ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
Departamento de Engenharia de Computação e Sistemas Digitais

PCS 3111 - Laboratório de Programação Orientada a Objetos para a Engenharia Elétrica

Exercício Programa 1 – 2º Semestre de 2023

1 Introdução

O Matlab é um programa muito utilizado nos estudos e projetos de engenharia elétrica. Há razões para utilizá-lo nas quatro ênfases de elétrica, sobretudo nas duas do departamento PTC (Telecomunicações e Controle).

Existem programas similares, como o Octave, que é bastante usado por alunos da Poli, principalmente na sua versão online. Porém, neste exercício programa (e no próximo) iremos construir um programa que também poderá ser utilizado para fins de simulação, tratando especialmente de projetos de controle e automação que vocês estudarão em **PTC3313 - Sistemas de Controle** e o seu respectivo laboratório, **PTC3312 - Laboratório de Controle**.

1.1 Objetivo

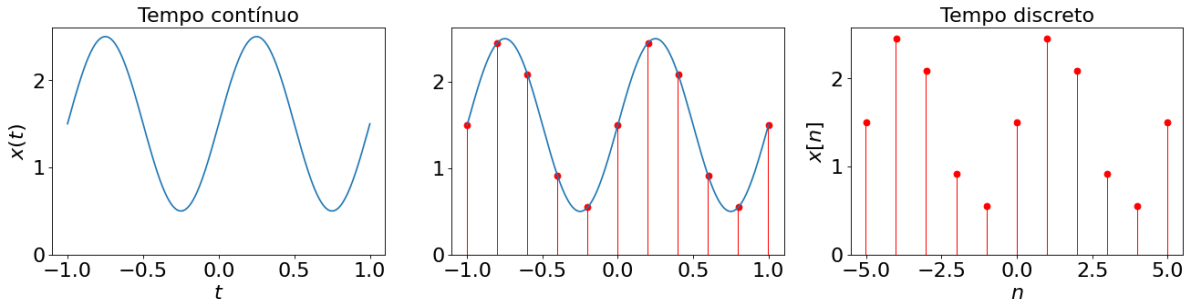
O objetivo deste projeto é fazer um programa de simulação análogo ao Simulink, componente do Matlab que será muito utilizado também na disciplina **PTC3307 - Sistemas e Sinais**. Este projeto deverá ser desenvolvido incrementalmente e **em dupla** nos dois Exercícios Programas de PCS3111.

Neste primeiro EP será implementada uma arquitetura de blocos operadores que processam Sinais entre suas entradas e suas saídas. No próximo EP serão feitas melhorias nessa arquitetura, por exemplo, para permitir a construção de circuitos mais complexos e mais práticos de serem utilizados.

A solução deve empregar adequadamente conceitos de Orientação a Objetos apresentados na disciplina: classe, objeto, atributo, método, encapsulamento, construtor e destrutor – o que representa o conteúdo até, *inclusive*, a Aula 5. A qualidade do código também será avaliada (nome de atributos/métodos, nome das classes, duplicação de código etc.).

1.2 Sinais

Por se tratar de um ambiente computacional, os **sinais** (que são **funções**), são representados como vetores (*arrays*). Ou seja, ao invés de termos infinitos pontos indexados pela reta real como nas funções dos estudos de cálculo e física, teremos pontos finitos indexados por números inteiros. Com isso, passamos do mundo das funções **contínuas** para as funções **discretas** (que ocorre por um processo de amostragem estudado com rigor em **PTC3361 - Introdução ao processamento digital de sinais**).



1.3 Circuitos Operadores

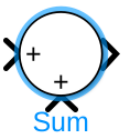
Os sinais serão passados por **circuitos** que realizam **operações.** As operações são relativas ao domínio do tempo discreto, logo, terão algumas modificações em relação àquelas que estamos acostumados a utilizar nos cursos de Cálculo e Física. Os circuitos que recebem apenas um sinal de entrada e uma saída são chamados de circuitos **SISO** (*Single Input, Single Output*), enquanto que os circuitos que recebem duas entradas e possuem uma saída são circuitos **MISO** (*Multiple Input, Single Output*).

1.3.1 Amplificador

O **bloco amplificador** é um circuito que simplesmente multiplica todo o sinal que passa por ele por uma constante, também chamada de **ganho**. Para facilitar, no EP o amplificador é restrito a uma constante real, portanto não se usarão constantes complexas. Nos programas de simulação, este bloco pode ser retratado como na figura a seguir.

1.3.2 Somador

Outro circuito importante é o **bloco somador**. Ele é um circuito **MISO**: ele realiza a soma de dois sinais diferentes e retorna um sinal resultante dessa soma, termo a termo. Ele tipicamente é representado como na figura abaixo.

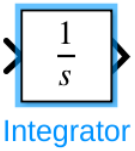


1.3.3 Integrador

Em se tratando de engenharia, é difícil fugir das operações estudadas nas disciplinas de cálculo! Para os cálculos realizados neste EP, a **operação de integral** será realizada a partir da seguinte convenção:

$$y(t) = \int_0^t x(\tau)\partial\tau$$

Ou seja, a operação do **bloco integrador** integra de 0 até t, a função de entrada. Ele pode ser representado pelo bloco da figura a seguir.



Contudo, como estamos no mundo discreto, será necessário fazer uma alteração de modo a trocar a operação de integral pela de **somatória** dos valores indexados de 0 a t, afinal, estaremos tratando de um número finito de elementos. Logo, a operação de integral será implementada como:

$$y[n] = \sum_{i=0}^{n} x[i]$$

o que também pode ser reescrito como:

$$y[n] = y[n-1] + x[n]$$

1.3.4 Derivador

Para realizar a **operação de derivada** usa-se o seguinte bloco em Simulink:

Além disso, nessa operação também será necessário fazer uma mudança para trabalhar com dados discretos. Um apelido dela é "**taxa de variação**", ou seja, pode-se compreender como "quanto é a variação da função naquele instante de tempo". Para trabalhar com dados discretos e ficar de acordo com a função de integral definida anteriormente, essa operação será realizada por meio da diferença entre as medidas de dois instantes consecutivos. Assim um derivador deve fazer:

$$y[n] = x[n] - x[n-1]$$

1.3.5 Outros operadores

Para abranger todas as principais operações dos circuitos de controle linear, faltaria um último bloco operador que fizesse a **convolução** da entrada por uma exponencial decrescente, ou seja $y(t) = x(t) * e^{at}u(y)$, em que u(t) seja o degrau unitário. Como assim? Simples, um bloco que possua um **polo fora na origem,** de preferência com parte real negativa para que seja um **sistema estável**. Ainda não entendeu? Talvez seja melhor o apresentarmos assim: a **transformada de Laplace** da sua **função de transferência** é da forma $\frac{a}{s-a}$. Faz sentido? Bom, como parece ser necessário muito trabalho para explicar tudo isso, vamos deixar que as disciplinas **PSI3211 - Circuitos I** e **PSI3213 - Circuitos II** façam isso ao decorrer do curso. Quem sabe no futuro você não se anima e volta para completar esse programa com o bloco que vai ficar faltando?

