ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
ENGENHARIA CIVIL
DEPARTAMENTO DE ENGENHARIA DE ESTRUTURAS E GEOTÉCNICA

Introdução à programação em ambiente Labview

Autor: Bruno Szpigel Dzialoszynski

Orientador: Prof. Valério S. Almeida

São Paulo, 2015.

1. INTRODUÇÃO ... 4
2. LABVIEW E O MONITORAMENTO DE ESTRUTURAS EM TEMPO REAL ... 4
3. CONCEITOS BÁSICOS E LÓGICA DE PROGRAMAÇÃO ... 5
3.1. VIRTUAL INSTRUMENT, A UNIDADE FUNDAMENTAL EM LABVIEW ... 5
3.2. PAINEL FRONTAL E DIAGRAMA DE BLOCOS – CONTROLES, INDICADORES, FUNÇÕES E CABOS . 5
3.3. A DINÂMICA DE FLUXO NO DIAGRAMA DE BLOCOS ... 8
3.3.1. A LÓGICA DE FLUXO ... 8
3.3.2. CONTROLANDO FLUXOS - DEPENDÊNCIA DE DADOS NATURAL E ARTIFICIAL ... 10
4. ESTRUTURAS BÁSICAS E SEUS ELEMENTOS E APRESENTAÇÃO GRÁFICA DE DADOS ... 13
4.1. ESTRUTURAS DE CASOS ... 13
4.1.1. ARRAYS – AGRUPANDO DADOS ... 15
4.2. ITERAÇÕES – ‘FOR LOOPS’, ‘WHILE LOOPS’ E SEUS ELEMENTOS ... 17
4.2.1. ESTRUTURA DOS LOOPS ... 17
4.2.2. AUTO-INDEXING ... 19
4.2.3. SHIFT REGISTERS ... 20
4.3. REUTILIZANDO ROTINAS – SUBVIs ... 21
4.3.1. CRIAÇÃO E UTILIZAÇÃO DE SUBVIS ... 21
4.3.2. EDIÇAO DE SUBVIS ... 23
4.4. CONTROLE DO TEMPO NOS VIs – WAIT E WAIT UNTIL NEXT ms MULTIPLE ... 24
4.5. REPRESENTANDO RESULTADOS GRAFICAMENTE – CHARTS E GRAPHS ... 26
4.5.1. CHARTS ... 26
4.5.2. GRAPHS ... 29
4.6. ESTRUTURA DE ORDENAMENTO – FLAT SEQUENCE STRUCTURE ... 33
4.7. EXEMPLO DE CONSOLIDAÇÃO – VELOCIDADE DO VENTO ... 34
5. OPERAÇÕES BÁSICAS FILE INPUT-OUTPUT, EXECUTÁVEIS, ERROS, PATHS E SMTP ... 38
5.1. REFERENCIANDO ARQUIVOS – FILE PATHS E REFNUMS ... 38
5.1.1. PATHS ABSOLUTOS E RELATIVOS ... 38
5.1.2. FILE REFNUMS ... 40
5.2. COMUNICANDO E TRATANDO FALHAS – ERRORS ... 41
5.2.1. TRATAMENTO AUTOMÁTICO OU CUSTOMIZADO ... 41
5.2.2. A ESTRUTURA DO ERRO ... 42
5.2.3. FORMAS DE LIDAR COM ERROS ... 43
5.3. FILE INPUT – OUTPUT PARA ARQUIVOS DE TEXTO ... 43
5.3.1. CRIANDO, ABRINDO, SUBSTITUÍNDO, FECHANDO E DELETANDO ARQUIVOS DE TEXTO ... 43

5.3.2. EDIÇÃO DE ARQUIVOS DE TEXTO 44
5.3.3. LEITURA E PROCESSAMENTO DE ARQUIVOS DE TEXTO 46
5.4. EMPREGANDO EXECUTÁVEIS 48
5.5. SIMPLE MAIL TRANFER PROTOCOLS 48
5.6. EXEMPLO DE CONSOLIDAÇÃO – CARAC. GEOMÉTRICAS DE UMA POLIGONAL PLANA 50
5.6.1. INICIALIZAÇÕES 52
5.6.2. ESCRITA DOS DADOS DE ENTRADA 55
5.6.3. CHAMADA DO EXECUTÁVEL 57
5.6.4. LEITURA DO ARQUIVO GERADO E APRESENTAÇÃO DE RESULTADOS 57
6. BIBLIOGRAFIA 60

1. INTRODUÇÃO

O estudo da programação na plataforma LabVIEW, bem como de suas características, nexos e aplicações práticas na engenharia de estruturas seguiu um método bem delimitado para o primeiro semestre do projeto.

Inicialmente estudou-se a apostila básica de Nery (2013) para obter uma primeira visão da interface LabVIEW, bem como o funcionamento básico do software utilizado na programação. Nesse sentido, buscou-se conhecimento sobre o uso e acesso aos diversos menus e opções da plataforma.

Essa apostila contempla os seguintes tópicos, em ordem de apresentação:

1. Conceitos Básicos e Lógica de Programação
2. Estruturas Básicas e Seus Elementos, e Apresentação Gráfica de Dados
3. Operações de File Input/Output, Executáveis, Erros, Paths e SMTP
4. Programação com Aquisição de Dados em Tempo Real

Para cada macro área - excetuando-se a primeira, dada sua natureza essencialmente conceitual e introdutória - foi proposto um programa cuja criação exigia a aplicação prática de todos os conceitos englobados. Assim, com o desafio e objetivo final da execução do programa proposto, o estudo teórico de cada macro área foi realizado de maneira flexível e dinâmica, conforme problemas e necessidades práticas surgiam na concepção do programa.

O aprendizado, dessa maneira, foi embasado na bibliografia supracitada e no apoio do professor orientador. Adicionalmente, para cada novo conceito estudado, o conteúdo foi sintetizado em texto e em programas exemplo, validados pelo orientador e apresentados nesse relatório.

2. LABVIEW E O MONITORAMENTO DE ESTRUTURAS EM TEMPO REAL

O LabVIEW é uma plataforma de programação na linguagem G, isso é, gráfica. Com efeito, utiliza-se de ícones e conexões gráficas ao invés de textos para estabelecer os fluxos, operações e relações lógicas do programa criado.

No caso particular do LabVIEW, a programação é baseada em fluxos de dados. De fato, de maneira intuitiva, estabelecem-se entradas de dados, que fluem pelo programa, representados por fios, processam-se e transformam ao passar por funções, representadas por caixas e finalmente saem e apresentam-se aos usuários.

A utilização dessa linguagem gráfica baseada em fluxos traz consigo, no entanto, uma série de vantagens e desvantagens.

A principal desvantagem frente à programação em linhas de comando é que, como a sequencialidade das ações a serem realizadas não é escrita linha a linha, é necessário um cuidado maior com a ordem das operações, garantindo fluxo adequado e relações de dependência. Outra desvantagem é que, como dados são representados por fios, as variáveis são tratadas de maneira diferente do usual em outras linguagens, por exemplo, com sua declaração, explícita ou implícita.

Por outro lado, especialmente para as aplicações de engenharia e monitoramento, a ferramenta LabVIEW apresenta inegáveis vantagens. Em primeiro lugar é saliente o quão intuitiva é a programação gráfica, garantindo rápido aprendizado. Em segundo lugar, como a própria estrutura da plataforma é baseada em fluxos de dados, a mesma apresenta-se altamente oportuna para a aquisição, tratamento e apresentação de dados. Olhado um código de LabVIEW, por sua estrutura intrínseca, há uma maior facilidade em identificar o fluxo dos dados, e como está ocorrendo seu processamento.

No entanto, o grande diferencial da plataforma LabVIEW é a de se tratar de um sistema com capacidades *multitasking* e *multithreaded*. Segundo a referida página de suporte "*Multitasking* refere-se à capacidade do sistema operante de rapidamente alternar entre tarefas, dando a aparência de execução simultânea"[1] (National Instruments, 2013, tradução livre). *Multithreading*, por sua vez, seria uma extensão desse conceito, para a possibilidade de executar várias aplicações em paralelo, dividindo automaticamente o tempo de processamento. Essa capacidade pode ser altamente vantajosa para flexibilizar os códigos de monitoramento, bem como operar em altas frequências de aquisição e exigência de CPU. De fato, o próprio manual do LabVIEW cita como vantagem dessa característica um exemplo de programa de monitoramento, com as aplicações de aquisição, controle de instrumentos e interface do usuário operando simultaneamente.

Outro diferencial do LabVIEW é sua flexibilidade e facilidade para estabelecer interfaces com diversos hardwares, por exemplo de aquisição. De fato, a própria fabricante do LabVIEW oferece hardwares de aquisição cuja utilização conjunta é altamente intuitiva, na maior parte dos casos.

Por fim, instalados os devidos pacotes de funcionalidades a plataforma é capaz de realizar aquisição, processamento e indicação de dados em tempo real. Isso significa que o sistema deve ser capaz de realizar tarefas de forma rápida, precisa e confiável.

Esse conjunto de vantagens mostra o quão oportuna é a plataforma LabVIEW para a aplicação de monitoramento de estruturas. Em primeiro lugar, a interface intuitiva pode torna-la acessível a uma maior quantidade de profissionais do ramo de engenharia civil, nem sempre familiares com linguagens de programação. Em segundo lugar, as características da interface, unidas às capacidades '*multitasking*' e '*multithreaded*' do LabVIEW podem significar, muitas vezes, simplificação do código, poupando muito trabalho, em comparação à programação convencional, por linhas de comando. Finalmente, o sistema pode atender plenamente à maior parte dos requisitos de um sistema de monitoramento – rapidez, precisão, confiabilidade e interface com hardwares de aquisição.

3. CONCEITOS BÁSICOS E LÓGICA DE PROGRAMAÇÃO

A plataforma LabVIEW possuí alguns conceitos básicos específicos e muitas vezes diferentes de outras formas de programação. Tais conceitos são essenciais para a elaboração de rotinas e aplicações.

3.1. VIRTUAL INSTRUMENT, A UNIDADE FUNDAMENTAL EM LABVIEW

Qualquer unidade de funcionamento completo, em LabVIEW, é denominado como um Virtual Instrument (VI), conforme Drummond (1998). Isso é, qualquer conjunto criado que, recebendo entradas estipuladas, seja capaz de realizar operações, por si só, operar sobre elas, entregando as saídas definidas pode ser considerado como um VI. Isso é, enquadram-se nessa categoria funções, rotinas e programas. Nesse sentido, um VI pode, em si, conter outros Vis, nesse caso denominados SubVIs.

Assim, o desenvolvimento de projetos em LabVIEW envolve principalmente a elaboração de VIs de diversas naturezas.. Como um VI pode conter outros em seu código, diz-se haverem, num projeto, níveis de VIs. Isso é, aqueles que não dependem de outros para funcionarem são ditos de nível mais baixo. Conforme um VI passa a utilizar em seu código outros, seu nível sobe, progressivamente.

3.2. PAINEL FRONTAL E DIAGRAMA DE BLOCOS – CONTROLES, INDICADORES, FUNÇÕES E CABOS

[1] - Multitasking refers to the ability of the operating system to quickly switch between tasks, giving the appearance of simultaneous execution of those tasks.

A programação em LabVIEW se organiza primordialmente em dois planos. O Painel Frontal (*Front Panel*) e o Diagrama de Blocos (*Block Diagram*). Cada um se apresenta em uma janela separada e possuí um significado diverso.

O Painel Frontal é a interface do usuário com o programa criado pelo programador. Isso é, nele o programador deve dispor os controles e os indicadores, além das decorações e informações adicionais que quiser transmitir ao usuário, como explica Nery (2013).

Os controles tratam-se das entradas do VI. Isso é, são variáveis controláveis pelo usuário que, uma vez definidas, fluirão pelo código, resultando em saídas.

Os indicadores, por sua vez, tratam-se das saídas. Em geral não são controláveis pelo usuário e resultam do processamento de dados, apresentando resultados.

Tanto indicadores como controles podem ser, evidentemente, numéricos, de *strings*, de *arrays*, de matrizes, de clusters, booleanos, etc. O tipo de variável, assim, depende da natureza dos dados representados.

Na figura abaixo tem-se o exemplo de um painel frontal.

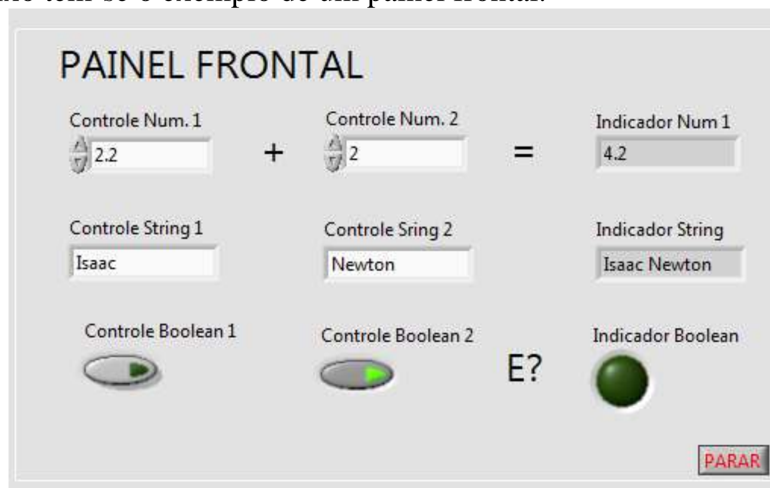


Figura 1 - Exemplo de Painel Frontal

Nele notam-se 7 controles e 3 indicadores. Dentre os controles têm-se 2 numéricos, 3 de texto (*strings*) e 3 booleanos, sendo um deles o botão 'PARAR', que controla o *loop* geral do programa, tratado mais adiante, em outra seção.

As operações realizadas são intuitivas. O indicador numérico apresenta a soma dos valores dispostos nos controles análogos. O indicador de texto apresenta a concatenação dos textos nos comandos, com um espaço entre eles. O indicador booleano verifica se ambos os botões estão pressionados.

O Diagrama de Blocos, em seu turno, é o código do programa propriamente dito. Em uma analogia presente em Drummond (1998), seria como o 'verso' do painel frontal.

Nele aparecem inicialmente diversas 'caixas'. Algumas são referentes aos comandos dispostos no painel, dos quais saem fios. Essas podem ser identificadas por possuírem terminais de saída, de seu lado direito. Outras referem-se aos indicadores, nas quais entram fios. Ou seja, são identificadas por terminais de chegada, do seu lado esquerdo.

A partir disso, o programador pode adicionar mais caixas, com entradas e saídas de fios correspondentes a funções disponíveis ou personalizadas. Dependendo do tipo de função, assim, o número de terminais de entrada e saída varia, bem como as operações realizadas.

Sendo os fios correspondentes aos fluxos de informações e dados, o programador conecta convenientemente os cabos, que partem dos controles, ou de constantes pré-definidas, transformam-se

nas funções e saem nos indicadores. A adição de funções, por sua vez se efetua por um menu intuitivo, no qual constam tanto funções padrão, oferecidas no pacote da plataforma, ou em outras expansões, como funções customizadas, criadas pelo próprio programador.

Abaixo consta o diagrama de blocos do mesmo programa mostrado anteriormente.

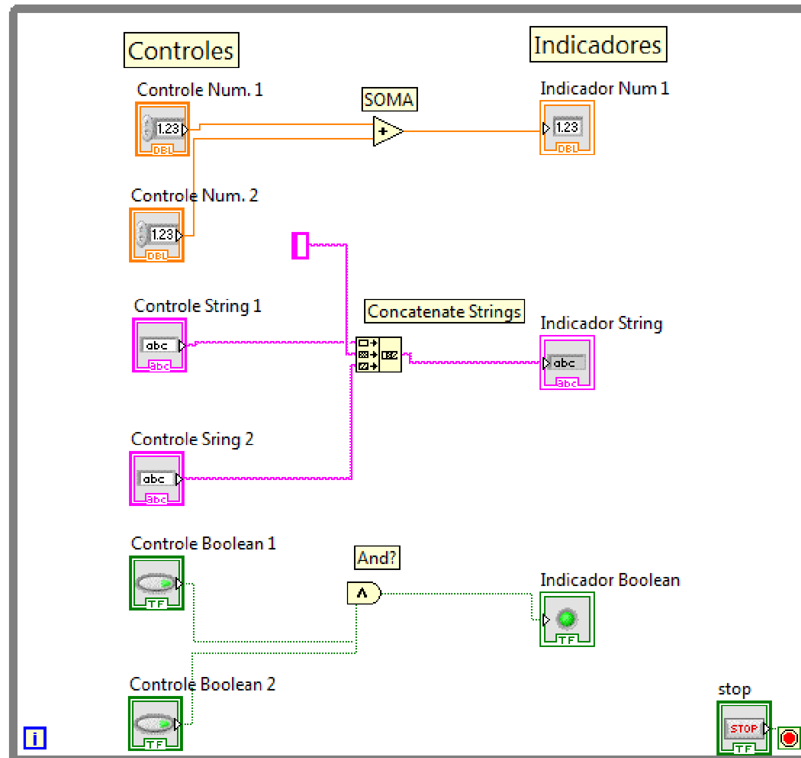


Figura 2 - Exemplo de Diagrama de Blocos

Já de início nota-se a presença de todos os controles – com terminais de saída – e indicadores – com terminais de chegada – dispostos no painel frontal. Outro detalhe saliente são os cabos. Como se percebe, cada tipo de dado possuí uma cor característica. Isso é, valores numéricos tipo *float* são cabos laranja e retilíneos, valores de texto, rosas e ondulados, e valores booleanos verdes e retilíneos. Assim, a cor e textura dos cabos define o tipo de dado que por eles fluí, havendo ainda muitos outros tipos, além dos citados. Como se verá adiante, na seção de *arrays*, a espessura do cabo dá ainda informações sobre a dimensão dos dados que por eles passam.

Finalmente percebe-se o uso de 3 funções.

Utilizou-se a função '*Add*'. Como se vê, essa caixa possuí duas entradas e uma saída. Assim, efetua a soma das entradas, que devem ser numéricas e devolve esse valor em sua saída, também numérica.

A função '*And*', por sua vez, recebe duas entradas booleanas. Caso ambas indiquem '*TRUE*', então a função devolve '*TRUE*'. Caso contrário, devolve '*FALSE*'.

A função '*Concatenate Strings*', no caso, utilizou 3 entradas e uma saída. No entanto, como diversas funções de LabVIEW, esse número não é fixo. Poder-se-iam, por exemplo, ter concatenado 4, ou apenas 2 strings. Para modificar esse número de entradas, bastaria arrastar para cima ou para baixo a base da caixa correspondente.

Há ainda o caso, como se verá adiante, em que não é necessário conectar todos os terminais de entrada – utilizando-se os valores default da função – ou saída – ignorando esses valores.

Outro fator importante nessa função é que se utilizou uma constante. No caso, utilizou-se a constante de texto ‘ ‘ (espaço) como *string* intermediário. Como se percebe, as constantes funcionam de maneira semelhante aos controles, mas não podem ser modificadas ou vistas pelo usuário.

