Por fim criamos uma case structure que, caso o botão de emergência estivesse ativo corria um ciclo while em que a luz amarela ligava e desligava e caso estivesse desligado corria o nosso programa anteriormente escrito. Por fim, introduzimos um timer que apresenta ao peão quanto tempo falta para o seu semáforo ficar vermelho.

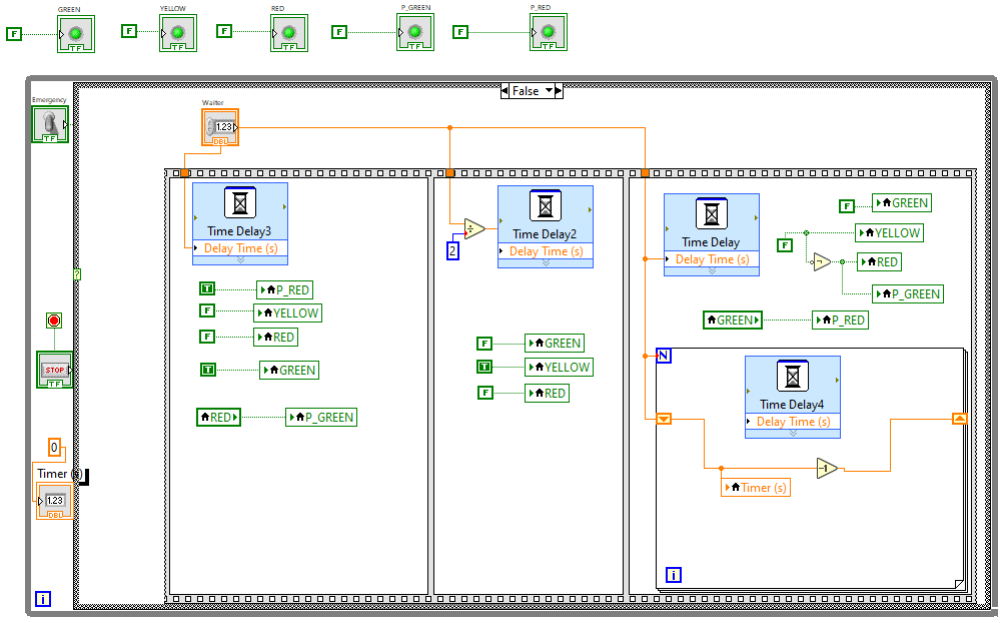

Figura 2: α_0.1

α-0.2

Após termos um propótipo inicial com as funcionalidades básicas (funcionamento normal e situação de emergência), decidimos tentar encontrar uma estrutura de código que nos permitisse construir um subVI com um semáforo e simplesmente clonar essa subVI várias vezes, construindo assim estruturas mais complexas. Um bom candidato para essa estrutura seria o "Event Structure" do LabView, que faz uma "sandbox" a uma certa porção de código e só o executa na ocorrência de um evento, como por exemplo a alteração do valor de uma variável local. Deste modo, a ideia era conter todo o código respetivo a um controlador dentro de uma destas estruturas e contê-la dentro da tal subVI. Depois de fazer esta implementação, ao tentar construir a subVI apercebemo-nos que o LabView não suporta conter um "Event Structure" dentro da mesma, pelo que tivemos que desistir desta ideia.

α-0.3

A ideia inicial era a implementação de um "state" que está associado ao semáforo, isto é:

- Se o semáforo está no estado 0, ele está vermelho e passa para o estado 1 quando se tiver passado no mínimo[2] um tempo $t = timer$.[3]
- Se o semáforo esá no estado 1 ele começa verde e, passado um tempo $t = timer$, muda para amarelo, onde permanece um tempo $t = \frac{timer}{2}$, por fim, muda para o estado 0.

Aplicando isto a um semáforo simples o que acontece é exatamente o esperado, e apenas tivemos de usar um case structure para definir os estados e uma flat sequence para fazer o semáforo ficar verde e depois amarelo.