Obs: Se você ainda quiser utilizar esse simulador para rodar filtros de telecomunicações, como aqueles que serão estudados em **PTC3360 - Introdução a Redes e Comunicações**, só precisaria acrescentar dois blocos mais simples: o **multiplicador** e o **convolucionador** (que na verdade não é tão simples quanto o primeiro e por isso deixaremos também para o futuro).

1.4 Modelagem de um piloto automático

Para que esse EP não fique somente com modelos teóricos, resolvemos simular uma aplicação real dos estudos de controle: **um piloto automático de carro**. Para isso, será preciso modelar um carro, um pé que pisa no acelerador e uma mente que controla esse pé.

1.4.1 O motor do carro

Para esse experimento o motor do carro será muito simples. Ele é somente algo que recebe um valor de **aceleração** proveniente do pedal do acelerador e produz, assim, uma **velocidade**. Ficarão para o EP2 o atrito e outros detalhes.

Enfim, se a entrada é uma aceleração e a saída uma velocidade, sem atrito, então esse motor nada mais é do que um **integrador**! Por isso, para o modelo dele será utilizado um operador do tipo integrador como descrito anteriormente.

1.4.2 O pé que acelera

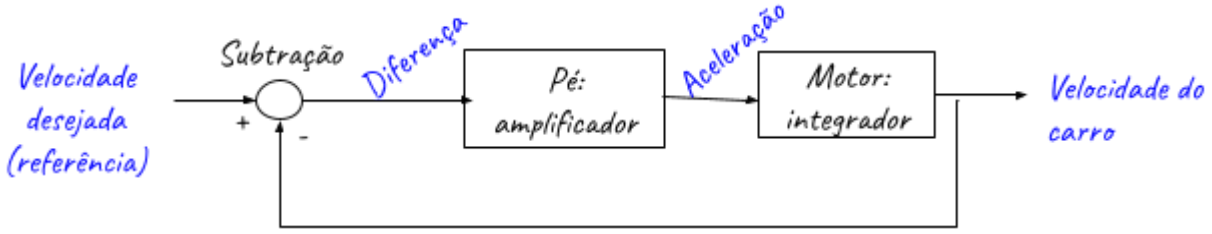
Seja com um motorista real, seja com um motorista robô, será necessário em ambos os casos algo que acelere o carro.

Logo, o princípio de funcionamento nunca muda: algo decide que precisa acelerar e então algo - como um pé - aciona o acelerador. Mas quanto? Um motorista mais apressado pode *pisar fundo*, mesmo que precise aumentar somente um pouco a velocidade. Se for um motorista mais calmo, ele vai *pisar pouco no pedal*. Por isso, para modelar esse sistema de acionamento do acelerador vamos utilizar um **Amplificador.** Assim, o quanto o motorista pisa é representado pelo valor do ganho do amplificador. Quanto mais apressadamente ele pisa, maior o ganho, ou seja, mais aceleração ele passa para o motor do carro.

1.4.3 A inteligência que controla o pé

Agora falta somente entender como será modelada a **tomada de decisão**. Na vida real, o humano (ou o robô) lê no velocímetro o valor atual da velocidade do carro e decide se vai acelerar ou não (podemos considerar aqui que esse acelerador até freia, ou seja, consegue passar valores de aceleração negativos para o carro). Assim, o comando que o cérebro passa para o pé depende da diferença entre a velocidade atual do carro e a velocidade desejada (a referência).

Todo esse processo de controle da velocidade é chamado de "Sistema de controle realimentado", isto é, um sistema que no processo de tomada de decisão leva em consideração o resultado do que acabou de acontecer. Esse modelo será estudado muito em **PTC3313 - Sistemas de Controle** mas, para facilitar a compreensão, segue abaixo um esquema de como ele deve funcionar.



No início do processo se considera que a velocidade do carro é 0. Portanto, na primeira iteração se tem como diferença a velocidade desejada, o que leva a uma aceleração elevada. Na próxima iteração o carro já alcançou alguma velocidade, a qual é considerada para calcular a diferença com a velocidade desejada. Isso levará a uma aceleração menor do que na primeira iteração. Isso será feito repetidamente de forma que a velocidade do carro tende a se aproximar à velocidade desejada.

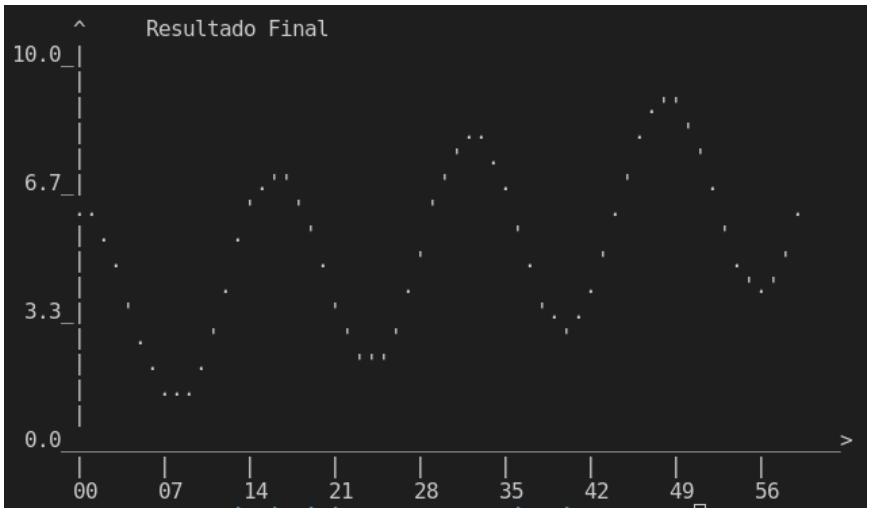
1.5 Biblioteca "Grafico.h"

Para facilitar a visualização das funções no próprio terminal, é fornecida uma biblioteca simples para imprimir os sinais criados. Seguindo o princípio da ocultação da informação, você pode considerá-la como uma caixa-preta, com a seguinte interface pública:

```cpp
class Grafico {
public:
  Grafico(string titulo, double *sequencia, int comprimento);
  virtual ~Grafico();
  void plot();
};
```

- O **construtor** recebe um vetor de double que contém a sequência de valores a serem apresentados no gráfico e o respectivo comprimento desse vetor. Além disso, há um parâmetro que permite escolher um título a ser inserido no gráfico.
- Para imprimir o gráfico no terminal basta utilizar a função `plot()`, que teve seu nome mantido em inglês para fazer referência à função de mesmo nome do Matlab.