Esses são exemplos de funções básicas e de uso muito comum em diversos tipos de código. Conforme as rotinas tornam-se mais complexas, também as funções aplicadas tornam-se mais diversas e específicas.

3.3.A DINÂMICA DE FLUXO NO DIAGRAMA DE BLOCOS

Uma das maiores vantagens da programação em LabVIEW representa também um dos maiores desafios para seus programadores. Diferente da linguagem por linhas de comando, com sequenciamento claro e explícito de ações, a linguagem gráfica da plataforma em questão muitas vezes não explicita claramente a ordem de execução de suas funções.

A execução de um código em LabVIEW, no entanto, possuí uma dinâmica que segue uma lógica bem definida. Essa lógica pode, inclusive, ser controlada de maneira mais ou menos rigorosa pelo programador, dependendo de suas necessidades.

Em aplicativos de monitoramento em tempo real, por exemplo, a ordem de execução de operações, por vezes, tem de ser rigorosamente controlada. Nesses casos opera-se no limite das capacidades de memória, CPU e velocidade das máquinas, sendo necessário estabelecer prioridades e ordenamentos.

Para a maioria das demais aplicações, por outro lado, esse controle pode ser mais brando. É, no entanto, sempre necessário compreender a dinâmica de fluxo de um código para programar de maneira eficiente.

3.3.1. A LÓGICA DE FLUXO

O princípio fundamental que deve ser considerado na dinâmica do fluxo no diagrama de blocos de LabVIEW, segundo a correspondente página online de suporte é que um nó – função, estrutura ou SubVI- do código executa apenas quando receber dados em todas as entradas conectadas. (National Instruments, 2013). Com sua execução, o nó libera dados por seus terminais de saída. Assim, é o fluxo de dados que determina a ordem de execução das funcionalidades de um VI.

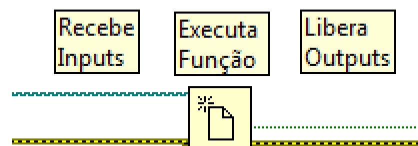


Figura 3 - Exemplo de Nó

É importante notar, no entanto, que nem todos os terminais de entrada ou saída de uma função tem que estar conectados. Quanto aos terminais de saída, a conexão de todos eles sempre é opcional. Quanto aos terminais de entrada, podem existir terminais obrigatórios ou opcionais, ou ainda terminais de entrada que, quando não conectados, assumem valores padrão (*default*). Entretanto, sempre que um terminal estiver conectado, a execução do código aguarda para prosseguir até que fluam dados pelo referido terminal.

Tal principio – recebimento de todos os inputs ativa função, que gera outputs – pode ser simples, mas ocasiona uma série de consequências.

A primeira consequência é que uma função que recebe como dado de entrada os dados de saída de outra função obrigatoriamente executará após a primeira. Isso ocorre pois a segunda função necessita dados liberados apenas após a execução da primeira. Nota-se, assim, que a posição em si de um nó no diagrama de blocos (acima, abaixo, à direita, à esquerda) nada tem a ver com sua ordem de execução.

No exemplo a seguir, os nós mantém-se na mesma posição nas duas operações. Entretanto, a mudança na ordem de conexão dos cabos modifica completamente o resultado da execução.

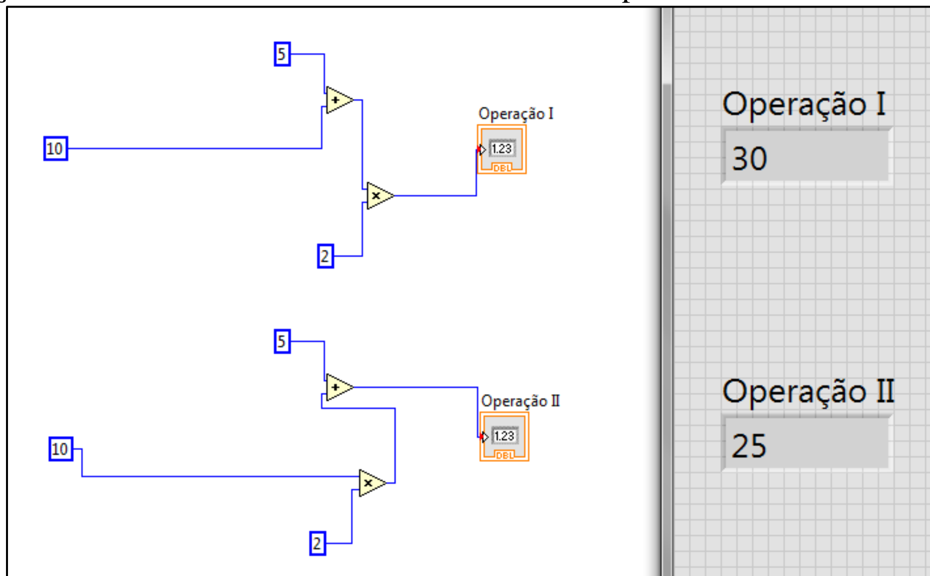


Figura 4 - Ordem de Execução

Quando uma função necessita de dados de uma outra para executar, diz-se que estabeleceu-se uma 'Dependência de Dados' (*Data Dependency*) (National Instruments, 2013.). Como se verá a seguir, essa dependência pode ser natural, ou criada artificialmente para conveniência do programador.

A segunda consequência da lógica de fluxo do LabVIEW é que o diagrama de blocos pode ter operações simultâneas. O LabVIEW é um sistema que pode realizar múltiplas tarefas praticamente de forma simultânea, alternando entre elas rapidamente. Diz-se, pois, tratar-se de um sistema *multitasking* e *multithreaded*, como desenvolvido na seção 3.1..

Como se verá adiante, essa alternância entre tarefas pode ser, inclusive, controlada quando necessário, como é o caso do monitoramento em tempo real. Entretanto, na maioria das aplicações, o a própria plataforma realiza essa alternância de forma a realizar as operações de forma simultânea.

Assim, se um número qualquer de operações possuir dados para sua execução num mesmo instante (note que, logicamente, não deve existir dependência de dados entre elas), todas executarão simultaneamente, em aparência. No diagrama de blocos a seguir, por exemplo, dois loops (ver ítem 2.3.3.) processam a apresentam dados simultaneamente e independentemente.

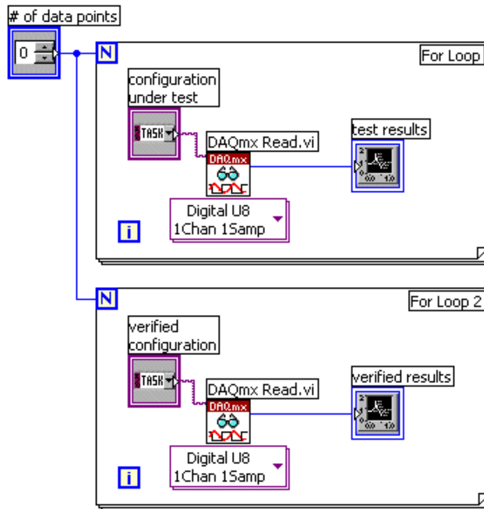


Figura 5 - Loops que Operam Simultaneamente

A simultaneidade na execução de operações, assim, dá caráter mais dinâmico e flexível ao código. Uma vez compreendida a dinâmica de fluxo no diagrama de blocos, existem diversas maneiras de controla-la. A seguir constam alguns artifícios que podem ser utilizados para que os dados fluam como desejado no diagrama.

3.3.2. CONTROLANDO FLUXOS - DEPENDÊNCIA DE DADOS NATURAL E ARTIFICIAL

Como já desenvolvido, quando uma função recebe dados de outra função, diz-se que a primeira tem dependência de dados frente a segunda. Muitas vezes, a dependência de dados é natural, isso é, de fato, sob a ótica funcional, a execução de uma função é impossível sem que essa recebe os dados de saída de uma outra função. Nesse caso reconhece-se uma dependência natural de dados. O diagrama de blocos a seguir ilustra esse tipo de dependência.

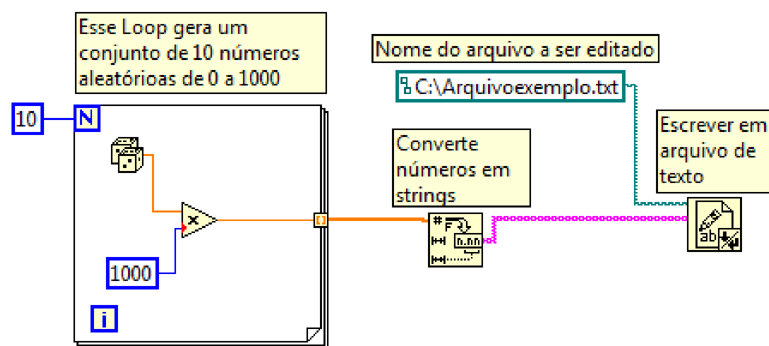


Figura 6 - Dependência natural de Dados

Esse exemplo utiliza algumas estruturas e funções que serão discutidas mais a frente. No entanto, seu funcionamento mais geral pode ser facilmente compreendido e ilustra uma dependência natural de dados. No caso, o programa gera, no loop da esquerda, um dado randômico, de 0 a 1000 a cada repetição. Foi programado para repetir a execução 10 vezes, então gerará um conjunto de 10 números randômicos. A função na parte central do diagrama recebe o conjunto de 10 números gerados pelo loop

(note que o cabo, ainda que laranja é mais espesso). Dessa maneira, só poderá executar quando o loop tiver realizado suas 10 repetições, gerando o conjunto de números, recebido pela segunda função. Essa função converterá o conjunto de dados numéricos em um conjunto de dados de texto. A última função, por sua vez, recebe o conjunto de dados de texto gerado e escreve-os em um arquivo. Com efeito, só poderá executar quando as duas outras partes do código tiverem sido encerradas.

Assim, nota-se que a função de conversão tem dependência natural frente ao *loop*. Isso é, sua execução seria impossível caso o cabo laranja fosse desconectado. A função de escrever em arquivo, por sua vez, tem dependência natural à função de conversão e, indiretamente, ao *loop*.

Muitas vezes, entretanto, mesmo sem haver dependência natural entre funções, deseja-se que as mesmas sejam executadas em uma ordem definida. Esse tipo de ocorrência é especialmente comum para funções do tipo *Input/Output* isso é, que atuam em interface com sistemas externos ao código.

Isso ocorre porque algumas funções, mesmo sem apresentar nenhum tipo de dependência natural entre si, tem funcionamento tal que a diferente ordenação de sua execução faz com que o efeito final no funcionamento do programa seja completamente diferente. Nesses casos, surge uma ambiguidade no código, pois a ordem de execução é imprevisível e, portanto, o funcionamento do código também o será.

No caso abaixo, por exemplo, está ilustrada uma ambiguidade muito comum em programas que trabalham com funções *Input/Output*.

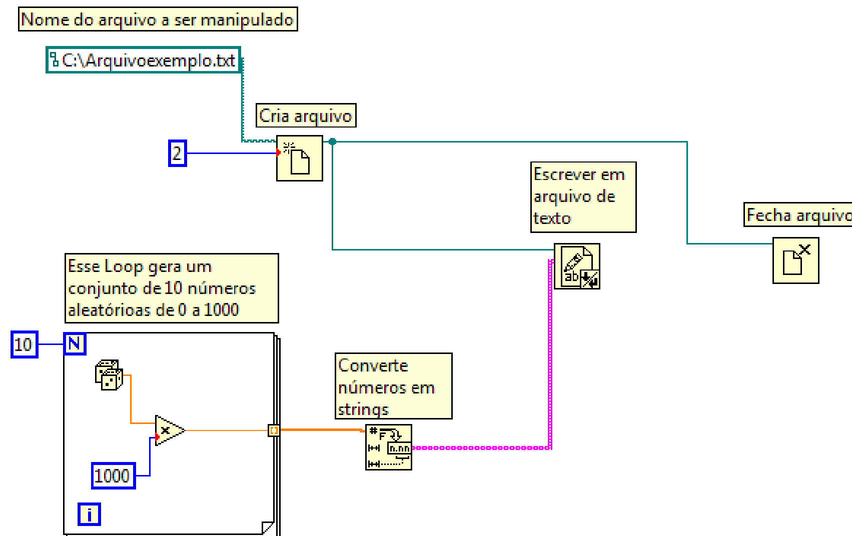


Figura 7 - Código Ambíguo

Esse diagrama de blocos usa um dispositivo idêntico ao do exemplo anterior para gerar um conjunto de dados. Na parte superior do diagrama, entretanto notam-se novidades. Recebendo um endereço, uma função cria um arquivo na localidade especificada. Essa função, por sua vez, libera um endereço relativo do arquivo criado, representado pelo cabo verde escuro fino. Nota-se que ambas as funções que vem em seguida, isso é, de escrever ou de fechar o arquivo dependem desse dado. Dessa maneira, não existe nenhuma relação de dependência de dados entre essas duas funções. Com efeito, não há como prever de forma precisa qual das funções executará primeiro.

O resultado da execução global do código, no entanto, é drasticamente diferente dependendo da ordem de execução das funções. Se a função de escrita executar primeiro, tem-se um arquivo com dados registrados. Se a função de fechar executar primeiro têm-se uma arquivo em branco. O funcionamento do código, portanto, tornar-se-ia imprevisível dada a existência da ambiguidade.

Fica latente, assim, a necessidade de empregar artifícios para estabelecer uma relação de dependência de dados entre funções desse tipo. Com efeito, quase sempre que se desejar ordenar a execução de funções, é necessário estabelecer dependência entre elas. Nos casos em que uma dependência é criada por conveniência do programador, diz-se tratar-se de uma dependência artificial de dados. (National Instruments, 2013.)

Existem algumas maneiras de criar dependência artificial de dados, cada uma com suas próprias vantagens e propriedades particulares.

A primeira maneira de criar dependência artificial de dados é por meio dos parâmetros '*flow-through*'. Dentre as principais vantagens de seu emprego constam a rapidez e facilidade de aplicação. Como o próprio nome indica, trata-se de parâmetros que são recebidos e entregues por certas funções sem sofrerem modificações. Isso é, são terminais que recebem um dado, entregando-o por um terminal de saída sem modifica-lo. Em geral os parâmetros *flow-through* são variáveis do tipo cluster de erro, ou *refnums* (discutidos adiante), o que é coerente com a maior necessidade de dependências artificiais em funções tipo *Input/Output*. Usualmente os terminais de entrada e saída que integram o mecanismo *flow-through* são nomeados por '*in*' e '*out*', respectivamente – por exemplo *refnum in* e *refnum out*.

A seguir, consta um diagrama de blocos no qual a ambiguidade do programa apresentado logo acime foi solucionada pelo emprego de parâmetros *flow-through*.

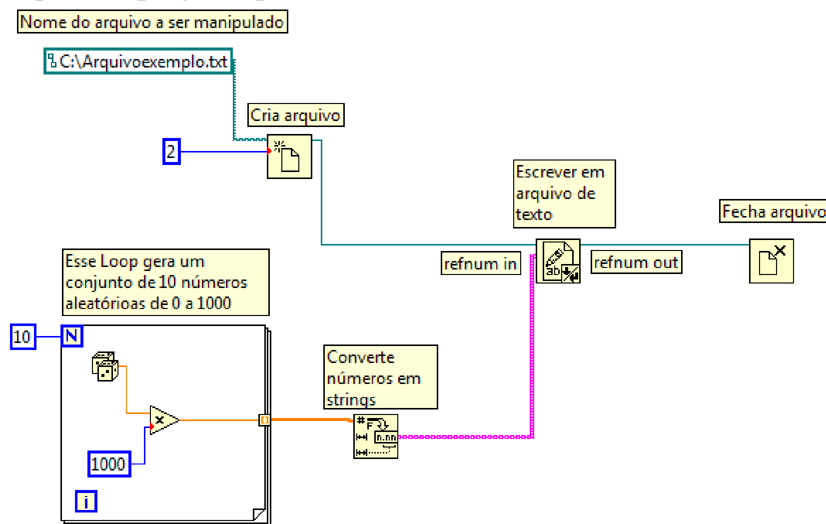


Figura 8 - Ambiguidade Solucionada com Flow-Through

Nesse caso, o cabo que carrega o *refnum* com o 'endereço' do arquivo a ser manipulado não foi conectado diretamente às funções de edição e fechamento do arquivo. Para garantir a dependência artificial, foi conectado ao terminal '*refnum in*' da função de edição e o terminal '*refnum out*' dessa mesma função foi ligado ao terminal da função de fechamento do arquivo.

É importante salientar que qualquer outro tipo de variável, que não os *refnums* poderia ter sido usada como parâmetro *flow-through*, dada a existência dos terminais adequados. Poderiam, por exemplo, ter sido usado os dados de erro como *flow-through*.

Os clusters de erro e refnums nem sempre são parâmetros '*flow through*'. Como será desenvolvido, por vezes podem ter seu valor modificado.

A segunda maneira de criar dependência artificial de dados é fazê-lo de forma personalizada. Dentre as vantagens desse método, consta como principal a flexibilidade. Muitas vezes, no entanto, a implantação pode ser mais complexa e demorada. Para aplicar uma dependência artificial de dados, assim, basta utilizar a lógica de fluxo no diagrama de blocos de maneira criativa.

A seguir está apresentado um exemplo onde criou-se uma dependência artificial de dados customizada. Nesse caso, claramente o uso de parâmetro *flow-through* é mais vantajoso. Em outros casos, no entanto, nem sempre esse é o caso.

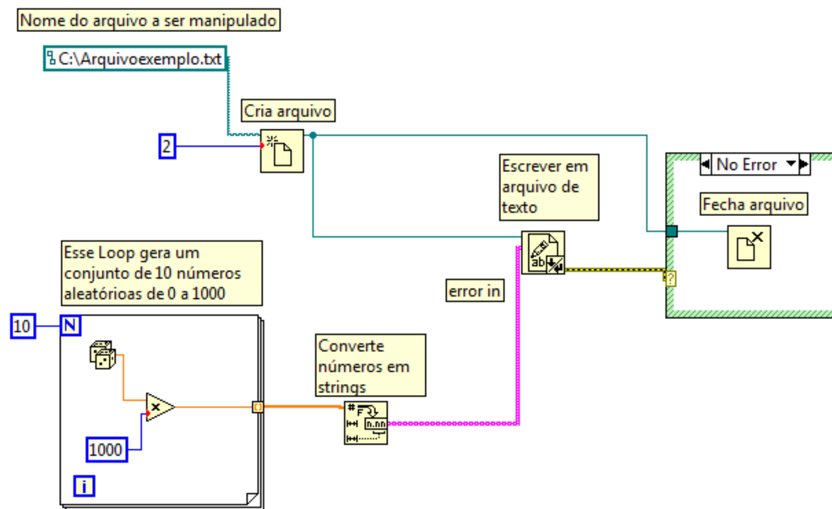


Figura 9 - Ambiguidade Solucionada com Estrutura de Casos

Nesse caso utilizou-se uma estrutura de casos para estabelecer a dependência. Conectando o cluster de erro à condição da estrutura, garante-se que ela só opere quando receber dados. Como a função de fechamento está contida na estrutura o mesmo se aplica à mesma. Nesse ponto pode-se notar uma possível vantagem desse método, na medida em que além de estabelecer a dependência de dados, é possível condicionar o programa a operar de maneira diferente caso ocorra ou não erro.