Agora, decidimos emparelhar outro semáforo. Para isso, estudamos um cruzamento simples entre duas vias de sentido único. Neste exemplo, tudo está em sincronia:

- Quando um semáforo está a permitir passagem (estado 1), o outro não permite (estado 0).
- Quando um semáforo está a permitir passagem (estado 1), o seu semáforo de peões associado não permite.

Para implementar esta ideia fizemos com que os estados dos semáforos estivessem conectados. Isto é, o semáforo 1 está num estado X, então o semáforo 2 está no outro estado.

Mas, com esta teoria deparamo-nos com um problema conceptual. Um semáforo fica no estado 0 um tempo $t = timer$ e fica no estado 1 um tempo $t = 1.5timer$ (1.0 no verde e 0.5 no amarelo). Se os semáforos estão perfeitamente conectados quanto tempo irá o semáforo estar no vermelho? A resposta é $1.5timer$ o que é o esperado, porque o estado só muda quando tudo o que tiver de acontecer naquela iteração do *for* (que é o nosso *for* temporal) terminar, ou seja, $1.5timer$ depois.

Para finalizar, adicionamos duas novas ferramentas:

[2] este "no mínimo" é interessante e será discutido mais à frente

[3] timer é um input do nosso sistema de forma a definir o tempo que o semáforo fica no verde

- Um mecanismo que, caso haja um carro a mais de $50Km/h$, o amarelo é acionado antecipadamente. No entanto, o semáforo fica no verde no mínimo um tempo $t = \frac{timer}{2}$. Este tempo foi introduzido para quando o semáforo fica verde não mudar imediatamente para amarelo, havendo sempre um delay justificado.

- Um mecanismo que, caso haja um enorme tráfego de carros, o semáforo permaneça menos tempo no vermelho de forma a haver um melhor fluído de tráfego.

Estes dois mecanismos estão ligados pois, no fundo, acionar um aviso de congestionamento na via 1 é o equivalente a ligar o mecanismo de velocidade na via 2, pois, se na via 2 está menos tempo no verde, por consequência, na via 1 está menos tempo no vermelho.

Adicionamos uns sliders para representarem o tráfego nas vias e fizemos com que o seu valor diminuísse quando está verde e amarelo e aumentasse quando está vermelho. Adicionamos também um input "rácio de entrada" que, se saem Y carros enquanto está verde, entram "rácio de entrada" ×Y carros enquanto está vermelho. Assim, podemos associar este valor, por exemplo, à altura do dia. Se, na hora de ponta entram 5× mais carros do que saem, à noite entram $\frac{1}{10}\times$ dos carros que saem até que não há trânsito de todo.

α-0.4

Nesta implementação decidimos estudar o caso escolhido, ou seja, o cruzamento da rua Latino Coelho e da rua da Alegria. Para isso, remodelamos o $\alpha\_0.3$ de forma a queria uma melhor visualização do cruzamento no Front Panel do LabView.