Vale ressaltar que esse código é bem simples (afinal foi um monitor da disciplina quem a fez): ela possui o eixo das abscissas limitado em até 60 pontos e as coordenadas são fixadas de 0 a 10. O bom é que se você desejar, você pode olhar o código para entender como ele funciona. Segue um exemplo de um gráfico gerado por ela:



2 Projeto

Deve-se implementar em C++ as classes `Sinal`, `Amplificador`, `Somador`, `Integrador`, `Derivador`, `ModuloRealimentado` e `Piloto` além de criar uma `main` que permita o funcionamento do programa como desejado.

Atenção:

1. O nome das classes e a assinatura dos métodos **devem seguir exatamente** o especificado neste documento. As classes não devem possuir outros membros (atributos ou métodos) **públicos** além dos especificados. **Note que você poderá definir atributos e métodos privados, caso necessário**.
2. Não faça outros `#defines` de constantes além dos definidos neste documento. Você pode (e deve) somente fazer `#ifndef/#define` para permitir a inclusão adequada de arquivos.

O não atendimento a esses pontos pode resultar em **erro de compilação** na correção automática e, portanto, nota `0` na correção automática.

Cada uma das classes deve ter um arquivo de definição (".h") e um arquivo de implementação (".cpp"). Os arquivos devem ter exatamente o nome da classe. Por exemplo, deve-se ter os arquivos "Sinal.cpp" e "Sinal.h". Note que você deve criar os arquivos necessários.

A sugestão é que se comece a implementar o EP a partir da *Aula 4*. Todas as classes possuem construtores e destrutores (conceito da *Aula 5* – são os métodos com o mesmo nome da classe), mas é possível começar a implementar os métodos ainda sem esse conceito. Coloque a palavra *virtual* nos destrutores, como indicado. Não se preocupe com isso – será explicado o significado na Aula 7.

2.1 Classe Sinal

Um `Sinal` irá guardar um vetor `sequência` de tamanho `comprimento`. Essa classe deve possuir apenas os seguintes **métodos públicos**:

```
Sinal(double *sequencia, int comprimento);
virtual ~Sinal();
double* getSequencia();
int getComprimento();
void imprimir(string nomeDoSinal);
```

- O **construtor** recebe um ponteiro para a sequência e o seu comprimento. O vetor `sequencia` deve ser **copiado em um outro atributo interno (alocado dinamicamente)**, para que não haja perdas de dados se por alguma razão esse vetor passado como parâmetro for apagado prematuramente.
- O método `getSequencia` deve retornar um ponteiro para o vetor armazenado. O método `getComprimento` retorna o comprimento da sequência, informado no construtor.
- Como o vetor será alocado dinamicamente no construtor, não se esqueça de destruí-lo no **destrutor**.
- O método `imprimir` deverá apresentar no terminal uma representação gráfica da função-sinal armazenada. Para isso, deve-se utilizar a classe `Grafico` fornecida. Basta criar um objeto `Grafico` com a `sequencia` e o seu respectivo `comprimento`, além do título do gráfico que será o `nomeDoSinal`, e executar o método `plot` .

Atenção: se for instanciado algum objeto com o comando `new`, será necessário o destruir após a sua utilização.

Obs.: o método `Grafico::plot` por padrão restringe os eixos ao imprimir no terminal da seguinte forma: o eixo das coordenadas limita os valores ao intervalo de 0 a 10 e o eixo das abscissas representa sequências de no máximo 60 amostras. Caso queria alterar isso em seus testes, basta alterar as constantes definidas em `Grafico.cpp.` Lembre-se somente de retornar os valores iniciais ao submeter a resolução no Judge.

2.2 Classes de Circuitos

Serão criadas algumas classes que representam circuitos.

- `Amplificador`
- `Somador`
- `Derivador`
- `Integrador`
- `Piloto`
- `ModuloRealimentado`

O método mais importante dessas classes é o `processar`. Ele recebe como parâmetro os sinais de entrada do circuito, realiza com eles a operação designada e retorna um novo objeto `Sinal` com a nova sequência gerada a partir do resultado das operações.

Obs. 1: Não destrua os `Sinais` passados como parâmetros. Quem os passou pode ainda os querer utilizar.

Obs. 2: O método `processar` irá instanciar um novo objeto dinamicamente alocado, mas não o destruirá ali. Deixaremos a responsabilidade de o destruir para quem chamar esse método. Por mais que não seja tão elegante, é necessário para o EP.

2.2.1 Amplificador

Esse operador deve possuir os seguintes métodos públicos:

```
Amplificador(double ganho);
virtual ~Amplificador();
Sinal* processar(Sinal* sinalIN);
void setGanho(double ganho);
double getGanho();
```

- O construtor recebe um ganho, o qual é retornado pelo método `getGanho` e que pode ser alterado pelo método `setGanho`. O destrutor não tem comportamento específico.
- O método `processar` deverá criar um novo objeto `Sinal` com uma nova sequência criada a partir da `sequencia` de `sinalIN` com cada elemento **multiplicado pelo ganho**. Por exemplo, para um sinal {1, 2, 3, 4}, ao amplificá-lo com um ganho de 1.1, teremos um sinal de {1.1, 2.2, 3.3, 4.4}.

2.3 Somador

Essa classe deve possuir os seguintes métodos públicos:

```
Somador();
virtual ~Somador();
Sinal* processar(Sinal* sinalIN1, Sinal* sinalIN2);
```

- Esse método `processar` deve retornar um ponteiro para um novo `Sinal` com uma sequência que seja a **soma "termo a termo"** dos elementos das sequências dos dois sinais de entrada passados como argumentos. Por exemplo, para os sinais de entrada {1, 2, 3, 4} e {1, 3, 1, 1}, a saída deve ser o sinal {2, 5, 4, 5}. Caso os sinais tenham tamanhos diferentes, o sinal de saída deve ter o menor tamanho dos sinais de entrada. Por exemplo, para os sinais {1, 2, 3, 4} e {1, 3}, o sinal de saída deve ser {2, 5}.

2.4 Derivador

Essa classe deve possuir os seguintes métodos públicos:

```
Derivador();
virtual ~Derivador();
Sinal* processar(Sinal* sinalIN);
```

- Esse método `processar` deve retornar um ponteiro para um novo `Sinal` com uma sequência que seja a **diferença** dos valores da entrada consecutivos, conforme

explicado anteriormente em 1.3.4. Por exemplo, para um sinal de entrada {1, 2, 3, 5}, a saída deve ser {1, 1, 1, 2}.

Importante: considere condições iniciais nulas para o primeiro valor do vetor, ou seja, que antes do sinal de entrada, esta era preenchida por zeros. Logo :

$$y[0] = x[0] - 0 = x[0]$$

2.5 Integrador

Essa classe deve possuir os seguintes métodos públicos:

```
Integrador();
virtual ~Integrador();
Sinal* processar(Sinal* sinalIN);
```

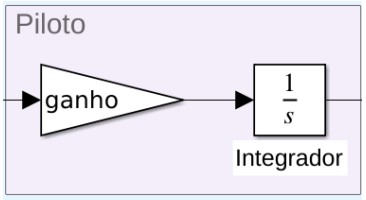
- Esse método `processar` deve retornar um ponteiro para um novo `Sinal` com uma sequência que seja a **somatória acumuladora** dos valores do sinal de entrada, conforme explicado anteriormente em 1.3.3. Por exemplo, para um sinal de entrada {1, 2, 3, 4, 5} o sinal de saída deve ser {1, 3, 6, 10, 15}.