Há uma ultima maneira de ordenar a execução de funções sem dependência natural. Nesse caso não se cria uma dependência artificial. Em lugar disso é utilizada uma 'estrutura de sequência', chamada '*flat sequence structure*'. Essa estrutura é discutida adiante, no item 7.7..

4. ESTRUTURAS BÁSICAS E SEUS ELEMENTOS E APRESENTAÇÃO GRÁFICA DE DADOS

Existem algumas estruturas gerais comuns a quase todos os aplicativos desenvolvidos em LabVIEW. Muitas delas estão presentes em outras plataformas de programação, como *loops* e estruturas de casos. No entanto, quase sempre sua apresentação e uso é diferente na programação em LabVIEW.

4.1.ESTRUTURAS DE CASOS

A estrutura de casos é uma estrutura essencial para a programação em LabVIEW. A partir dela é possível estabelecer uma condição segundo a qual diferentes ações serão realizadas. Seu correspondente análogo na programação em C serio o '*if*'.

Como as demais funcionalidade de LabVIEW, a aplicação da estrutura de casos é consideravelmente intuitiva. A estrutura possuí basicamente 3 elementos. Tem-se um retângulo dentro em cuja a borda superior existe um cursor e em cuja borda lateral existe um ícone com uma interrogação.

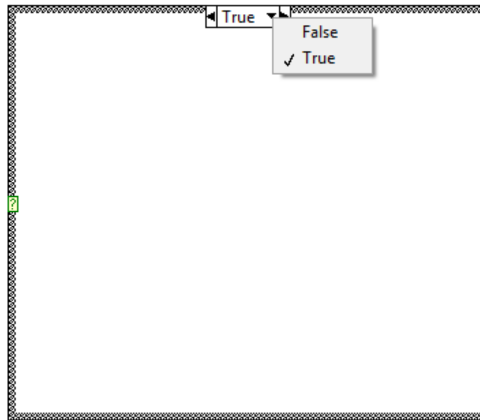


Figura 10 - Estrutura de Casos

O cursor na borda superior serve para a seleção da página em questão, isso é escolher qual dos casos se deseja visualizar. Ao ícone de interrogação, por sua vez, deve ser conectado o dado ao qual a condição para a execução dos diversos casos está atrelada. No retângulo, finalmente, para cada página, devem ser posicionadas as ações correspondentes (funções, constantes, controles indicadores, conexões, etc.) ao respectivo caso.

O que definirá o número de páginas possíveis é a variável conectada ao terminal de interrogação. Por exemplo, se for conectado uma variável booleana, haverá a página '*TRUE*', acionada quando o valor da variável for esse, e a página '*FALSE*'. Caso um valor numérico inteiro seja associado ao terminal de interrogação, no entanto, mais casos serão possíveis.

É importante notar que, se por um lado, para cada caso as ações que ocorrem dentro da caixa variam, por outro lado os cabos que chegam e saem da caixa são os mesmos. Esses pontos de entrada e saída de cabos na estrutura são representados por pequenos quadrados.

Os dados que entram na caixa podem simplesmente ser ignorados, se forem inúteis para o caso. No entanto, note que, para dados que saem da caixa, para todos os casos deve haver um valor associado, do contrário o programa fica com um cabo sem dado associado ocasionando erro. O LabVIEW indica essa irregularidade representando o quadrado vazio.

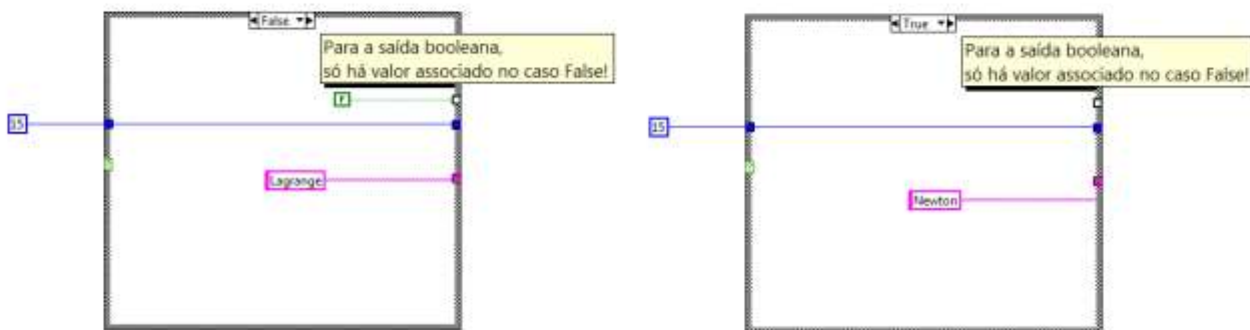


Figura 11 - Saída de Estrutura de Casos

Para explorar a estrutura de casos, criou-se um programa que recebe dois fatores. O usuário escolhe ainda uma operação dentre adição, subtração, multiplicação e divisão e o programa a efetua. Ou seja, os fatores entram na estrutura de casos, cujas páginas correspondem às diferentes operações, e o resultado sai da estrutura.

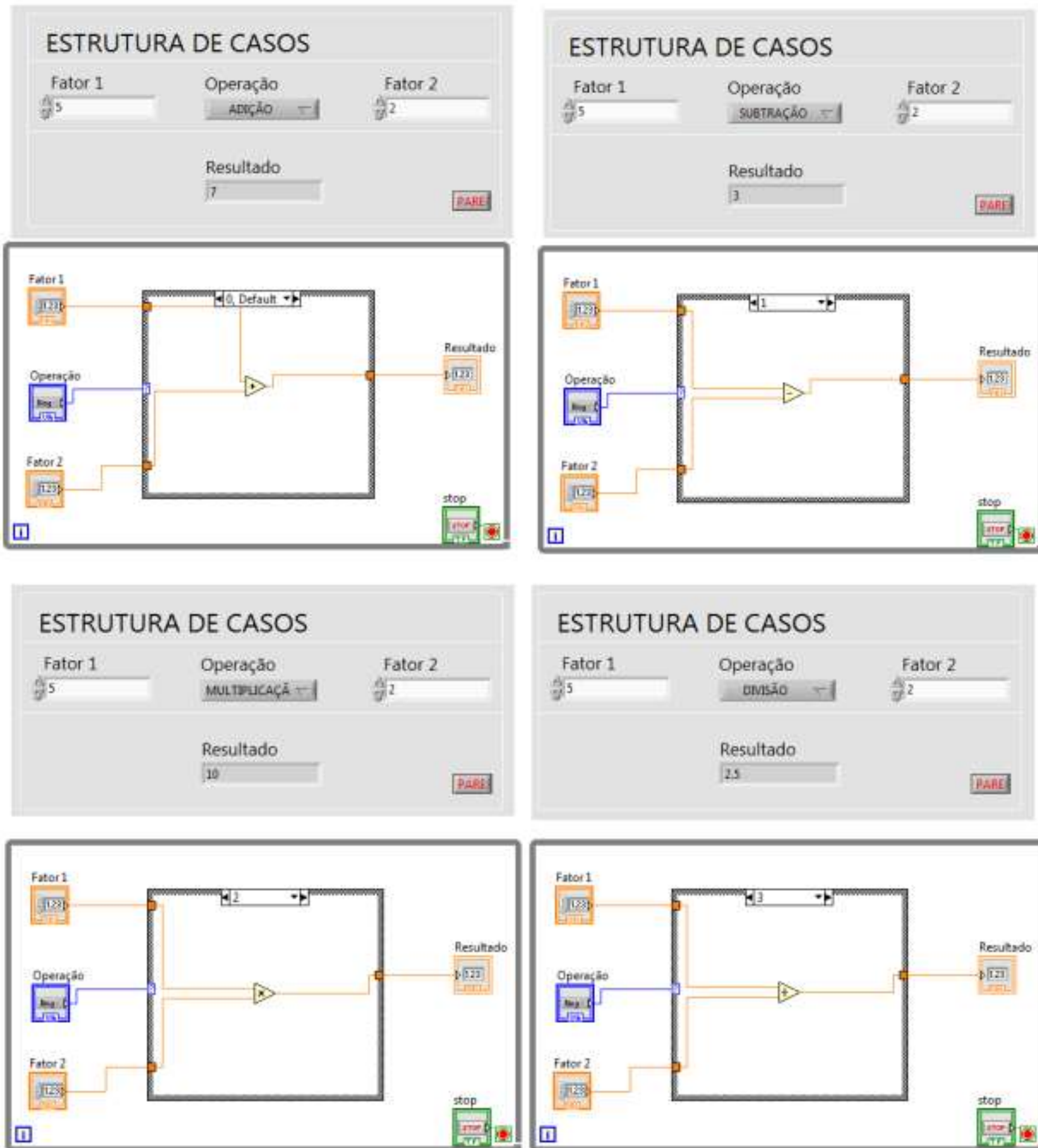


Figura 12 - Exemplo de Aplicação da Estrutura de Casos: Calculadora Símples

Como se percebe, o programa é consideravelmente simples. O controle ‘operação’ controla os 4 casos possíveis, as entradas são os fatores e a saída é o resultado.

4.1.1. ARRAYS – AGRUPANDO DADOS

Arrays em si não são estruturas, mas sua compreensão é muito importante para a utilização de *loops* de diversos tipos. Sua função é a de agrupar diversos dados em apenas um conjunto de dados, tratado como um dado de dimensão maior.

O agrupamento de dados é extremamente útil para seu tratamento e análise. A dimensão das vantagens do agrupamento de dados é mais bem compreendida na aplicação, como se verá nos exemplos adiante.

Existem basicamente duas maneiras de agrupar dados – *arrays* e *clusters*. *Arrays* agrupam dados de um mesmo tipo – por exemplo, diversos *strings* individuais, diversos valores numéricos, diversos valores booleanos. Clusters por sua vez podem agrupar dados de diferentes tipos, como é o caso de erros. Clusters são mais complexos, e, portanto, nessa seção só serão tratados *arrays*.

Sobre *arrays*, é importante considerar que podem ser constituídos não só de dados individuais, mas também de dados já agrupados. Por exemplo, pode-se ter um *array* de valores individuais, um *array* de *arrays* de valores individuais, um *array* de *arrays* de *arrays* de valores individuais e assim por diante.

Nessa lógica, um *array* sempre tem uma dimensão a mais que seus componentes. Pensando em números, o dados individual teria dimensão 0, o *array* de dados individuais seria um vetor, com dimensão 1, o *array* de *arrays* seria uma matriz com dimensão 2 e assim por diante.

Assim, a cada termo de um *array* está associado um número de índices igual a sua dimensão. Cada índice parte de zero e vai até o número de termos nessa dimensão decrescido de um.

A maneira mais fácil de construir um array é com a função '*build array*'. As entradas são os elementos do *array*, todos do mesmo tipo e dimensão (número de entradas modificável) e a saída é o *array* unido. Caso se seleciona '*concatenate arrays*' os elementos são justapostos em um *array* de mesma dimensão. Por exemplo, unindo (0, 1) e (2, 3, 4) se tem (0, 1, 2, 3, 4). Caso não se selecione, então a dimensão é incrementada em uma unidade. Por exemplo, unindo (0, 1) e (2, 3) se tem a matriz com cada coluna sendo um dos vetores somados. Note que nesse caso, o número de elementos dos *arrays* unidos deve ser igual.

Existem muitas outras maneiras de construir *arrays*, como se verá adiante. Abaixo segue um exemplo de programa construindo *arrays* e encontrando seus elementos por índice.

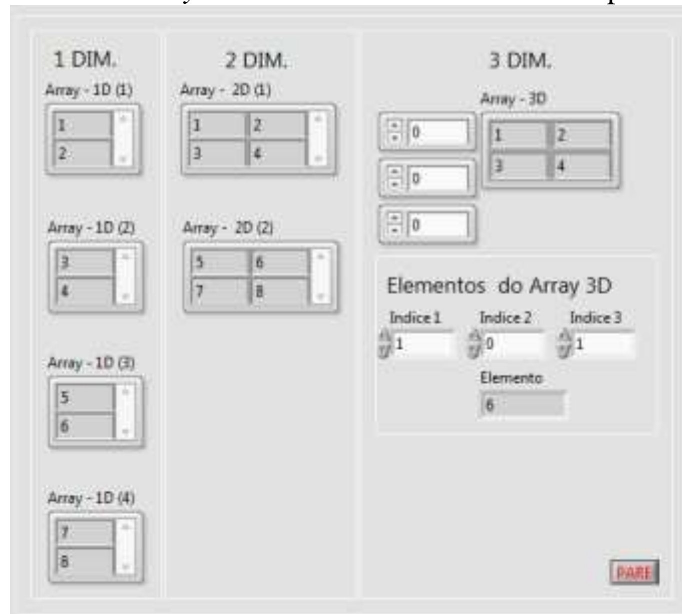


Figura 13 - Exemplo de Arrays, Painel Frontal

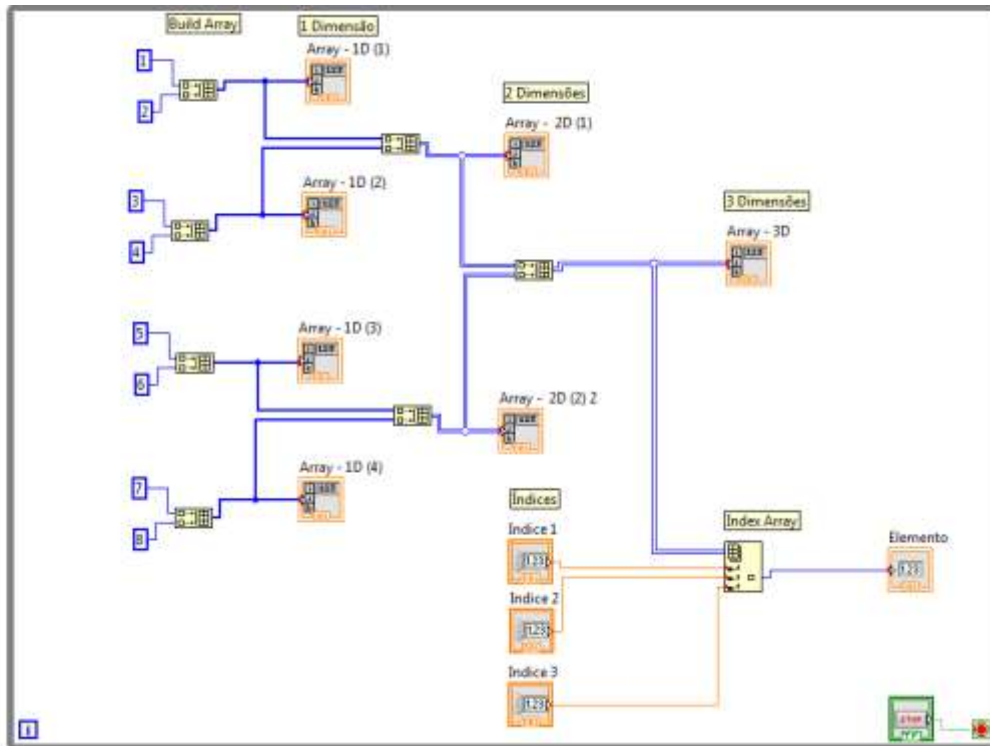


Figura 14 - Exemplo de Arrays, Diagrama de Blocos

Como se percebe, o programa tem basicamente duas funcionalidades. Partindo de 8 valores numéricos constantes, constrói *arrays* de dimensão crescente, unindo, gradualmente os elementos 2 a 2 até obter um *array* de 3 dimensões, cada dimensão com comprimento 2, ou seja $2^3 = 8$ elementos. Em seguida, recebendo 3 índices, devolve o elemento correspondente do array de 3 dimensões.

Para unir elementos em arrays de dimensão crescente, utilizou-se a função '*build array*' com a opção '*concatenate arrays*'. Nota-se que os *arrays* de uma dimensão são tratados como linhas. Como já antecipado também, conforme a dimensão dos dados transportados aumenta, aumenta também a espessura dos fios no diagrama. Essa característica é extremamente valiosa em códigos de maior complexidade e quando se está corrigindo erros em algum programa.

Para localizar o elemento no *array* de 3 dimensões (conjunto de matrizes), utilizou-se a função '*Index Array*', que recebe um *array*, um número correspondente de índices No caso, no painel frontal se verifica os índices (1, 0, 1). Em LabVIEW, como citado na página de suporte online da NI sobre a função *Index Array*, os índices são recebidos sempre em ordem decrescente de dimensão (National Instruments, 2013.). Ou seja, no caso, página – linha – coluna. No caso, trata-se do elemento na segunda página, primeira linha e segunda coluna.

4.2.ITERAÇÕES – 'FOR LOOPS', 'WHILE LOOPS' E SEUS ELEMENTOS

Uma das grandes vantagens de aplicações computacionais em engenharia é justamente a programação e rotinas iteradas. De fato, as estruturas que garantem a programação dessas rotinas são de suma importância.

4.2.1. ESTRUTURA DOS LOOPS

De uma maneira geral, a função intrínseca de programas computacionais para engenharia está ligada à rápida realização de operações repetitivas, em rotinas definidas. Com efeito, a estrutura de 'laços', os *loops*, tem papel importantíssimo nesse ramo de aplicações.

A estrutura de laços consiste em um conjunto de operações, aplicadas sobre entradas e resultantes em saídas, realizadas repetidamente enquanto uma condição for válida ou até que uma condição seja atingida.

Existem dois tipos básicos de *loops*, os de tipo '*for'* e os de tipo '*while'*.

Loops de tipo '*for*' possuem um número de iteradas definido. Ou seja, estabelecido esse número, os comandos definidos serão realizados esse número de vezes, após o qual o funcionamento do loop é interrompido.

Na plataforma LabVIEW os '*For Loops*' se apresentam de maneira intuitiva e semelhante à estrutura de casos.

Figura 15 - For Loop

Como se vê, a estrutura é composta por um retângulo, um ícone 'N' e um ícone 'i'.

De maneira análoga à estrutura de casos, os comandos efetuados em cada iteração devem ser dispostos dentro do retângulo e os pontos de entradas e saídas são representadas por quadrados.

O ícone 'N', por sua vez, possuí um terminal de chegada de dados. A esse terminal deve-se conectar um cabo com dado de valor numérico inteiro, o qual definirá o número de iterações. Note que esse dado pode ser proveniente de constantes, controles, processos, etc.

O ícone 'i', por sua vez, possuí um terminal de saída de dados. Para cada iteração, assim, esse ícone fornece o valor da iteração correspondente, de '0' até 'n-1'.

Loops de tipo *while*, no entanto, a principio não possuem número de iterações definidas. No caso específico da plataforma LabVIEW, ao final de cada iteração o loop verifica se uma dada condição foi atingida. Caso sim, então o funcionamento do loop é interrompido e as saídas são entregues. Do contrário uma nova iteração é realizada.

É extremamente importante levar em consideração o fato de que a verificação do atingimento ou não de uma condição é realizada ao fim da iteração. Assim, sempre ocorre, no mínimo, uma iteração.