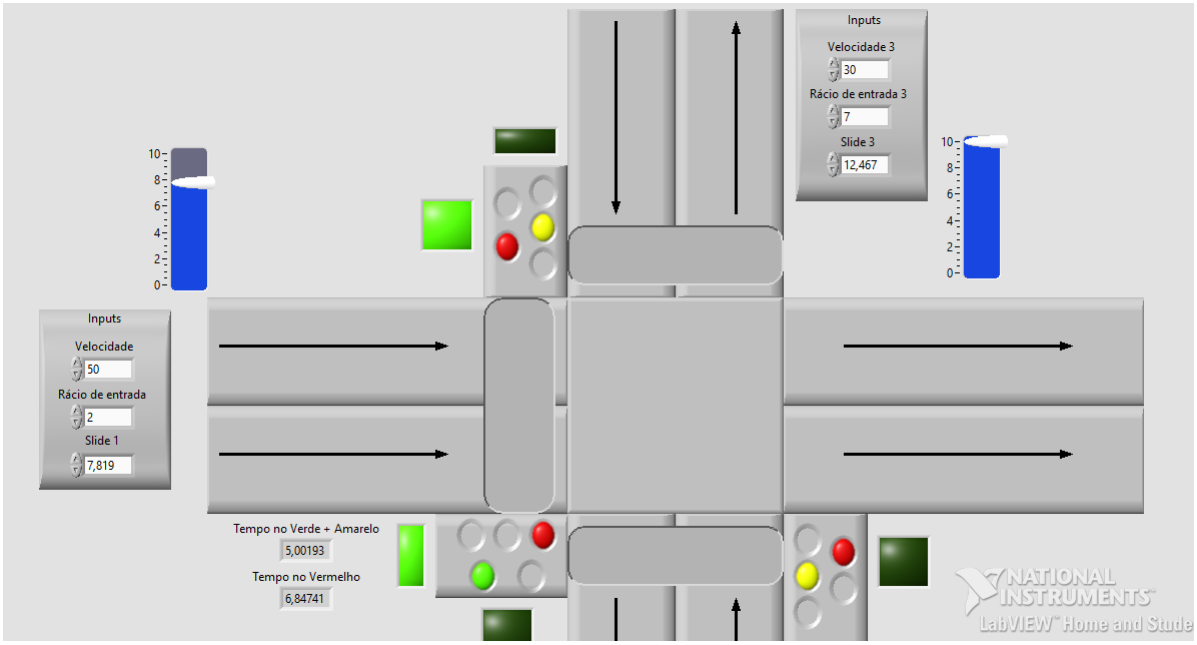

Figura 3: Front Panel do $\alpha\_0.4$ com o esquema do cruzamento entre a Rua Latino Coelho e a Rua da Alegria; o botão retangular está ativo quando a velocidade na via é $\geq 50km/h$; o botão quadrado está ativo quando há congestionamento na via (Valor no Slider ≥ 8)

α-0.5

Decidimos que o projeto estava numa fase suficientemente avançada para implementarmos um simulação automática. Mais pormenorizadamente, essa automatização consistiria em assumir uma determinada evolução temporal do rácio de entrada, em que o pormenor mais importante dessa evolução seria a existência de dois claros máximos ao longo dia: às 8h e às 17h. Como esse fenómeno ocorre com muita consistência, decidimos que uma evolução determinista que consiste numa combinação linear de duas distribuições de Cauchy com os seus máximos nos anteriormente referidos. Como a velocidade instantânea dos veículos é claramente um fenómeno mais aleatório, decidimos gerar a cada instante um valor aleatório da mesma, seguindo também uma distribuição de Cauchy. Contudo, não faria muito sentido que o valor médio da distribuição fosse uma constante, já que é pouco provável que este seja o mesmo em situações de alto ou baixo tráfego. Deste modo, fizemos com que o valor médio fosse uma função do rácio de entrada, mais especificamente uma sigmóide. Esta função é frequentemente utilizadada como função de ativação de redes neuronais.

Segue-se uma breve descrição das funções:

• "genspeed": função que gera velocidades instantâneas. Recorremos ao módulo random do pacote NumPy, ao método da inversão da função cumulativa da distribuição de probabilidade, e ao facto de que a distribuição que escolhemos tem uma primitiva facilmente determinável.

• "cauchy": retorna o valor da distribuição de probabilidade para um dado valor da variável aleatória, do valor médio e do desvio padrão.

• "gettime": usa o tempo real e converte-o num tempo virtual, sendo que estes diferem apenas de um fator multiplicativo.

• "ratio": função que retorna os rácios de entrada dada a altura do dia.