Importante: também considere condições iniciais nulas para o primeiro valor do vetor:

$$y[0] = x[0] + 0 = x[0]$$

2.6 Piloto

Para modelar um piloto automático, conforme descrito na seção **1.4**, será necessário antes de tudo modelar o sistema do piloto não automático, que chamaremos somente de "Piloto". Esse sistema contém o pé do acelerador atrelado ao motor do carro, o que pode ser modelado simplesmente com um amplificador e com um integrador em sequência.



Por isso, essa classe `Piloto` deverá atender a essas especificações, com os seguintes métodos públicos:

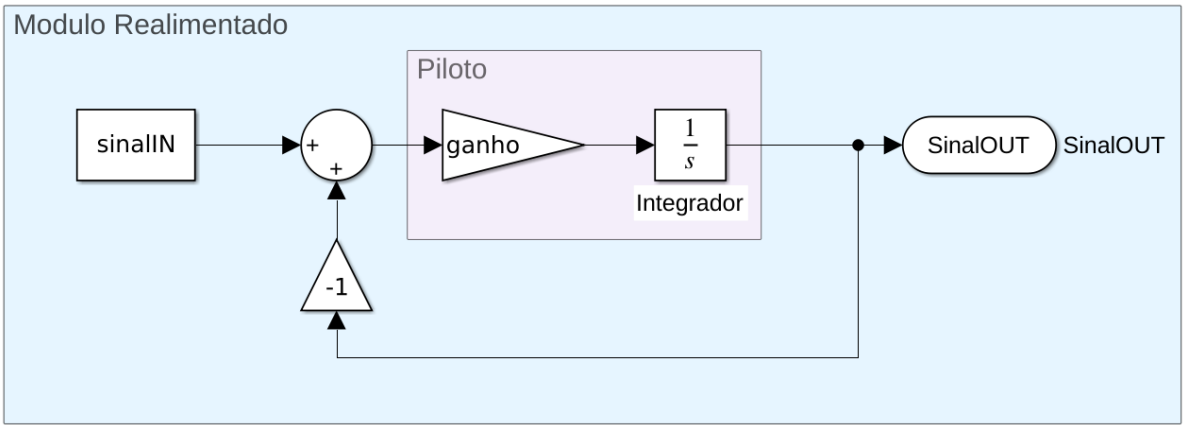
```
Piloto(double ganho);
 virtual ~Piloto();
 Sinal* processar(Sinal* sinalIN);
```

- Seu construtor recebe um ganho que será atribuído ao seu amplificador interno.
- O método `processar` deve retornar um ponteiro para um novo `Sinal` com a saída resultante de um processamento em série do sinal de entrada por um `Amplificador` de ganho `ganho` e um `Integrador`.

Obs.: internamente poderão ser instanciados alguns objetos para fazer essas operações. Caso isso ocorra, lembre-se de os destruir no momento apropriado.

2.7 ModuloRealimentado

Por fim, a última classe `ModuloRealimentado` deverá ser a inteligência descrita na Seção **1.4.3,** de forma a enfim criarmos um Piloto Automático. Seu esquema feito em Simulink pode ser visto abaixo já com os componentes detalhados:

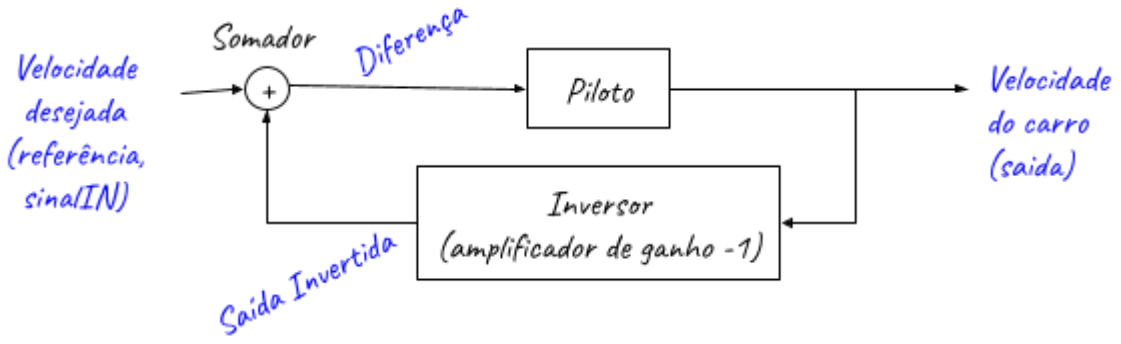


Observe que os circuitos internos dessa classe são um `Somador` e um `Amplificador` (para juntos fazerem a subtração), além de um `Piloto`. Porém, observe que externamente ela aparenta ser similar aos demais circuitos:

```
ModuloRealimentado(double ganho);
virtual ~ModuloRealimentado();
Sinal* processar(Sinal* sinalIN);
```

- Seu construtor recebe um valor de ganho a ser repassado ao modelo interno de `Piloto`.
- Seu método `processar` deve retornar um ponteiro para um novo `Sinal`. Este deverá ser criado a partir de inúmeras iterações desse sistema fechado. Sua implementação não será a mais otimizada possível por conta da escolha de arquitetura das demais classes. Porém, para facilitar, será fornecida uma descrição detalhada do seu funcionamento e do que deverá ser inserido em seu interior.

Primeiramente, utilizaremos o seguinte esquema para o seu circuito interno:



Para melhor compreensão do seu funcionamento, imagine que deseja-se um carro, com seu piloto com ganho de 0,3 andando a uma velocidade de 10Km/h durante 20s. Então, o primeiro ponto a ser observado é que o sinal de entrada (velocidade desejada) será uma função constante com valor 10 e tamanho 20. Ou seja:

`sinalIN` = {10 10 10 10 10 10 10 10 ...} (isso seria passado como parâmetro para a função)

Além disso, considera-se que o carro inicia parado, logo sua velocidade inicial é igual a 0 (nula).

`vInicial` = 0

Dessa forma, a sequência de passos do método `processar` deverá ocorrer como na tabela abaixo:

Iteração	Saída invertida	Diferença	Saída (vel. do carro)
0	{0} ([vInicial] * (-1) = 0 * (-1) = 0)	{10} (10 + 0 = 10)	{3} ([vInicial] + 0.3 * 10 = 3)
1	{0, -3} (3 * (-1) = -3)	{10, 7} (10 + (-3) = 7)	{3, 5.1} (3 + 0.3 * 7 = 3 + 2.1 = 5.1)
2	{0, -3, -5.1} (5.1 * (-1) = -5.1)	{10, 7, 4.9} (10 + (-5.1) = 4.9)	{3, 5.1, 6.57} (5.1 + 0.3 * 4.9 = 6.57)
3	{0, -3, -5.1, -6.57}	{10, 7, 4.9, 3.43}	{3, 5.1, 6.57, 7.60}
...			
8	{0, -3, -5.1, -6.57, -7.60, -8.32, -8.82, -9.18, -9.42}	{10, 7, 4.9, 3.43, 2.40, 1.68, 1.18, 0.82, 0.58}	{3, 5.1, 6.57, 7.60, 8.32, 8.82, 9.18, 9.42, 9.60}
...			