A estrutura dos '*While Loops*' é semelhante em vários aspectos à dos '*For Loops*'.

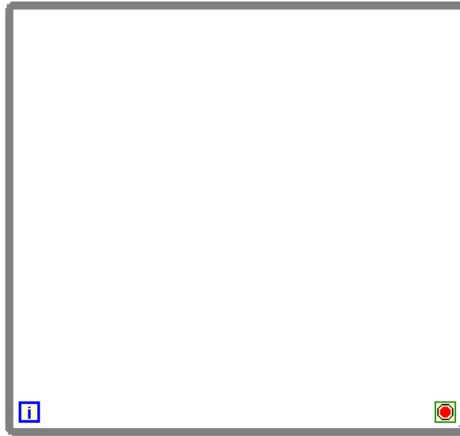


Figura 16 - While Loop

Novamente tem-se um retângulo e um ícone ‘i’, mas ao invés de um ícone ‘N’ se tem o ícone de uma ‘pare’.

O significado do retângulo e do ícone ‘i’ é idêntico ao do ‘*For Loop*’. No entanto, agora será o ícone ‘pare’, e não o ‘N’, que regulará as iterações do loop. O ícone ‘pare’ possuí um terminal de chegada de dados. Se ao final de uma iteração o valor verificado nesse terminal for ‘TRUE’, então o funcionamento do loop é interrompido. Do contrário uma nova iteração é efetuada.

É extremamente usual manter todo o código dos programas dentro de um grande ‘*While Loop*’ com condição de parada atrelado a um comando booleano de botão ‘PARE’. Note que a maioria dos exemplos apresentados até então segue essa recomendação.

Manter o código dentro de um *loop* de *while* é útil pois assim se garante que o programa continue rodando até que o usuário deseje parar. Do contrário, após realizar as atividades relacionadas uma vez o programa interrompe o seu funcionamento.

4.2.2. AUTO-INDEXING

Dois elementos são essenciais para a boa operação de *loops* – o *auto-indexing* e os *shift registers*. Nessa seção trata-se do *auto-indexing*.

Como visto, os *loops* realizam rotinas de maneira repetida até que uma condição seja estabelecida. Recebem assim, dados de entrada e possuem também dados de saída. Note, no entanto, que, para cada iteração, um valor diferente chega ao nó de saída.

Caso nada contrário seja indicado no código, a cada iteração, para cada saída, é liberado um dado. Dessa maneira, o valor entregue é atualizado a cada iteração, perdendo-se os resultados anteriores. Assim, possui-se um regime de saídas dinâmico, durante o funcionamento do *loop*, sendo as saídas de mesma dimensão que os dados entregues.

Existem aplicações, no entanto, onde se têm o interesse em investigar, ler e processar não as saídas do *loop* ao longo do tempo, mas sim o conjunto de suas saídas. Como indicado na página de suporte online da NI sobre aplicação de *Auto-Indexing*, para isso serve tal elemento (National Instruments, 2013). O *auto-indexing* pode ser ativado em qualquer saída de um *loop*, e o ícone do nó passa de um quadrado preenchido para um quadrado com o símbolo ‘[ ]’.

Uma vez ativada essa opção, a saída do *loop* passa a ser um array do tipo do dado associado. Assim, a saída passa a ter uma dimensão a mais que o dado que chega no nó por dentro do ‘Loop’. Note ainda que o número de elementos desse array é idêntico ao número de iterações do ‘Loop’. Veja abaixo:

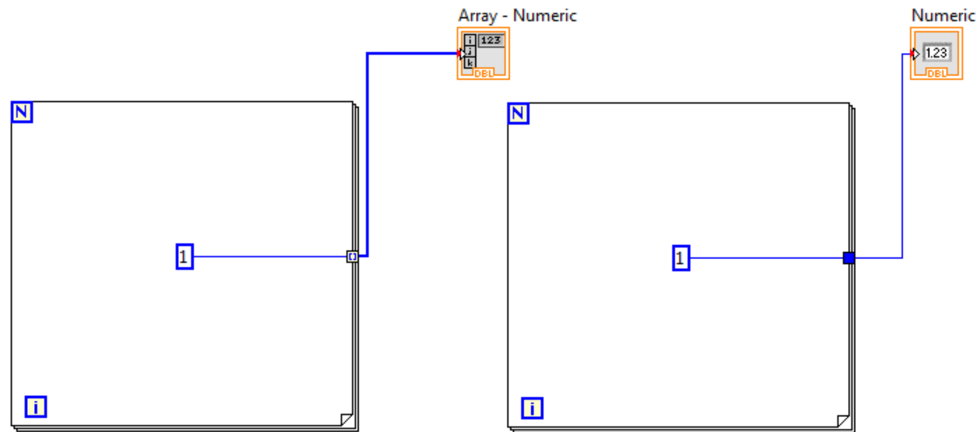


Figura 17 - Auto-Indexing

Nesse caso, a estrutura da esquerda tem *auto-indexing* e a da direita não. O dado que chega, por dentro do *loop* ao nó é sempre o valor numérico de '1', cuja dimensão, portanto, é '0'. Com efeito, no caso da esquerda a saída será um array de dimensão 1, de 'N' elementos, com valor '1' para cada elemento. No caso da direita, no entanto, a saída será simplesmente o valor de '1', dimensão '0'. Essa diferença se evidencia pela espessura dos cabos de saída.

Outra diferença é que, no caso da esquerda, o *loop* só libera o dado de saída – um *array* quando interrompe seu funcionamento. No caso da direita, no entanto, a cada iteração, mesmo enquanto opera o *loop*, é liberado um novo dado.

4.2.3. SHIFT REGISTERS

Caso nada contrário seja comunicado, a cada iteração de um *loop* os dados de entrada utilizados serão aqueles que chegam pelos cabos que entram no *loop*. Assim, a principio, no início de cada iteração, o *loop* verifica os dados que chegam por cada entrada e utiliza-os para a iterada.

Os *shift registers* permitem usar como dados de entrada para uma dada iteração 'n+1' (n maior ou igual a 0) uma saída de tipo igual da iteração 'n'. A iteração '0', por sua vez, terá valor de entrada do cabo que chega, por fora do *loop*, no nó. Ou seja, o valor que chega por fora do *loop* no nó de entrada funciona apenas como um valor de inicialização.

Shift registers são representados por ícones com setas na mesma horizontal. A entrada é uma seta para baixo e a saída, para cima.

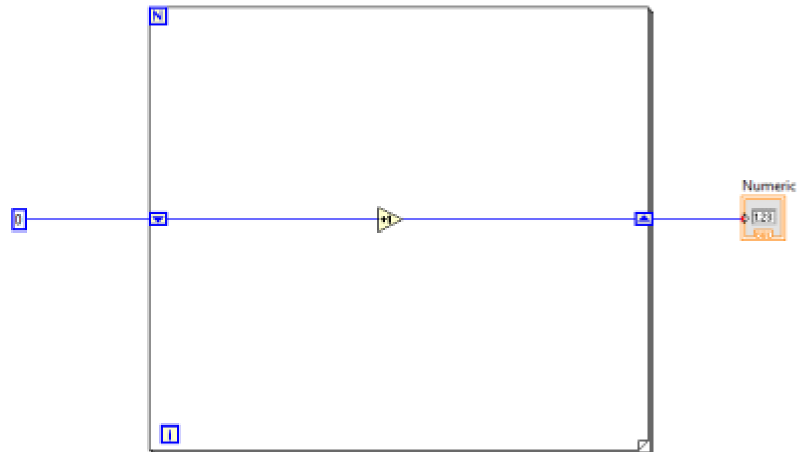


Figura 18 - Shift Registers

Nesse caso, para a iteração ‘0’ o valor de entrada vale ‘0’, e o de saída ‘1’. Para a iteração ‘1’, a entrada é ‘1’ e a saída ‘2’. Generalizando, para a iteração ‘n-1’ a entrada é ‘N-1’ e a saída é ‘N’. Com efeito, a saída do ‘loop’ como um todo será, portanto, ‘N’, a cada iteração.

4.3.REUTILIZANDO ROTINAS – SUBVIs

Muitas vezes, especialmente em aplicações de engenharia, uma rotina é reutilizada diversas vezes num mesmo código ou em diversos códigos diferentes. Por exemplo, toda vez que é realizada uma análise preliminar de uma estrutura, os diagramas de seus esforços são traçados. Assim, seria útil uma ferramenta que permitisse a rápida replicação da referida rotina.

4.3.1. CRIAÇÃO E UTILIZAÇÃO DE SUBVIS

Na plataforma LabVIEW qualquer VI ou parte de VI pode ser rapidamente convertida em uma módulo, a qual pode ser utilizada posteriormente como subunidade de outro VI, Drummond (1998). Assim, a qualquer momento pode-se selecionar parte de um código e convertê-lo em um módulo.

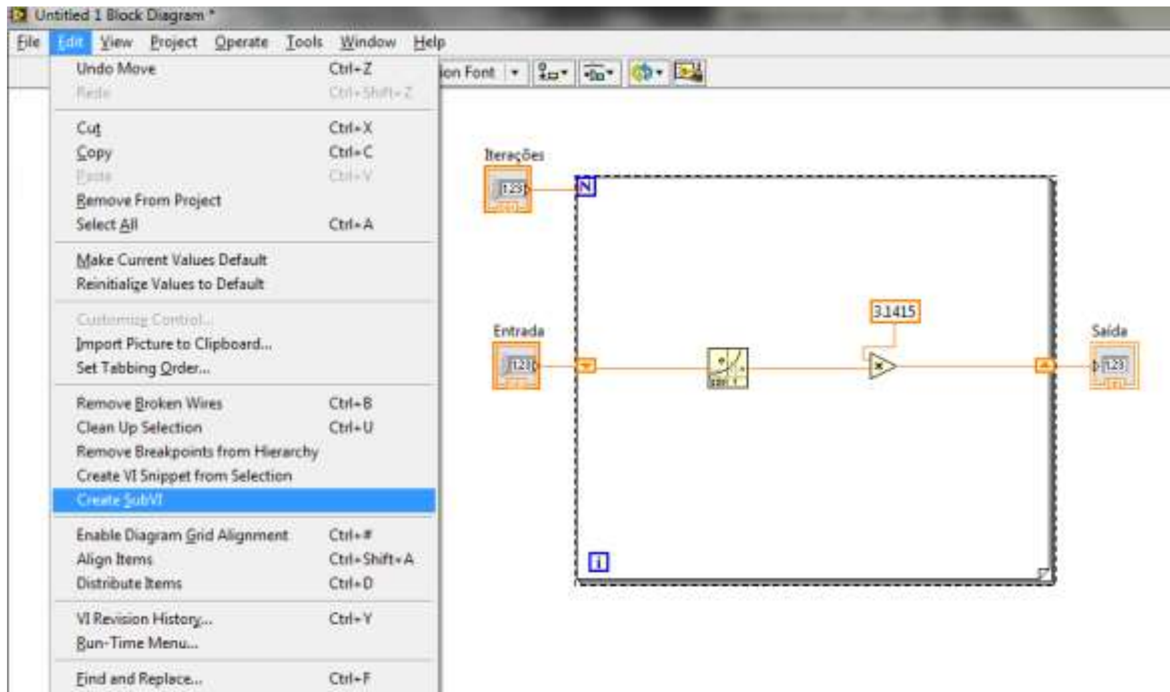


Figura 19 - Criação de SubVIs

Efetuada essa operação, a seleção se torna uma caixa que já pode ser manipulada como qualquer função. Os fios que entravam na seleção se tornam automaticamente os terminais de entrada e os que saíam, os de saída, como se observa na figura abaixo, resultante da operação da figura acima.

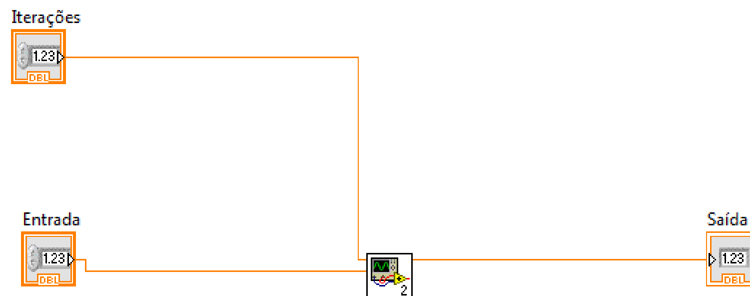


Figura 20 - SubVI Criado

Agora esse conjunto de ações, convertido em função pode ser copiado, colado e movido mais facilmente pelo código.

O SubVI criado também pode também ser editado como um VI qualquer. É importante notar no painel frontal que os terminais de entrada do SubVI correspondem a comandos e os terminais de saída correspondem a indicadores, isso é, saídas e entradas são referenciadas convenientemente. De maneira simplificada entende-se que os dados que o usuário entraria nos comandos do VI serão os dados que chegam aos terminais de entrada, quando o mesmo for aplicado como SubVI e os dados que apareceriam nos indicadores do VI serão os dados que sairão pelos terminais de saída do SubVI.

Nota-se que é extremamente oportuno salvar em uma pasta o SubVI criado, para poder utilizá-lo em outros programas. Uma vez salvo, dessa maneira, o SubVI é tratado como um VI qualquer.

Algo importante de ser considerado é que todo VI pode ser utilizado como um SubVI e todo SubVI é um VI em si, que pode ser aplicado isoladamente. De fato, a qualquer momento pode-se utilizar um VI da biblioteca do programador como SubVI .

4.3.2. EDIÇAO DE SUBVIS

Como já desenvolvido, todo SubVI é, em si, um VI e portanto pode ser editado e modificado como tal. Entretanto existem algumas particularidades que devem ser notadas quando se pretende utilizar um VI como SubVI.

Como qualquer função, um SubVI é apresentado no código como um bloco. É oportuno, assim, que a aparência desse bloco permita a rápida identificação das ações que o SubVI executa.

A aparência do bloco de um VI, quando aplicado como SubVI pode ser editado em seu painel frontal. O ícone do bloco aparece, como padrão, no canto superior da tela do painel frontal.

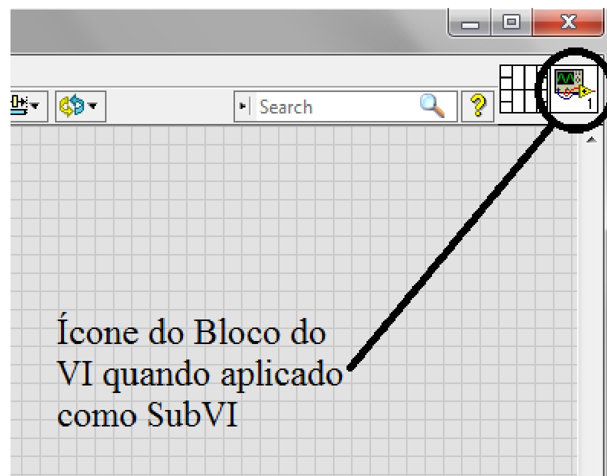


Figura 21 - Localização do Ícone

Um duplo clique sobre o ícone abre um editor amplo e intuitivo da aparência do ícone. O ícone pode indicativos verbais ou gráficos da função implementada.

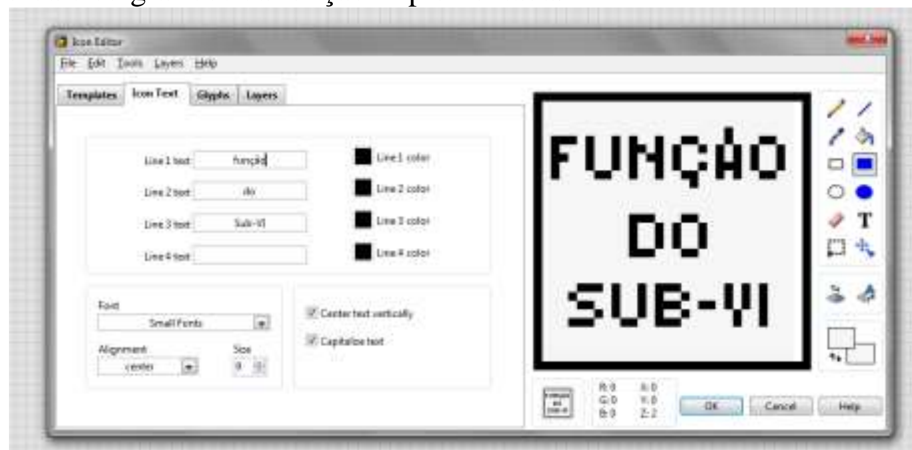


Figura 22 - Edito de Ícones

Outro elemento importante na edição de VIs para aplicação como SubVIs é a definição dos seus terminais de entrada e saída. Como já explicado, ao criar um SubVI a partir de um trecho de um VI, entradas e saídas são automaticamente definidas. Entretanto podem haver casos onde tais elementos

tenham que ser modificados, ou ainda casos onde um VI criado normalmente tenha que ser implementado como SubVI.

A edição dos terminais do VI se dá por um ícone localizado novamente no canto superior direito do painel frontal, ao lado do símbolo do VI.

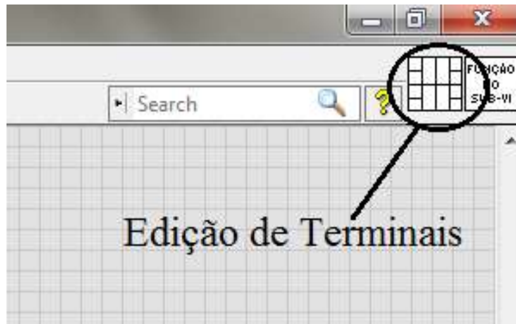


Figura 23 - Edição de Terminais

No referido ícone notam-se um série de quadrados. Cada um desses quadrados corresponde a um terminal de entrada ou saída do VI, quando aplicado como SubVI. Os terminais de entrada e saída do SubVI são referenciados a comandos e indicadores do VI correspondente. Como já explicitado, os dados que chegam aos terminais serão aplicados nos comandos referenciados, e os dados que saem pelos terminais de saída serão os mostrados pelos indicadores, no ato da aplicação do VI como SubVI.

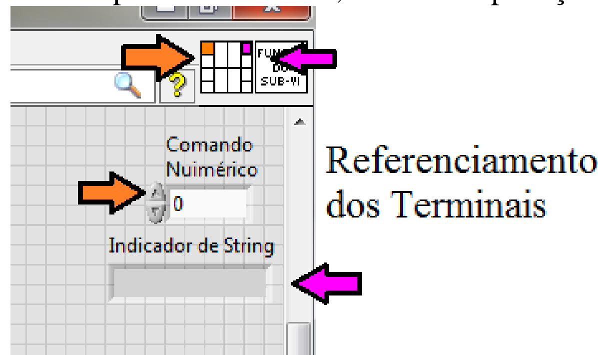


Figura 24 - Refernciamento de Terminais

Percebe-se que ao referenciar um terminal, o mesmo assume a cor correspondente a natureza do dado relacionado. É oportuno ainda que comandos sejam referenciados a terminais da esquerda e indicadores aos da direita, procurando manter o padrão das funções em geral.