α-0.6

Percebmos que chamar uma instância do interpretador de python para cada um dos valores que queríamos calcular causava problemas de ecidimos juntar todas as funções num só script e fazer mais uma função, que recebe como argumento todos variáveis consideradas num certo instante, chama as restantes funções e retorna todos os resultados noutro array, que é diretamente passado para o LabView. Desta

• "get": recebe o array com os parâmetros controlados no Front Panel, nomeadamente o fator de conversão entre os tempos e os valores instantâneos dos rácios de entrada, chama as restantes funções e organiza todos os valores num array de output que será recebido pela VI.

O código que diz respeito a todas estas funções está presente na seguinte Figura 4.

```python
from datetime import datetime
import numpy as np

###### speed generator ######
def genspeed(slider):
        pi = np.pi
        av_amount = 5
        dev       = 1
        min_dif   = 0
        max_dif   = 30
        av_speed  = -( 1 / ( 1 + np.exp(-((slider-av_amount)*2) ) ) * \
                        (max_dif - min_dif) - max_dif)
        speed     = abs(dev * np.tan(pi * np.random.rand() - (pi / 2)) + av_speed)
        return speed

###### ratio generator ######
def cauchy(x, center, dev):
    return dev / ( dev**2 + (x-center)**2 )

def gettime(conversion):
    rate = 24 / ( conversion/60 )
    a=datetime.now().time()
    realt = ( a.hour + a.minute/60 + a.second/60**2 )
    virtualt = (realt*rate) % 24
    return virtualt

def ratio(conversion): # conversion in minutes
    t = gettime(conversion)
    return (cauchy(t, 8, 3) + cauchy(t, 17, 3))*20

###### functions called in labview ######
def get(array):
    return [ratio(array[0]),    \
            genspeed(array[1]), \
            genspeed(array[2]), \
            genspeed(array[3])]
```

Figura 4: Script de Python.

5 Avaliação de desempenho

Resumidamente, o comportamento geral do controlador em siconsiste nas seguintes funcionalidades:

- Controlo do intervalo em que os semáforos permanecem com luz verde/vermelha com base no quantidade de veículos da via.
- A existência de um caso de emergência, no qual todos os semáforos entram numa intermitência da luz amarela.
- A passagem atempada para vermelho do semáforo da via onde um carro estava a ultrapassar um certo limite de velocidade, obrigando-o a imobilizar-se.

Na sua última versão do código, o algoritmo desempenha toda estas funções. A única parte da execução que pode não apresentar uma performance consistente é a parte da automatização. Como a utilização explícita do "Runtime" do LabView não é a mais simples, decidimos utilizar o tempo real do sistema onde o código está a correr e trabalhar com um tempo "virtual", sendo que estes diferem por uma constante mulptiplicativa. Deste modo, há sempre o limite em que o ritmo relativo de passagem do tempo virtual é demasiado elevado, maior que o tempo de processamento normal do LabView, e podem surgir complicações. Deste modo decidimos não sintonizar os tempos

durante os quais os semáforos estão nas respetivas configurações, já que tornava um pouco difícil a visualização das várias funcionalidades e podia resultar em mais complicações como as mencionadas.

6 Implementação real - refleções

Se pensarmos no controlador em si como software, a sua implementação na vida real seria em princípio bastante fácil. Contudo, o LabView, como se pode entender pelo nome, foi feito para utilização com um Display Manager, e apenas para Windows e MacOS. A melhor implementação dos controladores seria a utilização do software num dispositivo muito acessível e económicos como o Arduino ou o Raspberry Pi, que foram feitos para desempenhar um número limitado de processos, como seria o caso do nosso controlador. Assim, era apenas uma questão de converter o nosso algoritmo para uma linguagem mais apropriada como Python, C ou C++. Contudo, já que o nosso algoritmo envolve a recolha de informações como o número de carros na via e a sua velocidade instantânea, teríamos de adicionar um aparelho que recolhesse estes dados.

7 Conclusão

(...)

Recursos

https://github.com/grupo1lac/semaforo/ - Repositório de GitHub do projeto, contendo todo o código associado à evolução do projeto.