Observe que a Saída Invertida sempre acaba pegando os valores de Saída da iteração anterior Essa é a escolha sugerida no programa. Enfatizamos que essa não é a forma mais otimizada, mas funciona para aquilo que precisamos e para fins didáticos.

Uma outra forma de representar esse algoritmo é por meio de atribuições (o que pode ser útil caso você não tenha entendido a tabela):.

```
saida[0] = Processar_do_piloto(sinalIN[0] + Inversor( vInicial ) )
         = Processar_do_piloto(sinalIN[0] + 0 )
saida[1] = Processar_do_piloto(sinal[1] + Inversor(saida[0])[0] )
saida[2] = Processar_do_piloto(sinal[2] + Inversor(saida[de 0 a 1])[1])
saida[3] = Processar_do_piloto(sinal[3] + Inversor(saida[de 0 a 2])[2])
saida[4] = Processar_do_piloto(sinal[4] + Inversor(saida[de 0 a 3])[3])
saida[n] = Processar_do_piloto(sinal[n] + Inversor(saida[de 0 a n - 1])[n - 1])
```

Enfim, dada a explicação do funcionamento, segue abaixo a descrição do código que implementa essa solução:

- **Declarações iniciais:**
    - Observa-se no esquema apresentado acima que há 4 pontos do sistema nos quais teremos que manipular os sinais: `sinalIn`, `diferença`, `saida` e `saidaInvertida`. O primeiro, `sinalIn`, será recebido como parâmetro da função. Por isso será necessário declarar os outros três ponteiros para `Sinal`. Não é necessário criar qualquer objeto neste instante, por isso, atribuímos a esses ponteiros a constante de C++ `nullptr.`

- O sinal `saidaInvertida` terá seus valores deslocados em relação ao sinal `saida`. Por isso, teremos que alocar um vetor de double com tamanho idêntico ao sinal de entrada para ser a sequência do sinal `saidaInvertida` `(sequenciaSaidaInvertida).`
- Por fim, para facilitar a compreensão do código, pode ser interessante criar uma variável (que será constante) chamada de `velocidadeInicial` ou `vInicial` que possui valor 0.

- **Caso inicial**
  - A primeira posição de `sequenciaSaidaInvertida` receberá a velocidade inicial vezes -1. Ou seja, será zero.
  - Como a velocidade inicial é zero, a diferença entre a velocidade esperada e a inicial é simplesmente igual à velocidade esperada. Por isso, o ponteiro `diferença` receberá um novo objeto `Sinal` criado com o primeiro elemento da sequência do `sinalIN` (logo, possui tamanho 1).
  - A velocidade de saída receberá o valor retornado pelo `processar` do `piloto` com a `diferença` sendo seu parâmetro de entrada.
  - Por fim, uma vez que o sinal `diferença` já foi utilizado, é importante destruí-lo.

- **Iterações**

Cada iteração terá um índice `i,` que será iniciado como 1, pois já fizemos a iteração 0.
  - Primeiro, para iniciar a iteração é necessário processar no inversor a posição `i-1` da saída e colocar na posição i da `sequenciaSaidaInvertida`.
  - Depois, cria-se o sinal `saidaInvertida` com a sua respectiva sequência e comprimento `i+1,` ou seja, com os valores calculados até aqui.
  - Atribui-se ao ponteiro `diferença` o sinal retornado pelo somador ao processar o sinal de entrada e a `saidaInvertida`
  - Antes de gerar um novo `Sinal` de saída é necessário destruir o já existente.
  - Agora pode-se criar uma nova `saida` decorrente do processamento no carro da `diferença`.
  - Por fim, para não acumular lixo para a próxima iteração, é preciso destruir os sinais intermediários `saidaInvertida` e `diferença`.

- **Finalização**
  - Após terminar a iteração se pode destrutir o vetor `sequenciaSaidaInvertida` que foi alocado dinamicamente.
  - Por fim, retorna-se o sinal saída como resultado desse processo todo

3 Main e menu.cpp

Coloque a `main` em um arquivo separado, chamado `main.cpp`. Nele você deverá simplesmente chamar uma função menu, a qual ficará no arquivo `menu.cpp`. Não faça include de menu no arquivo com o main (jamais faça include de arquivos .cpp). Portanto, o `main.cpp` deve ser.

```
void menu();

int main() {
    menu();
    return 0;
}
```

O menu deve criar um programa que se utiliza as classes especificadas para criar sinais e circuitos que os processem conforme o desejo do usuário. Será possível realizar atividades em **dois modos principais**: um **modo livre** em que o usuário cria um sinal e realiza operações em cima dele e um outro modo que simula um **piloto automático** conforme descrito anteriormente.

3.1 Biblioteca cmath

Além das bibliotecas apresentadas no curso e os arquivos ".h" das classes criadas, será necessário incluir a biblioteca <cmath> padrão de C++. Ela será utilizada para criar um sinal cossenoidal com sua função `cos` e sua constante interna π intitulada de M_PI.