4.4.CONTROLE DO TEMPO NOS VIs – WAIT E WAIT UNTIL NEXT ms MULTIPLE

Para muitos aplicativos e programas computacionais o controle do tempo, apesar de sempre útil, não assume importância tão grande, principalmente na ótica do usuário. Entretanto, no caso das aplicações de engenharia, de monitoramento em tempo real esse controle assume grande relevância, dadas as altas frequências de aquisição e uso da CPU.

Caso nada se especifique, o tempo de execução de um programa, ou da iteração de um loop será simplesmente o tempo necessário, pela máquina, para executar as operações necessárias. Esse tempo quase nunca é controlável facilmente e pode inclusive variar de máquina para máquina, ou até na mesma máquina, dependendo de inúmeras condições. De fato, segundo o manual do módulo Real-Time do LabVIEW, "*A qualquer momento, o Sistema Operacional pode atrasar a execução de um programa do usuário por várias razões: rodar um escaneamento de vírus, atualizar gráficos, realizar tarefas*

secundárias, e assim por diante"[2]. (National Instruments. NI LabVIEW for CompactRIO Developer's Guide 2013, tradução livre)

Essa falta de controle do tempo pode causar uma série de problemas, como perda da sincronia de operações que deveriam ser simultâneas ou problema na análise de dados em tempo real – por exemplo, o sensor envia um valor por segundo, mas o loop tem execução em 0,1 segundos – dentre outros. Um programa que roda muito rápido pode ainda ser inoportuno para a visualização dinâmica de dados pelo usuário.

Existem uma série de funções na plataforma LabVIEW cujo intuito é controlar o tempo e solucionar os problemas enunciados. As diversas funções possuem maior ou menor rigor, precisão e diferentes aplicações.

As funções que garantem controle do tempo mais flexível, completo e preciso, de uma maneira geral, só podem ser acessadas com a instalação dos módulos complementares FPGA e *Real Time* do LabVIEW, bem como o uso de hardware adequado, como o *CompactRIO*. Para aplicações que exigem maior confiabilidade e precisão, como controle de algumas máquinas e monitoramento em tempo real, essas funções podem ter maior importância. Seu uso é consideravelmente mais complexo e está sendo ainda estudado nesse momento do projeto de Iniciação Científica, para melhor compreensão.

Entretanto, um grande número de impasses pode ser solucionado com duas funções – *Wait e Wait Until Next ms Multiple* – presentes no módulo padrão do LabVIEW.

A função '*Wait*' é muito útil para estipular, com precisão, a duração de atividades do programa - em geral, de uma iteração de um *loop*. Essa função, se colocada dentro de um loop, fará com que cada ciclo tenha uma duração controlada, isso é, aguarde uma quantia determinada de tempo até iniciar o próximo ciclo. Essa quantia é definida pelo número de milissegundos referente a um valor numérico inteiro, conectado a um terminal da função.

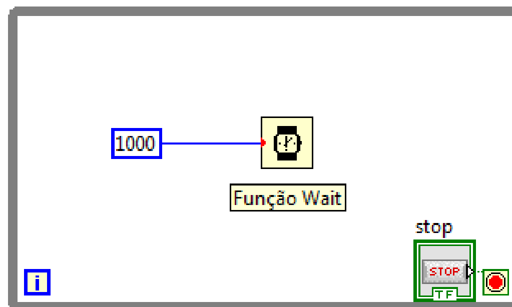


Figura 25 - Função Wait

Note, no entanto que a função '*Wait*' não define de maneira absoluta a duração de um loop. Essa função meramente estipula um limite inferior para a duração do *loop*. Isso é, a iteração durará no mínimo o valor estipulado. Assim, se, por exemplo, no loop indicado acima as operações demandadas exigirem tempo maior que 1 segundo para serem efetuadas, a duração do loop não será mais controlada plenamente.

Com efeito, nota-se que a estipulação do valor a ser esperado deve ser feita com muito cuidado, de forma a ser consideravelmente maior que o tempo requerido pelas operações em questão.

Note ainda, que se por um lado a função '*Wait*' é muito útil para determinar a duração de iterações, não é muito útil para efetuar sincronização de operações paralelas. Isso porque trata o

---

[2] - At any time, the operating system might delay execution of a user program for many reasons: to run a virus scan, update graphics, perform system background tasks, and more.

tempo de maneira essencialmente relativa, pois meramente mede intervalos de tempo com origem no instante do início da iteração do loop, quase sempre desconhecidos.

Para a sincronização de operações paralelas, portanto, pode-se utilizar a função *Wait Until Next ms Multiple*. Essa função faz com que cada iteração de um dado *loop* aguarde até o valor registrado no timer de milissegundos seja um múltiplo do valor associado. Assim, como o timer possuí um referencial de tempo definido e igual para todos os loops de um dado VI, essa função pode ser utilizada para garantir que as iterações dos diversos loops do programa ocorram de maneira síncrona.

Novamente há que se tomar o cuidado de estipular durações de espera maiores que as de execução da iteração. Do contrário o loop pode ter de esperar mais de um múltiplo do valor especificado, o que pode gerar problemas.

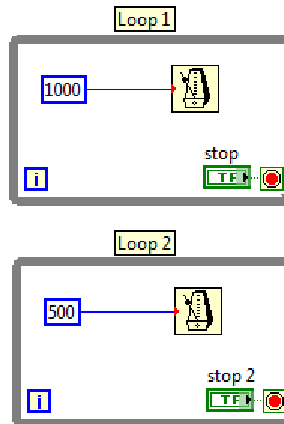


Figura 26 - Função Wait Unil Next ms Multiple

Nesse caso, se não houver restrições de duração das operações realizadas, o *Loop* 1 irá iniciar nova iteração sempre que o timer registrar múltiplos de 1s, ou seja, a cada 1000ms. Já o *Loop* 2 iniciará iteração sempre que o timer registrar múltiplo de 0,5s, ou seja, a cada 500ms.

Portanto, a cada iteração do Loop 1, o Loop 2 irá realizar duas iterações, sendo que uma delas se iniciará no exato mesmo instante que a iteração do Loop 1.

A grande limitação da função '*Wait Until Next ms Multiple*', que faz com que em certos casos a função '*Wait*' seja mais oportuna é o fato de que, de maneira geral, como indica o manual da função, a primeira iteração de um *loop* terá duração menor que a estipulada. Isso porque quase sempre o valor no timer, no instante do início da execução da primeira iteração do loop não será um múltiplo do valor estipulado (National Instruments, 2013).

4.5.REPRESENTANDO RESULTADOS GRAFICAMENTE – CHARTS E GRAPHS

É de consenso geral que nas aplicações de engenharia é extremamente útil representar dados graficamente, para sua análise, identificação de suas tendências, suas ordens de grandeza, etc. No ambiente LabVIEW as há duas maneiras principais de representar dados graficamente – os *Graphs* (que podem ser tipo XY) e os *Charts* – ambos indicadores gráficos, os quais podem ser dispostos no painel de controle.

4.5.1. CHARTS

Charts são indicadores em forma gráfica que recebem e apresentam dados ao longo do tempo. De fato, segundo Drummond (1998), distingue um indicador gráfico como Chart quando apenas um

valor do dado está disponível por vez. Isso é, os *Charts* representam em tempo real os dados que nele chegam pela estrutura do VI, de maneira dinâmica. São úteis, portanto, para representar dados coletados ainda no decorrer de um experimento em curso, tendo maior relação com o monitoramento em tempo real.

Assim, o eixo das abscissas de um Chart, de uma maneira geral, será o tempo real, decorrido durante o funcionamento do programa. Esse tempo pode partir de uma origem pré-definida, nas propriedades do Chart. O eixo das ordenadas, por sua vez, representa os valores dos dados que chegam ao terminal do Chart, referenciados ao instante de sua chegada.

O Chart, portanto, é um indicador que possuí apenas uma entrada, que recebe valores numéricos de dimensão zero.

Uma das maneiras mais comuns de emprego de *Charts*, portanto é seu uso dentro de *loops* com tempo de iteração controlado. Por estar dentro do *loop*, o *Chart* apresenta o valor assumido pelo dado que flui por seu terminal em cada iteração, confrontado com o tempo, controlado pela duração de cada iteração.

Esse modo de aplicação pode ser exemplificado pelo diagrama de blocos que consta abaixo.

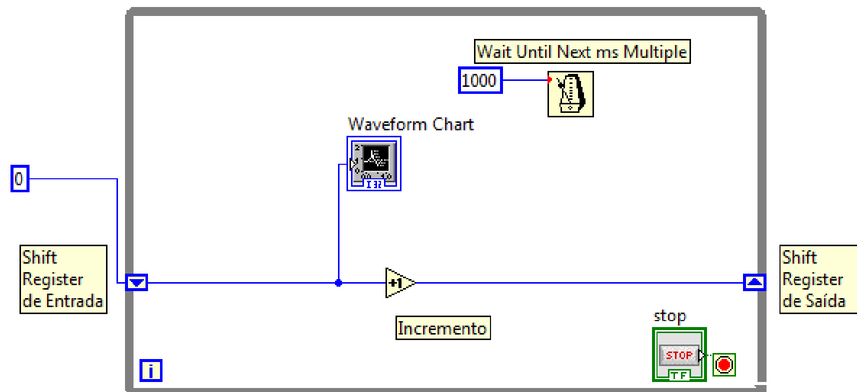


Figura 27 - Exemplo de Charts, Diagrama de Blocos

Nesse caso construiu-se um *loop* e dentro dele alocou-se a função '*Wait Until Next ms Multiple*'. Assim, garante-se que, excetuando-se a primeira iteração, todas as iterações durarão um segundo. Logo, obtém-se certo controle da duração dos ciclos do *loop*.

Observa-se ainda, que o dado que flui pelo cabo azul, partindo de zero na primeira iteração é sucessivamente incrementado graças aos *shift registers*. Assim, a cada iteração o *Chart* deve plotar novos pontos, cuja abscissa é o tempo decorrido e cuja ordenada é o índice da iteração (partindo de zero). O gráfico obtido consta abaixo:

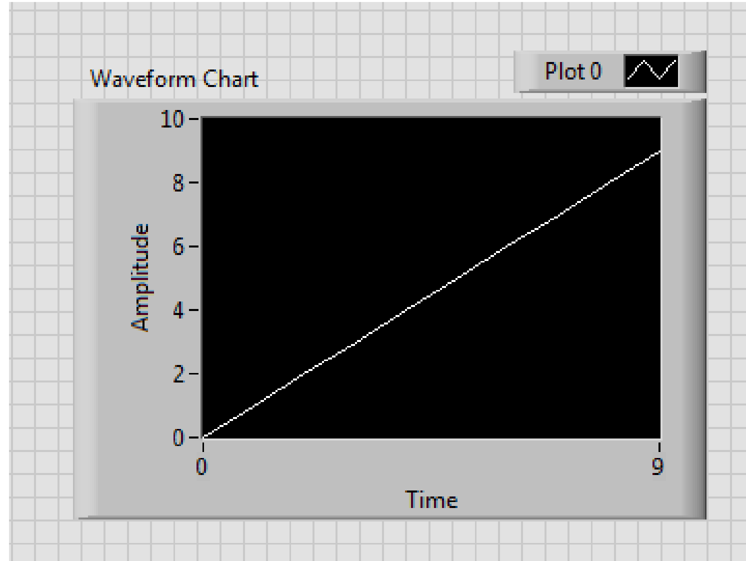


Figura 28 - Exemplo de Charts, Painel Frontal

Nota-se ainda que é possível plotar mais de um dado no mesmo *Chart*, o que é muito útil para efeito de análises comparativas. Uma das maneiras de fazê-lo é ligar todos os cabos com dados a serem plotados nos terminais de entrada função '*Bundle'* e então conectar o cluster gerado no terminal do *Chart*.

No caso ilustrado abaixo se adicionou ao *Chart* do exemplo anterior uma plotagem de um dado que se mantém constante no valor de 5. Note como a função '*Bundle*' foi utilizada para agrupar os dados.

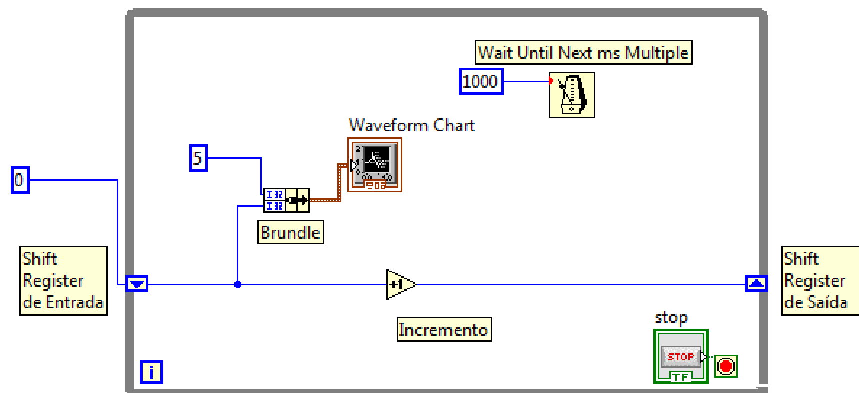


Figura 29 - Múltiplas Plotagens, Diagrama de Blocos

O gráfico gerado por esse diagrama de blocos, por sua vez deverá ser o seguinte:

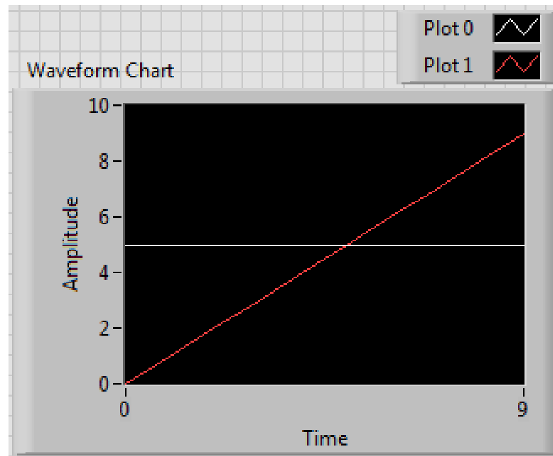


Figura 30 - Múltiplas Plotagens, Painel Frontal

4.5.2. GRAPHS

Graphs diferenciam-se de Charts basicamente por não receberem ou representarem dados ao longo do tempo real. Isso é, tratam-se de indicadores gráficos que recebem e apresentam, de maneira *estática*, todos os dados de um determinado espaço amostral.

Assim, *graphs* recebem um dado 'pacote' de pontos (coordenadas X e Y) ou de informações de uma curva e representa-os graficamente. Como explica Drummond (1998), em graphs, todos os dados a serem plotados já estão disponíveis juntos. São úteis, portanto, para representar dados já coletados e armazenados, referentes a um experimento já concluído. Os tipos de dados que recebem podem ser arrays, mas mais comumente são dados tipo '*Dynamic Data*', cujo cabo é marrom tracejado e espesso.

Em primeiro lugar têm-se os *Waveform Graphs*. Uma das maneiras de utilizar esses indicadores é conectá-los a arrays numéricos de duas dimensões, onde a primeira linha explicita as abscissas e todas as linhas representam as ordenadas de casa plotagem. Entretanto, para esse tipo de gráfico o indicador ideal são os *XY Graphs*.

Waveform Graphs são mais úteis para plotar curvas geradas a partir de dadas informações. Existem inúmeras funções que geram e editam curvas (*Dynamic Data*) na biblioteca LabVIEW. Assim, gerada a curva com as informações disponíveis e convenientes, basta conectar o referido cabo a um *Waveform Graph*.

O exemplo a seguir ilustra a construção de uma curva e sua plotagem em um *Waveform Graph*.

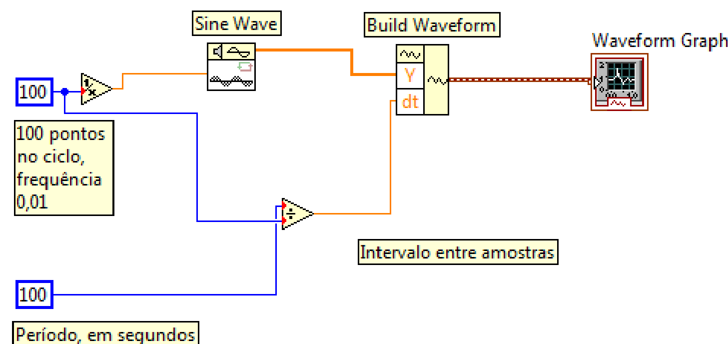


Figura 31 -Exemplo Waveform Graph, Diagrama de Blocos

No caso, primeiramente utilizou-se a função ‘*Sine Wave*’ para gerar os valores das ordenadas da curva. Assim, utilizou-se a amplitude e fase inicial padrões (1 e 0, respectivamente), especificando-se apenas a frequência, no caso 1 ciclo da função seno a cada 100 valores gerados.

Em seguida empregou-se a função ‘*Build Waveform*’ para criar a curva. No caso especificaram-se os valores ordenados gerados anteriormente e o intervalo entre os mesmos, obtido pela divisão do período, tomado como 100 segundos, pelo número de amostras, tomado como 100.

O gráfico obtido pelo programa é o que consta abaixo. Nota-se que de fato é uma senóide se amplitude 1 e período 100.

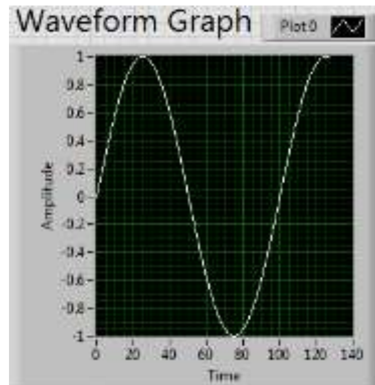


Figura 32 - Exemplo de Waveform Graph, Painel Frontal

Assim como no caso dos *Charts*, é possível plotar mais de uma série de dados no mesmo gráfico. O processo, entretanto, é um pouco diferente, dada a natureza dos dados empregados.

No caso dos *Waveform Graphs*, para plotar mais de uma curva num mesmo gráfico emprega-se a função ‘*Merge Signals*’. Essa função entrega em seu terminal de saída um dado que contém as informações da todas as curvas conectadas a seus terminais de entrada. Com efeito, basta posicioná-lo no diagrama de blocos, conectando as curvas nos terminais de entrada e ligando o terminal de saída ao *Waveform Graph*.

O exemplo a seguir ilustra esse processo.