3.2 Sinais disponíveis ao usuário

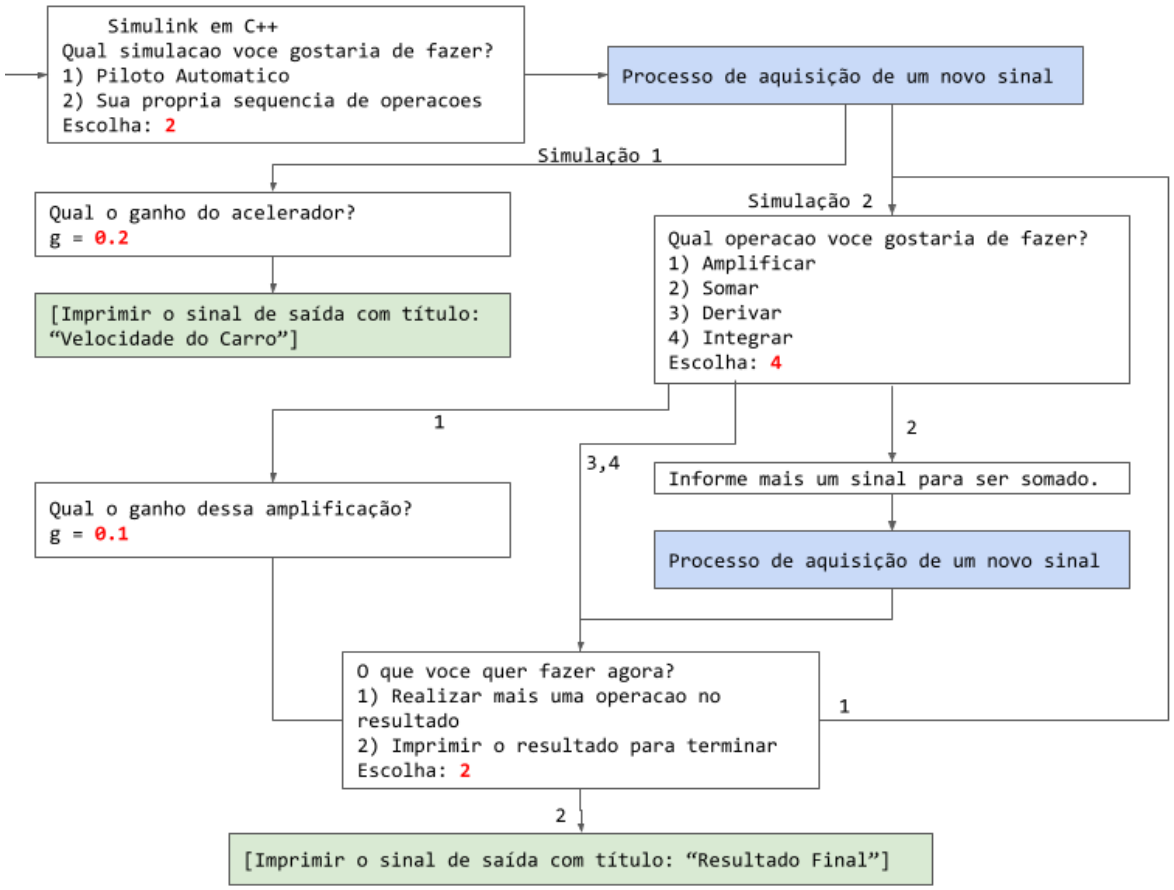
Os sinais de entrada disponíveis para o usuário utilizar poderão ser de três formas: constante, rampa e 5+3*cos(n*pi/8). Para gerar esse último formato, indica-se a utilização da biblioteca cmath citada acima e ressalta-se que o valor de n inicia-se em zero, logo o início da sequência é {8; 7,7716; 7,12132; 6,14805; 5; ...}.

Além disso, explicita-se que a rampa é definida por: n*inclinação, ou seja, dada, por exemplo, uma inclinação de 0.5, a rampa ficaria assim: {0; 0.5; 1; 1.5; 2; 2.5; ...}

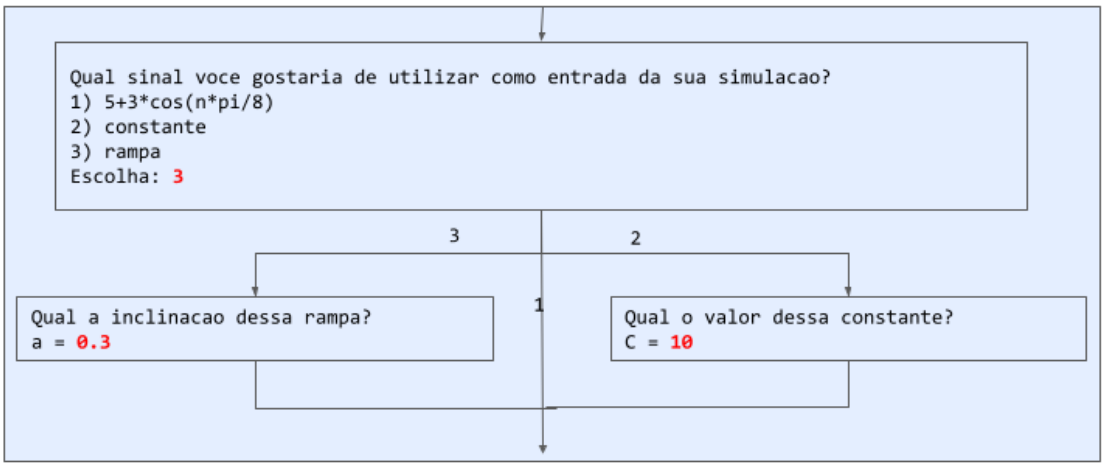
Obs.: considere que todos os sinais criados pelo usuário possuem tamanho 60.

3.3 Interface

Para que o usuário interaja com o programa é necessária uma interface no terminal. A seguir é apresentado um diagrama que contém as telas do programa. Em **vermelho** estão os dados digitados pelo usuário. Os blocos brancos são as impressões que serão lançadas no terminal. Em verde estão os blocos que indicam a finalização do programa com a execução de uma função `imprimir` do `Sinal`. Por fim, em azul estão os blocos que referenciam a ação de adquirir um novo sinal, apresentado posteriormente.



*Processo de aquisição de um novo sinal



Observação: pode-se considerar que o usuário somente digitará entradas com escolhas possíveis. Ou seja, não é preciso tratar entradas incorretas.

Atenção: A interface com o usuário deve seguir exatamente o especificado (incluindo ":" e espaços entre ")" e ":" e o restante do texto - os demais espaços não serão verificados). Se ela não for seguida, haverá desconto de nota. Não adicione outros textos além dos apresentados no diagrama e especificados.

3.4 Sugestões de implementação

O processo de aquisição de um **novo sinal** do usuário foi representado isoladamente, pois seu bloco foi referenciado mais de uma vez. Isso indica que uma boa maneira de implementar essa tela seja criando uma função, por exemplo `Sinal* novoSinal()`, que execute todo o processo de aquisição de um novo sinal de entrada e retorne um ponteiro para esse objeto criado. Assim, haverá um reaproveitamento de código.

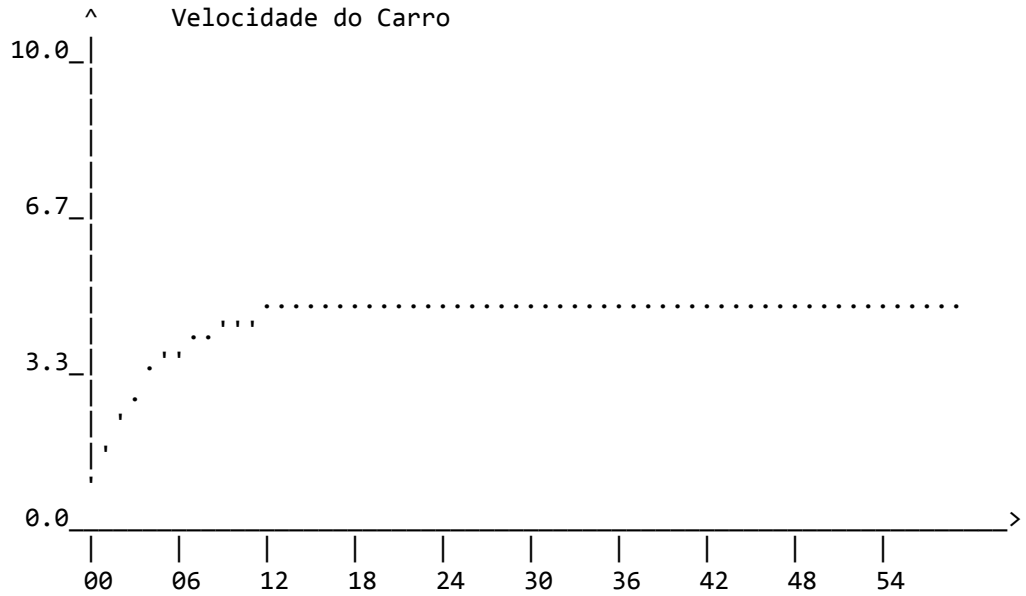
Além disso, nota-se no mesmo diagrama que próximo ao final, se o usuário quiser realizar uma nova operação, há uma reexecução de toda a parte iniciada por "`Qual operacao voce gostaria de fazer?`". Assim, uma boa escolha de implementação será realizar isso **recursivamente**. Ou seja, criar uma função como `void novaOperacao(Sinal *sinalIN)`, que realiza todas as etapas de processamento do sinal de entrada e ao final, se o usuário desejar realizar mais uma operação, chama-se a si mesma passando como parâmetro o sinal que fora processado até então.

3.5 Exemplos

Seguem alguns exemplos de funcionamento do programa com a saída esperada e ressaltando em **vermelho** os dados digitados pelo usuário.

3.5.1 Exemplo 1

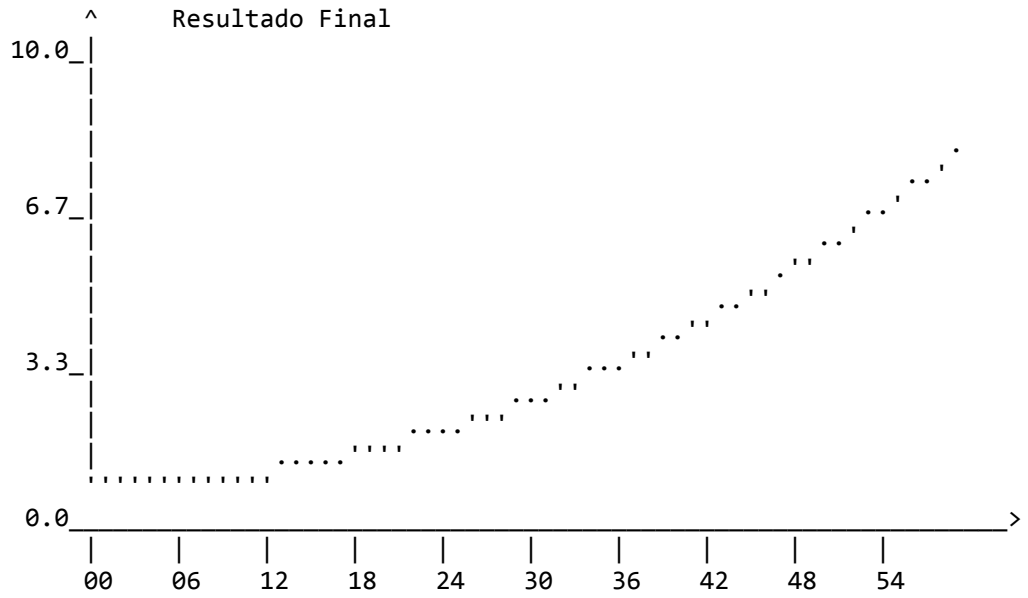
```
        Simulink em C++
Qual simulacao voce gostaria de fazer?
1) Piloto Automatico
2) Sua propria sequencia de operacoes
Escolha: 1

Qual sinal voce gostaria de utilizar como entrada da sua simulacao?
1) 5+3*cos(n*pi/8)
2) constante
3) rampa
Escolha: 2

Qual o valor dessa constante?
C = 5

Qual o ganho do acelerador?
g = 0.2
```



3.5.2 Exemplo 2

```
        Simulink em C++
Qual simulacao voce gostaria de fazer?
1) Piloto Automatico
2) Sua propria sequencia de operacoes
Escolha: 2

Qual sinal voce gostaria de utilizar como entrada da sua simulacao?
1) 5+3*cos(n*pi/8)
2) constante
3) rampa
Escolha: 3

Qual a inclinacao dessa rampa?
a = 0.01

Qual operacao voce gostaria de fazer?
1) Amplificar
2) Somar
3) Derivar
4) Integrar
Escolha: 4

O que voce quer fazer agora?
1) Realizar mais uma operacao no resultado
2) Imprimir o resultado para terminar
Escolha: 1

Qual operacao voce gostaria de fazer?
1) Amplificar
2) Somar
3) Derivar
4) Integrar
Escolha: 1

Qual o ganho dessa amplificação?
g = 0.4
```

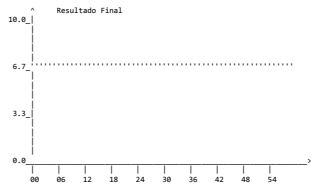
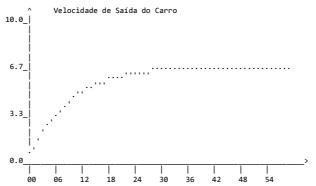
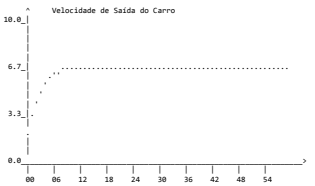
```
O que voce quer fazer agora?
1) Realizar mais uma operacao no resultado
2) Imprimir o resultado para terminar
Escolha: 1

Qual operacao voce gostaria de fazer?
1) Amplificar
2) Somar
3) Derivar
4) Integrar
Escolha: 2

Informe mais um sinal para ser somado.
Qual sinal voce gostaria de utilizar como entrada da sua simulacao?
1) 5+3*cos(n*pi/8)
2) constante
3) rampa
Escolha: 2

Qual o valor dessa constante?
C = 1

O que voce quer fazer agora?
1) Realizar mais uma operacao no resultado
2) Imprimir o resultado para terminar
Escolha: 2
```



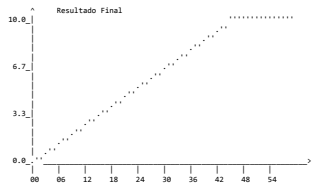
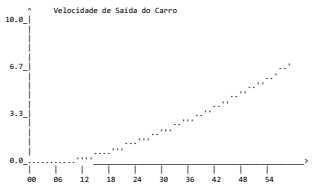
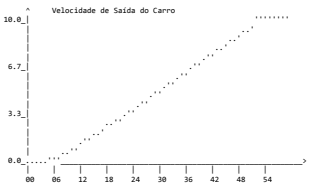
4 Análise dos resultados do piloto automático

Nas matérias de controle será possível aprofundar mais e justificar o porquê desses resultados, porém, fica como sugestão verificar o efeito que o ganho causa na velocidade do carro.