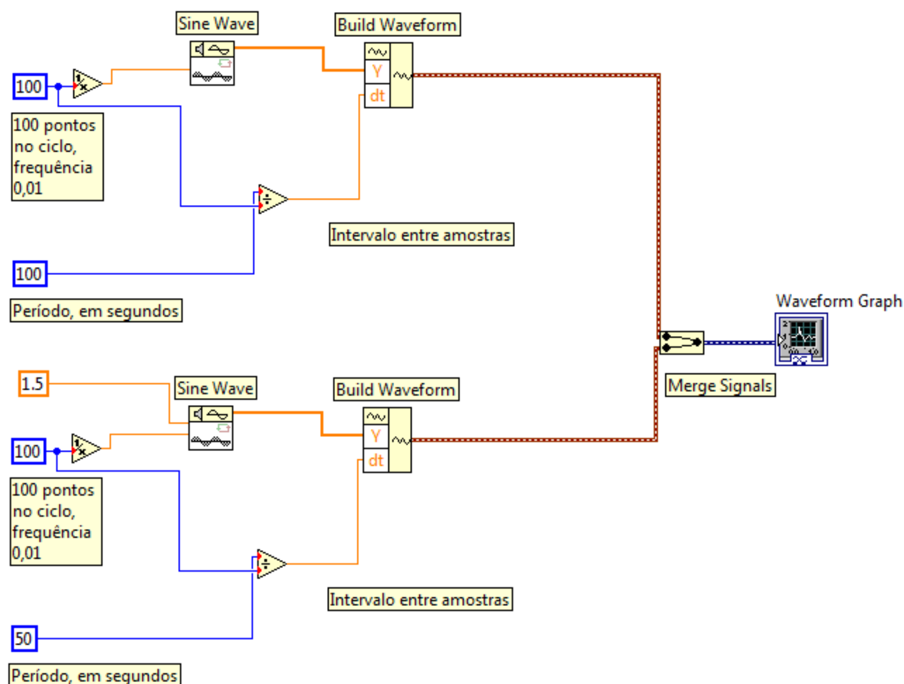


Figura 33 - Múltiplas Plotagens em Graph, Diagrama de Blocos

No caso, ao exemplo anterior adicionou-se uma nova curva, também senoidal, mas de amplitude 1,5 e período 50. Em seguida uniram-se as duas curvas pela função '*Merge Signal*', de forma a plotá-las em um mesmo gráfico. O resultado pode ser conferido a seguir.

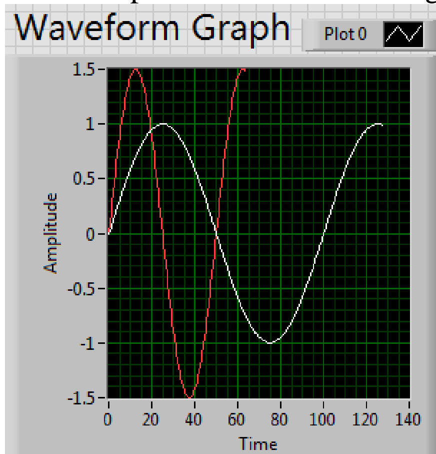


Figura 34 - Multiplas Plotagens em Graph, Painel Frontal

Novamente nota-se a coerência dos dados de entrada com os de saída.

O outro tipo básico do indicador tipo *Graph* disponível é o *XY Graph*. Assim como o *Waveform Graph*, seu emprego pode se dar de uma série de maneiras. Aqui tratar-se-á de uma delas, de simples aplicação, mas de larga utilidade.

Nesse caso, o indicador *XY Graph* recebe um *array* de uma dimensão (para uma curva) ou duas dimensões (para mais curvas) de clusters dos pares ordenados de cada ponto.

Esse método meramente utiliza uma função denominada '*Build XY Graph*', a qual recebe parâmetros do gráfico a serem plotados e retorna a curva, em arrays de clusters equivalentes a pares

ordenados. Nesse contexto, as informações essenciais a serem entregues são, obviamente, os valores de X e de Y, em '*Dynamic Data*'. Nota-se, entretanto, que escrevendo *arrays* numéricos de uma dimensão com tais informações e os conectando na função *Build XY Graph*, o LabVIEW automaticamente aplica a função '*Convert to Dynamic Data*', adequando o tipo de dado. O exemplo a seguir ilustra a aplicação desse tipo de indicador.

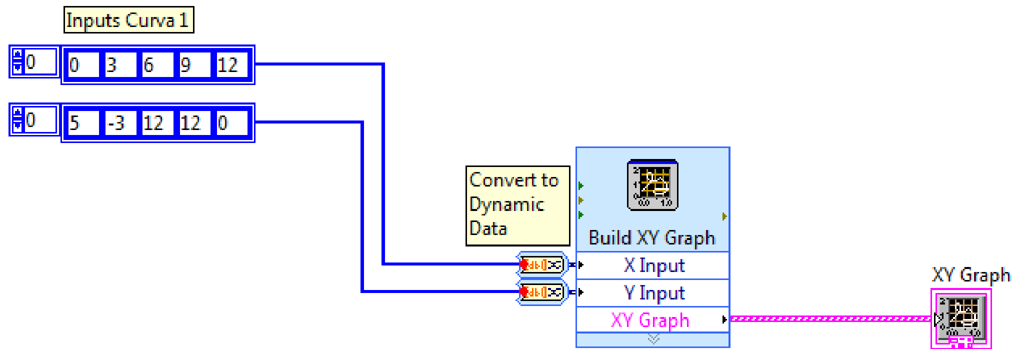


Figura 35 - XY Graph, Diagrama de Blocos

No caso, simplesmente alocou-se a função '*Build XY Graph*' e escreveu-se em *arrays* de uma dimensão os inputs de X e Y, de maneira ordenada. Conectando tais arrays as respectivas entradas o LabVIEW automaticamente dispôs as funções Convert to Dynamic Data, já configuradas adequadamente.

O gráfico obtido, portanto, foi o seguinte:

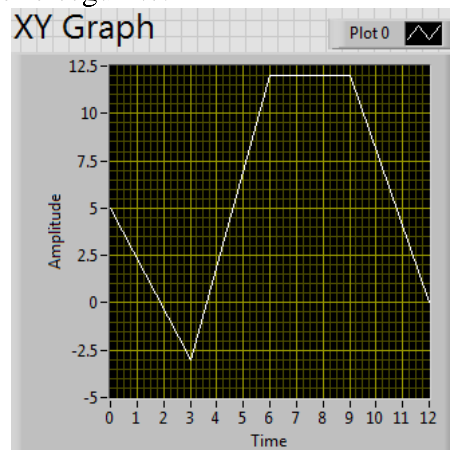


Figura 36 - XY Graph, Painel Frontal

A plotagem de mais de uma curva no *XY Graph* se da de maneira um pouco diferente. Nesse caso, ao invés de escrever para os inputs de X e de Y arrays de uma dimensão, deve-se escrever arrays de duas dimensões, onde cada linha seja o input de uma das curvas, de maneira ordenada. Ligando-se esse array novamente ao '*Build XY Graph*1, o LabVIEW automaticamente aloca a função '*Convert to Dynamica Data*'. É importante frisar que nesse caso a configuração, no *menu* '*proprieties*', dessa função é outra, então é necessário reconectar os *arrays* diretamente na função '*Build XY Graph*' para que esse acerto seja efetuado automaticamente.

O exemplo a seguir apresenta três curvas plotadas no mesmo XY Graph.

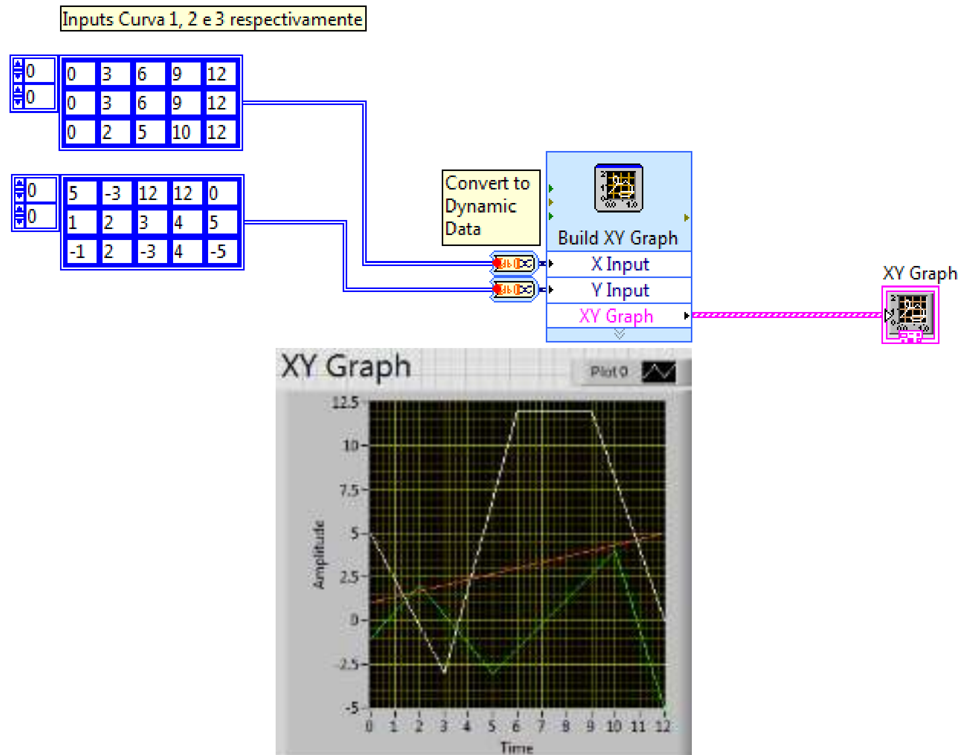


Figura 37 - Múltiplas Plotagens, XY Graph

Poder-se-ia ter utilizado também a função '*Merge Signals*' nas '*Dynamic Data*' individuais de cada séria de valores X e Y.

Os diversos tipos de indicadores gráficos da plataforma LabVIEW tem suas vantagens, facilidades e limitações. A escolha de qual deles utilizar deve ser efetuada de acordo, portanto, com as necessidades do código em questão.

4.6.ESTRUTURA DE ORDENAMENTO – FLAT SEQUENCE STRUCTURE

Como antecipado, existe ainda uma ultima estrutura básica empregada para ordenar a execução de funções sem dependência natural de dados entre si. Muitas vezes é mais rápido e simples ordenar funções criando dependências artificiais. Existe, no entanto, a opção de utilizar as '*Flat Sequence Structures*' para tal fim.

A principal vantagem desse método é que a ordem de execução das funções fica explícita e evidente no diagrama de blocos. Outra vantagem é que é possível programar facilmente funções para serem executadas após um número diverso de eventos.

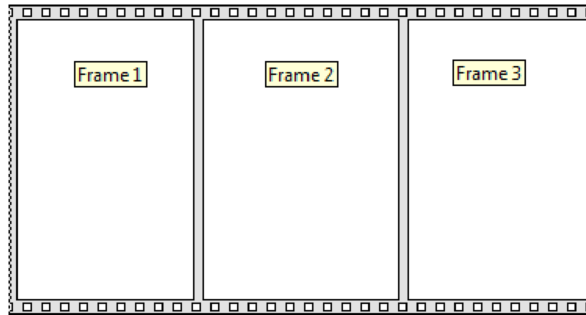


Figura 38 - Flat Sequence Structure

Uma estrutura de sequência é composta, como mostra a figura, por uma série de frames. Iniciando pelo frame da esquerda, as ações colocadas dentro dos frames são executadas, da esquerda para a direita. A execução de um frame, no entanto, somente se inicia quando a execução de todos os frames à esquerda houver findado.

4.7.EXEMPLO DE CONSOLIDAÇÃO – VELOCIDADE DO VENTO

No âmbito de consolidar os conhecimentos desenvolvidos e motivados nessa seção, foi construído um exemplo de programa utilizando as funções básicas de LabVIEW, com o uso de estruturas de casos, *loops*, indicadores gráficos e um SubVI. Buscou-se um exemplo, que, apesar de sua simplicidade, tivesse relação com uma aplicação de monitoramento em engenharia.

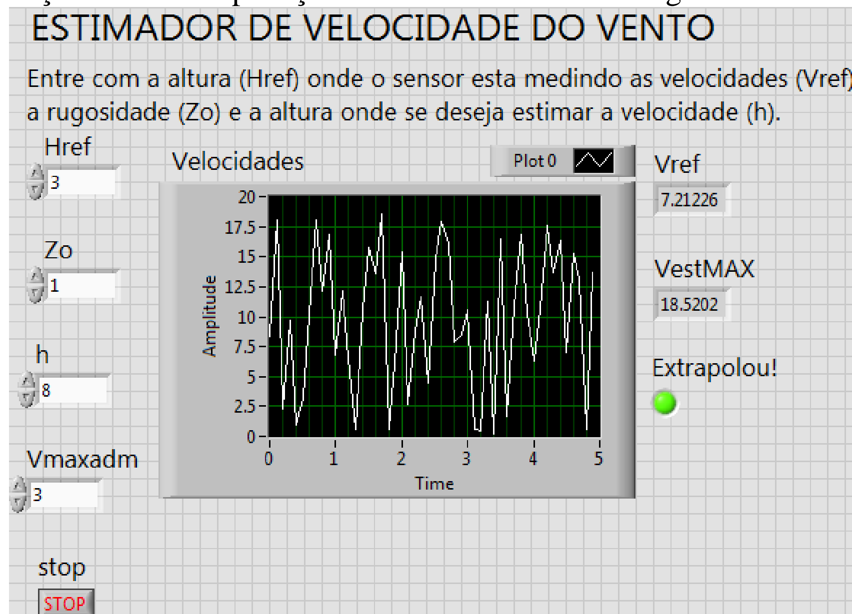


Figura 39 - Velocidade do Vento, Painel Frontal

O aplicativo em questão recebe do usuário a altura onde um sensor hipotético se instalaria, a rugosidade da superfície hipotética estudada e a altura onde se deseja estimar a velocidade, bem como uma velocidade máxima admissível.

Uma função randômica gera dados aleatórios desse sensor simulado, que mediria as velocidades do vento a uma dada altura ‘Href’, a cada 1 segundo. Tais dados simulados são processados e os resultados extrapolados – isso é, estimativas para a altura ‘h’ - são plotados em gráfico. Além disso, o

aplicativo indica a velocidade hipoteticamente medida e a velocidade estimada máxima. Caso uma das velocidades extrapoladas ultrapasse o valor admissível o usuário é comunicado pelo LED correspondente. O método utilizado para a extrapolação foi extraído de Camelo *et al*. (2010). Consta em seguida o diagrama de blocos do referente programa.

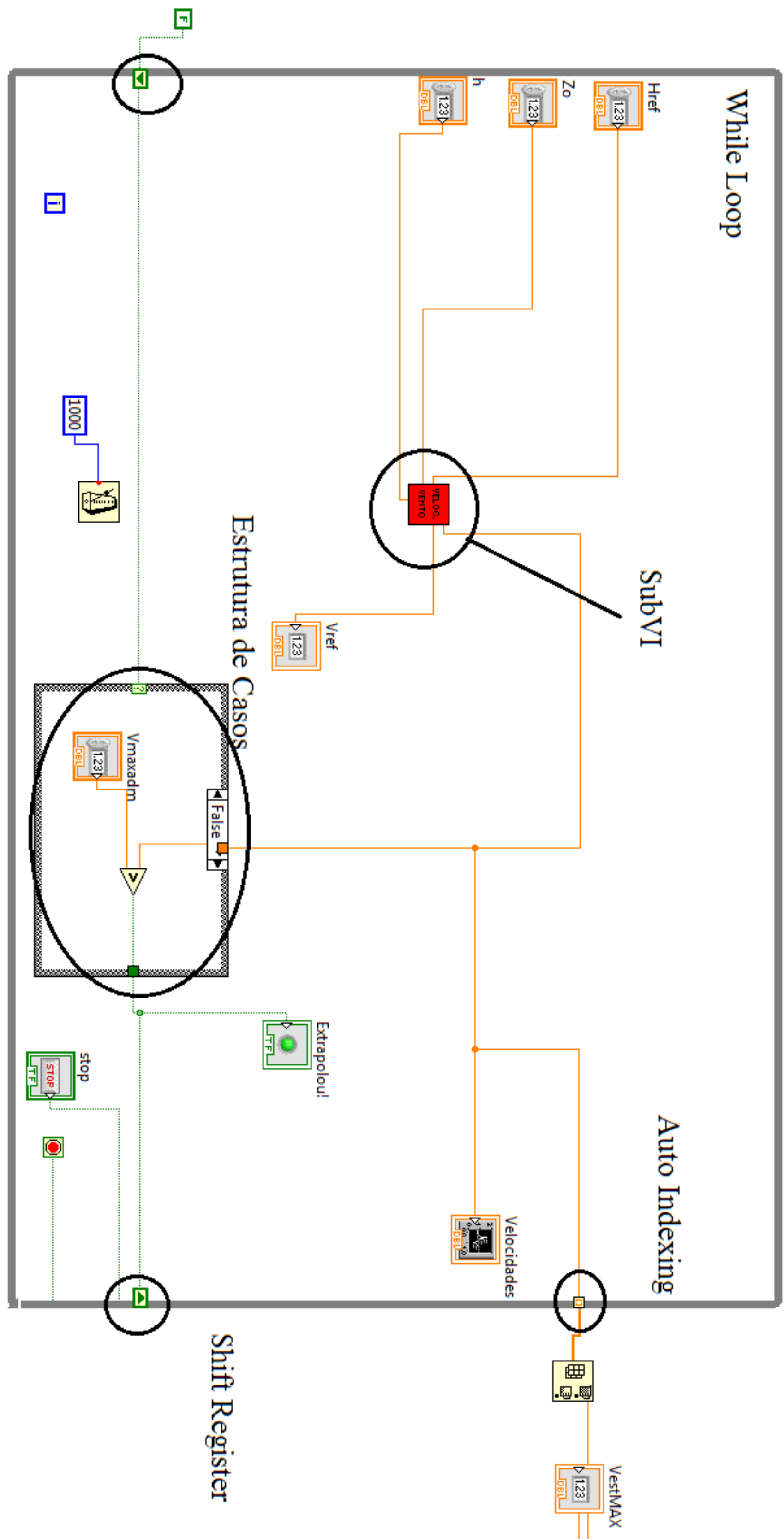


Figura 40 - Velocidade do Vento, Diagrama de Blocos

Nesse código, destacam-se alguns conceitos já elaborados.

Em primeiro lugar, destaca-se o uso de um '*While Loop*'. No caso, trata-se da repetição contínua do cálculo da velocidade extrapolada, sendo que cada iteração dura 1 segundo, graças à função '*Wait Until Next ms Multiple*' colocada dentro do *loop*, que recebe a constante 1000. Assim, a cada 1 segundo o programa obtém e plota o valor calculado para a velocidade extrapolada em um *Chart*, uma das saídas, localizada dentro do *loop*. O *loop* para quando o botão STOP é pressionado, passando a retornar '*TRUE*'.

Em segundo lugar nota-se o uso de um SubVI. O LabVIEW permite que partes de códigos e até códigos inteiros sejam convertidos em funções, que podem ser aplicadas. No caso, a caixa vermelha corresponde às operações aritméticas que, recebendo os parâmetros necessários, retorna a velocidade estimada.

Essa caixa corresponde ao seguinte código, referente ao calculo das velocidades e ao sinal enviado pelo sensor hipotético (função '*Random*', com símbolo de dados). Esse código também utiliza algumas funções matemáticas de exponencial, facilmente encontradas com uma busca na biblioteca da plataforma.