Se, por exemplo, for colocada como referência uma **velocidade constante**, ou seja, deseja-se que o carro mantenha a mesma velocidade em todo instante, o que acontece ao elevarmos o ganho do nosso sistema de controle?

Entrada (referência)	Resultado com ganho 0,1	Resultado com ganho 0,3
		

Além disso, se por outro lado desejarmos uma **velocidade crescente**, qual o papel do ganho na resposta do sistema?

Entrada (referência)	Resultado com ganho 0,03	Resultado com ganho 0,14
		

5 Entrega

O projeto deverá ser entregue até dia **22/10** em um Judge específico, disponível em <https://laboo.pcs.usp.br/ep/> (nos próximos dias vocês receberão um login e uma senha).

As duplas podem ser formadas por alunos de qualquer turma e elas devem ser informadas no e-Disciplinas até dia 08/10. Caso não seja informada a dupla, será considerado que o aluno está fazendo o EP sozinho. **Note que no EP2 deve-se manter a mesma dupla do EP1 (será apenas possível *desfazer* a dupla, mas não formar uma nova).**

> **Atenção:** não copie código de um outro grupo. Qualquer tipo de cópia será considerado plágio e **todos** os alunos dos grupos envolvidos terão **nota 0 no EP**. Portanto, **não envie** o seu código para um colega de outro grupo!

Entregue todos os arquivos, inclusive o `main` e o `menu` (que devem **obrigatoriamente** ficar nos arquivos "main.cpp" e "menu.cpp", respectivamente), em um arquivo comprimido no formato ZIP (outros formatos, como RAR e 7Z, *podem* não ser reconhecidos e acarretar **nota 0**). O nome do arquivo não pode conter espaço, ".", acentos ou ter mais de 11 caracteres. Os códigos fonte não devem ser colocados em pastas. A submissão pode ser feita por qualquer um dos membros da dupla – recomenda-se que os dois submetam.

> **Atenção:** faça a submissão do mesmo arquivo nos 3 problemas (Parte 1, Parte 2 e Parte 3). Isso é necessário por uma limitação do Judge. Caso isso não seja feito, parte do seu EP não será corrigido – impactando a nota.

Siga a convenção de nomes para os arquivos ".h" e ".cpp". O não atendimento disso pode levar a erros de compilação (e, consequentemente, **nota zero**).

Ao submeter os arquivos no Judge será feita **apenas** uma verificação básica buscando evitar erros de compilação devido à erros de digitação do nome das classes e dos métodos públicos. **Note que a nota dada não é a nota final**: neste momento não são executados testes

– o Judge apenas tenta chamar todos os métodos definidos neste documento para todas as classes. Por exemplo, essa verificação é a seguinte para a classe `Sinal`:

```
double sequencia[2] = {1, 2};
Sinal* s = new Sinal(sequencia, 2);
double* d = s->getSequencia();
int c = s->getComprimento();
s->imprimir("a");
delete s;
```

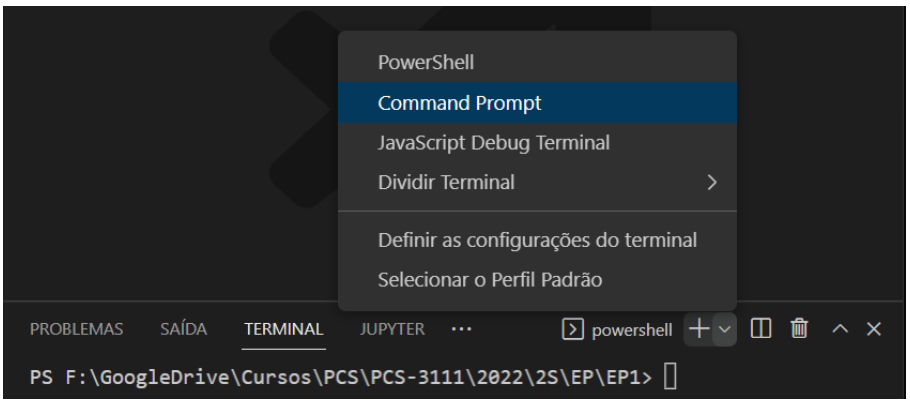
Você pode submeter quantas vezes quiser, sem desconto na nota.

6 Dicas

- Caso o programa esteja travando, execute o programa no modo de depuração. O depurador informará o erro que aconteceu – além de ser possível depurar para descobrir onde o erro acontece!
- Faça `#include` apenas das classes que são usadas naquele arquivo. Por exemplo, se o arquivo .h não usa a classe **X**, mas o .cpp usa essa classe, faça o `include` da classe **X** *apenas* no .cpp. Incluir classes desnecessariamente pode gerar erros de compilação estranhos (por causa de referências circulares).
    - Inclua todas as dependências necessárias. Não dependa de `#includes` feitos por outros arquivos incluídos.
- É muito trabalhoso testar o programa ao executar o `main` *com menus*, já que é necessário informar vários dados para inicializar os registradores e a memória de dados. Para testar o programa faça o `main` chamar uma função de teste que cria objetos com valores interessantes para testar, sem pedir entrada para o usuário. Não se esqueça de remover a função de teste ao entregar a versão final do EP.
    - Uma outra opção para testar é usar o comando:

      `ep < entrada.txt > saida.txt`

      Esse comando executa o programa `ep` usando como entrada do teclado o texto no arquivo `entrada.txt` e coloca em `saída.txt` os textos impressos pelo programa (sem os valores digitados). No caso do Windows, para rodar esse comando você precisa de um prompt de comando (por uma limitação do *PowerShell*). Para fazer isso, clique na seta para baixo do lado do + no terminal e escolha Command Prompt.



- Implemente a solução aos poucos – não deixe para implementar tudo no final.
- Submeta no Judge o código com antecedência para descobrir problemas na sua implementação. É normal acontecerem *RuntimeErrors* e outros tipos de erros no Judge que não aparecem ao executar o programa no Windows. Veja a mensagem de erro do Judge para descobrir em qual classe acontece o problema. Caso você queira testar o projeto em um compilador similar ao do Judge, use o sitehttps://github.com/features/codespaces.
    - Em geral *RuntimeErrors* acontecem porque você não inicializou um atributo que é usado. Por exemplo, caso você não crie um vetor ou não inicialize o atributo quantidade, para controlar o tamanho do vetor, ocorrerá um *RuntimeError*.

Atenção: jamais deixe o código fonte do seu EP público na Internet - algum aluno pode usá-lo e isso será identificado como plágio. Se você usar o GitHub, deixe o repositório privado (adicionando o outro membro da dupla como colaborador).

- Use o Fórum de dúvidas do EP no e-Disciplinas para esclarecer dúvidas no enunciado ou problemas de submissão no Judge.
- **Evite submeter nos últimos minutos do prazo de entrega. É normal o Judge ficar sobrecarregado com várias submissões e demorar para compilar.**