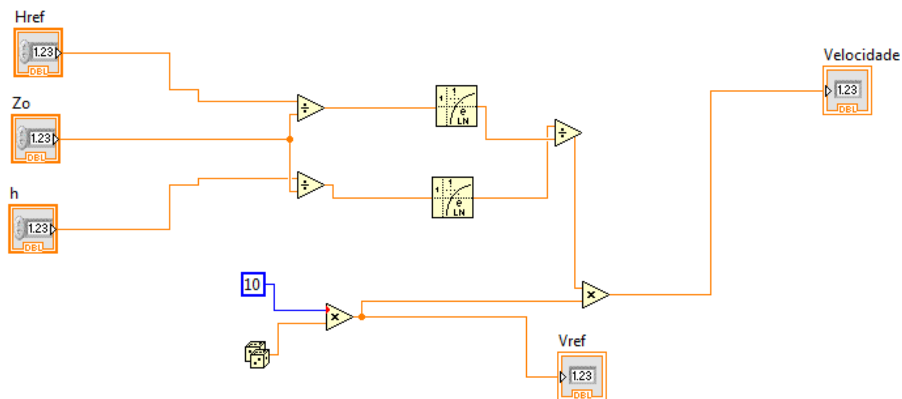


Figura 41 - SubVI de Extrapolação das Velocidades

Em terceito lugar salienta-se o uso de Shift Registers, muito úteis na operação de Loops. Com eles, possibilita-se que o valor de um dado seja carregado para a próxima iteração de um *loop*. Note que o valor foi inicializado em '*FALSE*' e o valor que chega na seta para cima iteração 'n' será o valor que sai da seta para baixo na iteração 'n+1'.

Finalmente, uma ferramenta muito útil utilizada foi o auto indexing. Ativado o *Auto-Indexing*, o dado de saída passa a ser o *array* dos dados resultantes de cada iteração do loop. Note que, assim, a saída sempre será de uma dimensão maior em uma unidade que os dados de dentro do *loop*.

Nesse caso, dentro do *loop* tinha-se os valores de cada velocidade, ao longo do tempo. Com o auto indexing, a saída do loop passa a ser um array com o registro de todos os valores calculados, a partir do qual, com a função Max. Min. será obtida a velocidade máxima. Note, no entando, que essa saída só é entregue ao fim do funcionamento do *loop*, ou seja, calcula-se o valor máximo no fim da execução.

Há ainda que se reparar na utilização de uma estrutura de casos para garantir que, uma vez extrapolada a velocidade admissível, o LED de alerta se mantenha acionado permanentemente. No caso '*FALSE*' o LED ainda não foi acionado, e se efetua uma comparação da velocidade calculada com a admissível. No entando, uma vez que essa comparação retorne '*TRUE*', graças aos *Shift Registers*, o

caso acionado sempre será ‘*TRUE*’. E nesse caso o *loop* retorna sempre o valor de ‘*TRUE*’. As figuras a seguir ilustram a lógica.

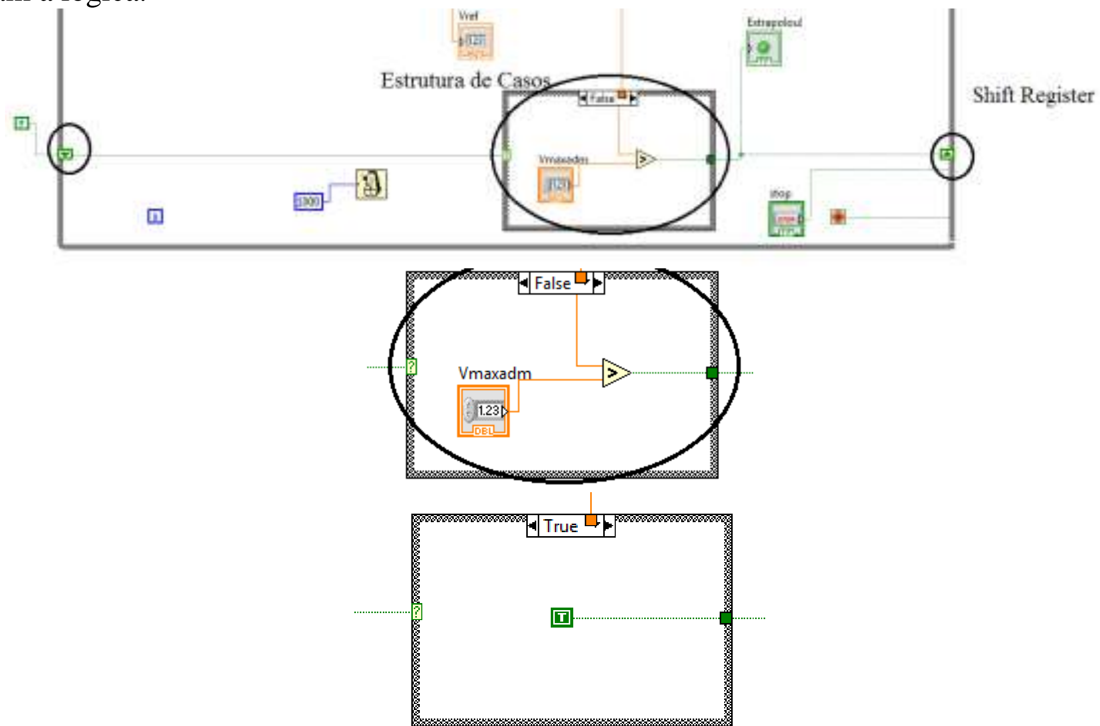


Figura 42 - Estruturas de Casos Para Manter Alarme

A primeira figura corresponde à estrutura geral, a segunda ao caso false e a terceira ao caso true.

5. OPERAÇÕES BÁSICAS FILE INPUT-OUTPUT, EXECUTÁVEIS, ERROS, PATHS E SMTP

Muitas vezes é oportuno que um programa execute tarefas cujo efeito e ultrapassa o nível do VI. Isso é, o aplicativo faz modificações no computador, cria, manipula, edita e lê arquivos e documentos; chama executáveis, programados em outras plataformas por motivos diversos; ou até envia informações pela internet.

Essa seção apresenta noções e exemplos de operações básicas envolvendo a ação de VIs em LabVIEW sobre domínios e elementos externos. Tratam-se de operações do tipo Input-Output (I/O) de diversas naturezas. A aplicação dessas operações abrange a manipulação e execução de arquivos, envio de e-mails, geralmente exigindo conhecimento de gerenciamento de erros e *paths*.

5.1.REFERENCIANDO ARQUIVOS – FILE PATHS E REFNUMS

Quando se deseja fazer referência a um arquivo qualquer num código em LabVIEW existem duas maneiras básicas de fazê-lo. Cada uma das maneiras relaciona-se a um tipo de variável, que por sua vez fluí por um tipo de cabo no diagrama de blocos.

5.1.1. PATHS ABSOLUTOS E RELATIVOS

A primeira maneira está relacionada aos '*Paths*', os quais podem ser relativos ou absolutos. De qualquer maneira, paths são variáveis semelhantes aos *strings*, mas que possuem a função exclusiva de indicar, de maneira explícita, o endereço de um dado objeto – em geral um arquivo - no sistema de arquivos do sistema operacional onde o código está sendo executado. De fato, o Linux Information Project define paths como "*o endereço de um objeto (por exemplo, arquivo, diretório ou link) em um sistema de arquivos*."[3] (Linux Information Project, 2013, tradução livre.) Assim, se conectados a certas funções, os '*Paths*' permitem a localização dos arquivos desejados.

Em suma, '*Paths*' são variáveis de texto que referenciam arquivos.

É importante notar, no entanto, que o formato no qual Paths são escritos diverge de um sistema operacional para outro. Por exemplo, na divisão de diretórios o sistema operacional Windows utiliza contra barras, ao passo que o Linux utiliza barras e o Mac dois pontos. Essas divergências devem ser levadas em conta na escrita do código, mas, de maneira geral a sintaxe para a escrita de paths é simples e de fácil compreensão.

Os VIs utilizados nos exemplos desse texto foram programados em um Windows OS, e portanto será essa a sintaxe utilizada. Ainda assim, não há grande perda de generalidade, já que a conversão é muito simples, havendo inclusive textos de auxilio no próprio LabVIEW, nas opções de 'Help'.

O primeiro tipo de *path* são *paths* absolutos. Nesse caso, estão presentes todas as informações necessárias para localizar cada arquivo. Isso é, consta informação do drive relativo, de toda a sucessão de diretórios, de nível mais alto ao mais baixo, onde o arquivo está sendo guardado, bem como o nome e extensão do arquivo propriamente dito. Toda essa comunicação, entretanto, é realizada de maneira padronizada.

Por exemplo, no caso do computador em questão, o path do executável do LabVIEW é *C:\Program Files (x86)\National Instruments\LabVIEW 2011\LabVIEW.exe*. Ou seja, o *path* absoluto de um dado arquivo é dado por seu drive seguido de ":", seguido da sucessão de diretórios, de nível mais alto a mais baixo, cada um antecedido por "\", e finalizado pelo nome do arquivo propriamente dito antecedido de "\", mais sua extensão.

Nota-se assim que um *path* absoluto fornece o endereço de um arquivo iniciando no nível mais alto do sistema. Um *path* relativo por sua vez, seria um *path* que oferece a localização de um arquivo de maneira relativa a uma localidade do sistema, em geral, o diretório atual (onde o usuário ou programador estão trabalhando) (Linux Information Project, 2013). Isso é, um *path* relativo de um dado arquivo necessariamente será um trecho do *path* absoluto do mesmo. Por exemplo, o *path* do LabVIEW.exe relativo ao diretório Program Files (x86) seria *National Instruments\LabVIEW 2011\LabVIEW.exe.*

Um path relativo, em si, não oferece informação completa para localizar um arquivo. Entretanto, pode ser muito útil pra fazer referência a arquivos de maneira a tornar um VI compatível com diversas máquinas, com estrutura de arquivos diferentes. Assim, se unido com informações, por exemplo, dadas pelo usuário pode-se montar o path absoluto, a partir do path relativo.

Na plataforma LabVIEW, paths absolutos são um tipo específico de variável. Semelhante a um *string*, seus cabos são verde escuro ondulado.

Figura 43 - Path Absoluto no Diagrama de Blocos

[3] - A path is the address of an object (i.e., file, directory or link) on a filesystem.

Paths relativos, por sua vez, podem ser convertidos em paths absolutos com a função '*Build Path*'. Nesse caso, basta conectar o path do diretório – em formato path – e o path relativo – em formato *string* - à função, como mostra o diagrama de blocos referente.

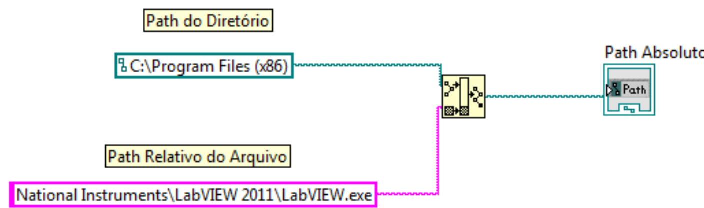


Figura 44 - Path Absoluto a Partir de Path Relativo

Um exemplo de flexibilização do código, com o emprego de paths relativos pode ser observado na figura que segue.

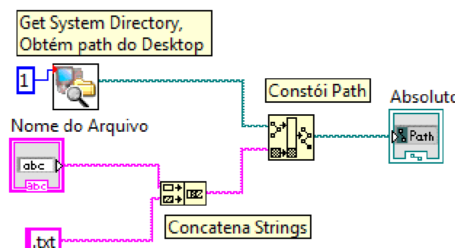


Figura 45 - Flexibilização do Código por Path Relativo

No caso, obtém-se o path de um arquivo com um dado nome entrado pelo usuário e localizado no Desktop. Nota-se, no entanto, que esse path será válido para qualquer máquina e qualquer sistema operacional, pois foi obtido a partir de seu *path* relativo e do path do Desktop, obtido com a função '*Get System Directory*'.

Com efeito, paths absolutos, de uma maneira geral, são utilizados para fazer a primeira referência a um dado arquivo, em um VI. Como se verá adiante, no entanto, dada sua estrutura, para referenciar arquivos no decorrer do funcionamento do VI pode ser interessante o emprego de *refnums*.

5.1.2. FILE REFNUMS

Por um lado, paths absolutos são muito úteis por fazerem referência precisa a um arquivo em um dado sistema sem serem relativos a nada. Isso é, seu significado independe, por exemplo, do VI onde é empregado, ou da execução do programa.

Entretanto, algumas desvantagens são notadas. Em primeiro lugar, variáveis do tipo *path*, por serem semelhantes a *strings*, exigem mais espaço que, por exemplo, variáveis numéricas. Em adição, diversos problemas poderiam ser gerados caso se quisesse continuar fazendo referência a um mesmo arquivo depois de sua renomeação ou realocação no sistema. Nesse sentido, torna-se oportuno o emprego de *file refnums*.

A principal vantagem, no entanto, dos *file refnums* é que possuem a capacidade de carregar outros atributos dos arquivos, como uma posição em um arquivo de texto, como se verá adiante, ou até mesmo grau de acesso dos usuários. Isso é muito relevante na leitura de arquivos de texto.

File Refnums tem seus cabos representados em verde escuro retilíneo. *File refnums* são variáveis que associam um número a um dado arquivo do sistema, isso é, são apontadores temporário para

objetos. De fato, trata-se de uma variável que um VI entende como referência a um dado arquivo, de maneira a substituir o *path* do mesmo e alocar memória do sistema para esse dado objeto.

Uma consideração importante a ser feita, no entanto, é quanto à natureza relativa dos *file refnums*. Com efeito, segundo a página *'Refnum Controls'* do suporte da Nationa Instruments (2013), um *refnum* configura um apontador temporário para um objeto, valendo apenas enquanto o dado objeto se mantém aberto. Ou seja, no ato do fechamento do objeto, o *refnum* é desassociado do mesmo, tornando-se obsoleto. Por outro lado, a nova abertura do mesmo objeto acarreta na geração de um novo *refnum*, diferente do anterior, mesmo que aponte o mesmo objeto. Isso é, quando um VI que possuí *refnums* é executado, o mesmo gera *refnums* para cada arquivo solicitado, por exemplo, com a função de abrir arquivos. Entretanto, esses números possuem valor de referência aos arquivos apenas nesse VI, nessa execução e nessa abertura.

Assim é importante ter cuidado quando se deseja referenciar arquivos em múltiplos VIs, ou em múltiplas execuções, sendo por vezes favorável o emprego de *paths*.

Ainda assim, em geral o que ocorre é o emprego de *file refnums* no decorrer do código. De fato, muitas funções que aceitam dados de entrada no formato *path* aceitam também, ou apenas *refnums*. Por outro lado, muitas funções que recebem como dados de entrada *paths* apresentam como dados de saída *refnums*, direcionando o programador no sentido de emprega-los. É o caso da função '*Open/Creat/Replace File*', como se verá adiante.

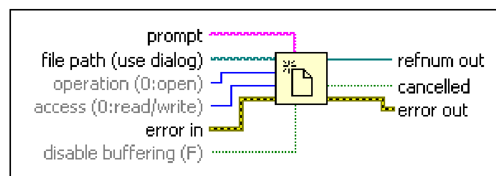


Figura 46 - File Paths e Refnums

É possível ainda converter *refnums* em *paths*, com a função '*Refnum to Path'*, bem como converter *paths* em *strings* ou *strings* em *paths*, com as funções '*Path to String*' e '*String to Path*', respectivamente. O análogo a converter um *path* em um *refnum* seria abri-lo no diagrama de blocos, com uma função como '*Open/Creat/Replace File*', alocando memória do sistema para o *refnum*.

5.2. COMUNICANDO E TRATANDO FALHAS – ERRORS

Dado o número maior de variáveis envolvidas – funcionamento do sistema operacional, existência dos paths comunicados, etc. – muitas vezes as funções I/O não apresentam o comportamento esperado. Nesse sentido, é muito útil monitorar e controlar possíveis erros ocorridos na execução das funções dessa natureza, do contrário, tudo o que se saberá é que o VI não está funcionando adequadamente (National Instruments, 2013)

5.2.1. TRATAMENTO AUTOMÁTICO OU CUSTOMIZADO

Segundo o suporte da National Instruments (2013) para tratamento de erros, "Como padrão, na eventualidade da ocorrência de um erro o LabVIEW suspenderá a execução do VI que está rodando, destacando a função ou SubVI onde o erro ocorreu, além de apresentar uma caixa de diálogo do erro"[4].

[4] - By default, LabVIEW automatically handles any error when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying an error dialog box.

Entretanto, muitas vezes esse comportamento não é favorável à finalidade do programa. Pode ser que, para dado aplicativo seja interessante executar uma rotina de segurança antes da suspensão do programa, ou ainda que o mesmo não seja interrompido. É esse o caso, por exemplo, de programas que operam máquinas perigosas. Há ainda muitos casos onde a identificação de um erro faz parte do funcionamento natural de um aplicativo. Por exemplo, um programa que interrompe uma ação, e prossegue sua execução quando uma função que procura uma pasta não a encontra, retornando erro.

Assim, existem maneiras de evitar que o VI seja suspenso quando da ocorrência de um erro, optando por analisar e tratar informações do mesmo.

Caso se deseje que, em um dado VI, nenhum erro seja tratado automaticamente da maneira padrão, já mencionada, basta desativar a opção '*Enable automatic error handling*'.

No entanto, o que muitas vezes se faz em nos códigos de LabVIEW é desabilitar o tratamento automático de erros de cada função ou SubVI de maneira individual.

Diversas funções, principalmente as do tipo I/O apresentam *inputs* (*error in*) e *outputs* (*error out*) de erros. Esses terminais são compatíveis com o tipo de variável *error*, de cor marrom tracejado, e tratam-se de *clusters* de erro, ou seja, 'pacotes' de variáveis de diversas naturezas que oferecem informações sobre o erro.

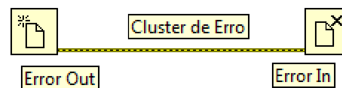


Figura 47 - Error In e Error Out

Note que é sempre possível fazer com que um SubVI qualquer possua também terminais de '*error in*' e '*error out*'. Assim, quando se quer tratar de forma não automática o erro de alguma função ou SubVI, basta conectar seu terminal de '*error out*' ao terminal de '*error in*' de alguma outra função. Nesse caso o código automaticamente entende que o programador deseja tratar o erro de maneira customizada.

5.2.2. A ESTRUTURA DO ERRO

Algumas funções comunicam seus erros simplesmente por códigos numéricos. Entretanto, a maior parte das funções, principalmente as I/O, utilizam clusters de erro, obtidos pelos terminais de '*error out*'.

O cluster de erro liberado pelo terminal '*error out*' de uma dada função apresenta uma estrutura própria e padronizada, para que seja compatível com funções já existentes ou criadas pelo programador. Ordenando a informação fornecida, portanto, é possível que funções e SubVIs de tratamento de erros tenham uso universal.

De fato, o cluster de erro é um pacote com três elementos de informação, ou seja, três variáveis, de tipos distintos. São eles '*status*', '*code*' e '*source*'. A figura a seguir representa uma constante de cluster de erro, com cada um de seus elementos.

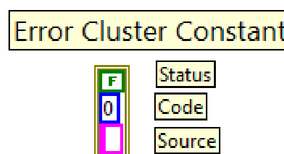


Figura 48 - Cluster de Erro

De fato, cada um desses elementos tem os seguintes significados:

O primeiro elemento do cluster é o '*status*'. Trata-se de uma variável booleana que retorna 'TRUE' caso um erro tenha ocorrido.

O segundo elemento do cluster é seu 'code'. Trata-se de uma variável de número inteiro a qual codifica o erro ou advertência ocorridos. Caso a variável apresente valor diferente de '0', algum erro ou advertência ocorreu. Cada função apresenta na sua página de apoio (localizada na opção '*help*' do LabVIEW) informações detalhadas sobre a interpretação do código de cada um de seus erros.

O ultimo elemento é um *string* que indica onde o erro ocorreu.

Assim, se o cluster retorna status '*TRUE*' e código diferente de zero ocorreu algum erro. Caso retorne status '*FALSE*' e código diferente de zero, alguma advertência ocorreu. Caso retorne status '*FALSE*' e código '0' então nada ocorreu e as funções operaram normalmente.

5.2.3. FORMAS DE LIDAR COM ERROS

Assim como outras variáveis no LabVIEW, os erros devem ser tratados como dados em fluxos. Como já explicado, ao conectar o terminal '*error out*' de uma função ao '*error in*' de outra, o código interpreta que o programado lidará com o erro. Assim, o programa segue sua execução, sem ter realizado, entretanto, a função na qual ocorreu erro adequadamente.

Existem diversas formas de lidar com erros. Em muitas aplicações, é desejável que o código continue executando, e ao seu fim, exista um '*loop*' que informa ao usuário os erros ocorridos. Em outras aplicações o que se deseja é que, caso ocorra um erro, algumas ações não sejam executadas. Nesse caso há que se pensar em estruturas de casos que utilizem informação de erros.

O módulo padrão do LabVIEW já apresenta funções que auxiliam o usuário a interpretar e comunicar erros, por exemplo, com caixas de diálogo. Tais funções encontram-se na paleta *'Dialog & User Interface*'. Combinando essas funcionalidades com o conhecimento da sintaxe dos clusters de erro, bem como estruturas de casos e *loops* é possível criar um número ilimitado de rotinas de tratamento de erros customizadas.

De fato, ao ligar um erro a um terminal de condição de um *loop*, apenas seu valor de status (booleano) é lido. Estruturas de casos com erros conectados a seus terminais também apresentarão dois casos, '*error*' e 'no *error*'.

5.3.FILE INPUT – OUTPUT PARA ARQUIVOS DE TEXTO

Uma das maneiras mais simples de registrar resultados de um aplicativo para referência futura é escrevendo-os de maneira automática em um arquivo de texto. Com efeito, registrar informações em um arquivo de texto exige domínio da manipulação de documentos de tal natureza – isso é, sua abertura, criação, substituição e eliminação, tudo de forma automática. Uma vez que o programador tem capacidade de realizar tais tarefas, torna-se possível implementar rotinas de edição e leitura desses documentos.

5.3.1. CRIANDO, ABRINDO, SUBSTITUÍNDO, FECHANDO E DELETANDO ARQUIVOS DE TEXTO

As operações de criação, abertura e substituição de arquivos – não necessariamente de texto – se dão pela mesma função, '*Open/Create/Replace File*'. Com uma entrada numérica define-se qual operação deve ser realizada. Entra-se ainda com o *path* do referido arquivo, ou, se não se conecta cabo a

esse terminal, o programa lança uma caixa de diálogo para o usuário selecionar o arquivo. Há um terminal de saída que entrega o refnum do arquivo, usado para outras operações com esse elemento, como por exemplo, edição e adição de dados. Finalmente, sendo uma função I/O, há terminais de *error in* e *out*.

Nota-se que essa função basicamente converte um *path* de um arquivo preexistente ou não para um *refnum*. Ou seja, basicamente a operação que se está realizando é a alocação de memória para um *refnum*, referenciando um arquivo a ele.

O exemplo abaixo recebe o path de um diretório, o nome e tipo de um arquivo, monta um path para o arquivo e, em seguida, o cria. Adicionalmente, libera o *refnum* do arquivo, convertido em *path* – com a função adequada), o qual é entregue no terminal de saída.

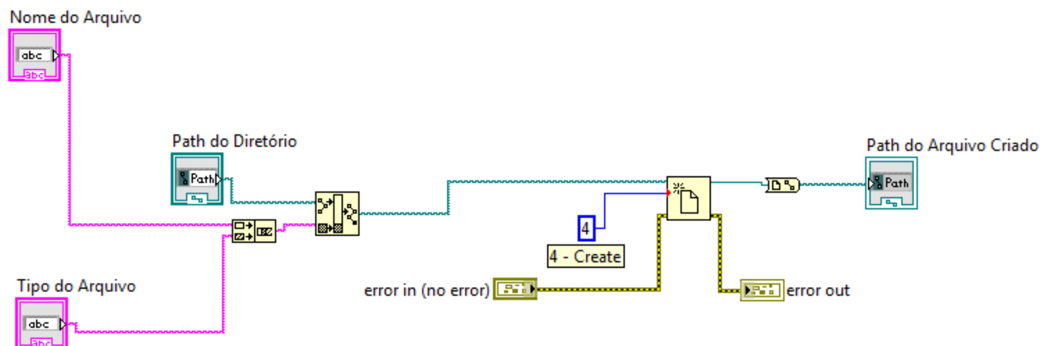


Figura 49 - Exemplo Refnums I

Fechar um arquivo constituí tarefa ainda mais simples. Para tal, basta aplicar a função '*Close File*', e entrar com o *refnum* do arquivo. O *path* é entregue, e há terminais de erro. O exemplo abaixo cria um arquivo e em seguida o fecha.

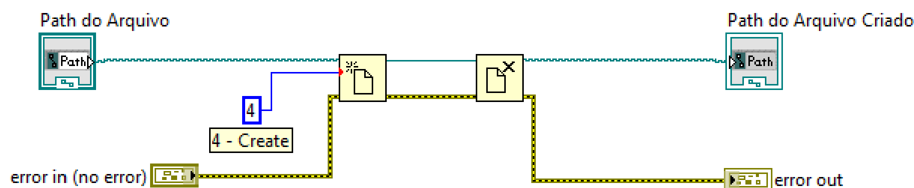


Figura 50 - Exemplo Refnums II

Para excluir uma arquivo utiliza-se a função '*Delete*', a qual recebe o path de uma arquivo, ou chama caixa de dialogo para o usuário, indicando o arquivo a ser excluído. Podem ainda ser deletadas pastas ou diretórios inteiros. A função apresenta ainda terminais de erro e uma série de outras opções, as quais podem ser exploradas pelo programador.

5.3.2. EDIÇÃO DE ARQUIVOS DE TEXTO

Como já discutido, a edição de arquivos de texto em geral representa uma opção relativamente simples e segura para armazenar dados. Em geral, para escrever em arquivos de texto, utiliza-se a função '*Write to Text File'*.

Tal função simplesmente recebe o *path* ou *refnum* do arquivo desejado por um terminal de entrada ou, alternativamente, por uma caixa de diálogo. Recebe ainda um *string* ou *array* de *strings* e os escreve no arquivo de texto.

Nesse ponto algumas observações sobre como a escrita é efetuada são importantes. A referida função aceita como endereço do arquivo em questão tanto *refnums* quanto *paths*.

Segundo o suporte da National Instruments (2013) para tal função, quando se conecta um path, a função, antes de efetuar a escrita, abre ou cria um arquivo de texto com o path indicado. Nesse caso, qualquer conteúdo previamente escrito no arquivo é eliminado, passando a haver apenas os novos elementos. Quando se conecta um refnum, por outro lado, a variável guardará também a posição atual do arquivo. Nesse caso, a escrita começa (criando ou substituindo caracteres) a partir da dada posição.

Lidar com a posição do arquivo tem diversas particularidades. Por exemplo, logo após abrir o arquivo, a posição obtida é o primeiro caractere. Por outro lado, após a aplicação da função de escrita, o refnum entregue carrega a posição imediatamente posterior ao ultimo caractere escrito. Assim, existem diversas funções que modificam a posição de referência.

A maneira mais segura de gerenciar a posição, assim, é aplicar a função '*Set File Position*'. Tal função recebe um *refnum*, e variáveis numéricas de forma a ser possível obter um *refnum* de saída do mesmo arquivo, mas na posição desejada. No caso, define-se uma posição inicial, que pode ser o início, fim ou posição atual do arquivo, bem como um *offset* de caracteres da referência onde será demarcada a nova posição. Por exemplo, para concatenar novas informações ao arquivo, utilizar-se-ia um *offset* nulo a partir do fim do texto.

A seguir constam alguns exemplos de programas que escrevem em arquivo de texto, com seu respectivo resultado. Os pares ilustram como a ordem das funções, bem como posições de referência diversas geram resultados diferentes.

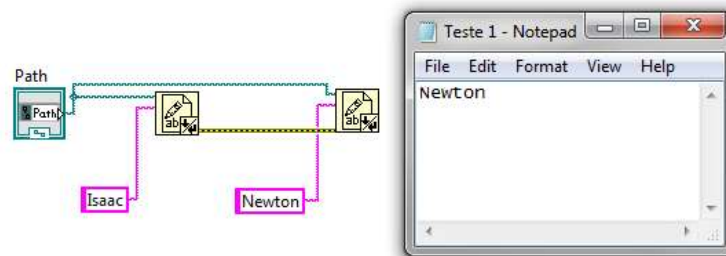


Figura 51 - Escrita em Arquivo I

Nesse caso, como se referenciou o arquivo pelo path, 'Isaac', anteriormente escrito foi deletado, restando apenas Newton.

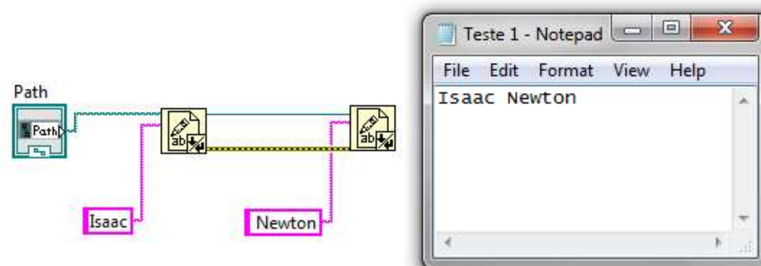


Figura 52 - Escrita em Arquivo II

Nesse caso, utilizou-se o refnum entregue pela função de escrita (fim do arquivo), acarretando resultado adequado.

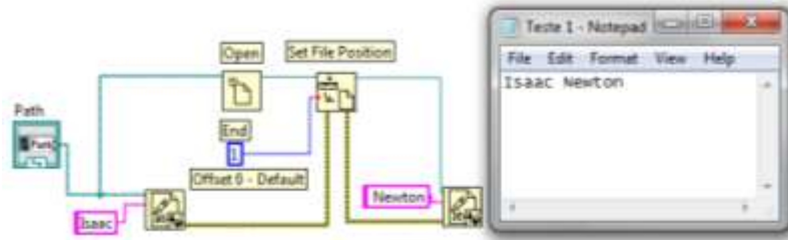


Figura 53 - Escrita em Arquivo III

Novamente o resultado é adequado. Nesse caso, após escrever 'Isaac' abriu-se o arquivo, definindo como posição o fim, posição essa recebida pela segunda função de escrita.

Finalmente vale citar a funcionalidade do LabVIEW de escrever '*spreadsheets*'. Trata-se simplesmente de uma maneira oportuna de organizar *arrays*, a qual facilita a escrita e leitura de arquivos, dada a padronização. Basicamente, o *array* de *strings* é convertido num único *string* com '*Tabs*' ou outro delimitador pré definido separando colunas, pulando linhas do texto entre as linhas, e índices de páginas para *arrays* de dimensão maior que 2. Para tal utiliza-se a função '*Array to Spreadsheet String*'.

5.3.3. LEITURA E PROCESSAMENTO DE ARQUIVOS DE TEXTO

Ler e processar arquivos de texto pode ser uma tarefa muito mais difícil do que escrevê-los. Em geral, a dificuldade na leitura de um arquivo será diretamente proprocional a organização e padronização com a qual a escrita foi feita.

A ferramenta básica para a leitura de arquivos de texto é a função '*Read Form Text File*'. Recebendo um path ou refnum de um arquivo de texto, a função entrega um *string* com o texto lido, ou ainda um *array* de *strings* com as linhas lidas, caso escolha-se a opção de ler linhas. É possível ainda especificar o número de linhas ou caracteres a serem lidos. O exemplo abaixo ilustra a aplicação da função.

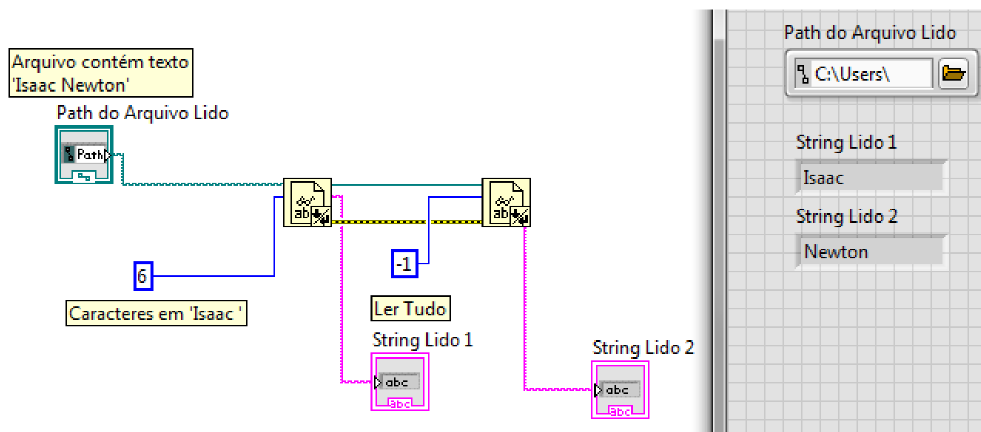


Figura 54 - Leitura de Arquivos de Texto

Como se nota, os *refnums* continuam seguindo a lógica supracitada. Isso é, guardam também uma posição de referência. Nota-se, no caso, que a primeira função de leitura entregou em seu refnum a posição imediatamente posterior ao último caractere lido.

Nesse exemplo foi muito fácil separar as palavras 'Isaac' e 'Newton' pois sabia-se exatamente o que estava escrito no arquivo. Entretanto, geralmente não é isso o que ocorre e, pelo contrário, deseja-se

descobrir o conteúdo dos arquivos. Assim, uma vez que se lê um arquivo, e se obtêm seu *string* ou *array* de *strings*, é necessário interpretá-lo para adquirir os dados.

A transformação do *string* em dados de fato pode ser feita de inúmeras maneiras. Uma maneira que garante, ao mesmo tempo, simplicidade e generalidade e o emprego da função '*Match Pattern*'. Essa função busca em um dado *string*, a partir de uma posição especificada, um padrão se sequência se caracteres definido pelo usuário. Ao encontrar o padrão, a função divide o *string* em 3 *substrings* – o *substring* de correspondência, o *substring* anterior à correspondência e o posterior, entregando-os. Em adição, fornece a posição imediatamente posterior à correspondência.

A definição do padrão a se buscar se dá por meio de um *string*, onde se descreve, em uma linguagem específica, a sequência de caracteres buscada. Detalhes sobre esse tipo de codificação podem ser encontradas nos manuais de LabVIEW inclusos no aplicativo.

Por exemplo, pode-se definir para procurar uma palavra em específico, ou uma palavra qualquer com um dado número de letras, ou até mesmo um conjunto de frases. Nesse ponto, nota-se que, para codificar o padrão de busca é essencial considerar que tipo de dados se está buscando e como os dados foram organizados no arquivo de texto.

A seguir, apresenta-se um programa que emprega os conecitos explicados. No caso, o VI recebeu o path do seguinte arquivo de texto:

Figura 55 - Arquivo de Texto a Ser Lido

Esse programa objetiva passar os dados em arquivo de texto para o programa, no formato de dois arrays, sabendo de antemão que os nomes estão na segunda linha e as notas respectivas na terceira. Dentre os *arrays*, sejam um de *strings*, contendo nomes dos alunos e um de números, contendo as notas. O VI apresenta, assim, o seguinte diagrama de blocos e resultados no painel frontal:

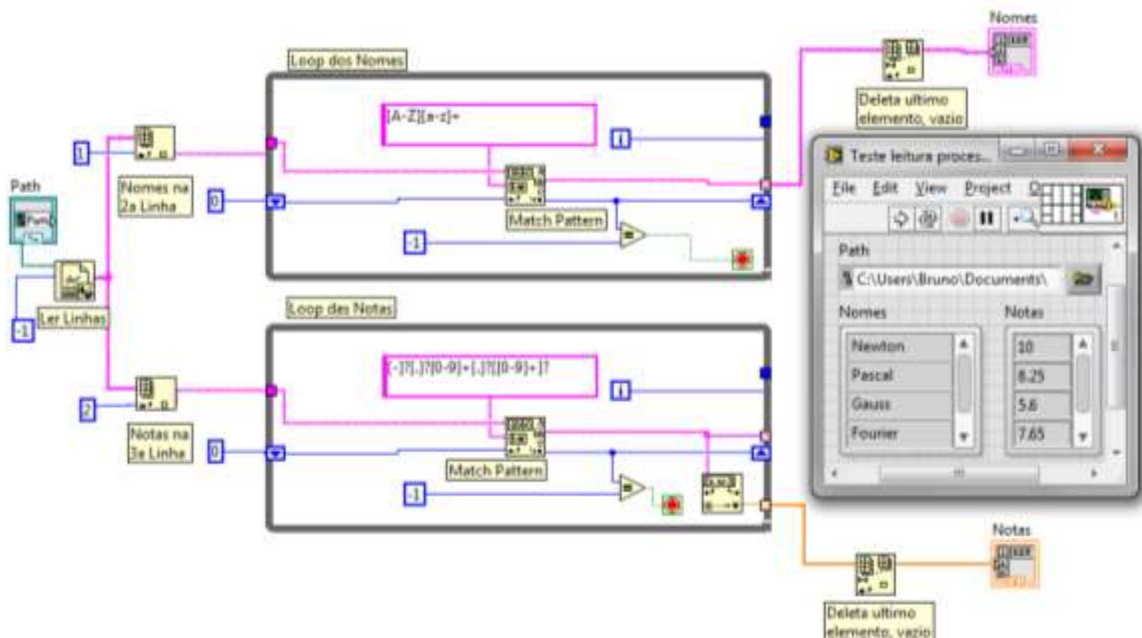


Figura 56 - Leitura e Tradução do Arquivo de Texto

Inicialmente o programa recebe o path do arquivo de texto e lê todas suas linhas, entregando um array de strings das mesmas. A partir daí o fluxo divide-se em dois ramos análogos. Um deles processará a segunda linha – índice 1 – a qual sabe-se que contém os nomes dos alunos e o outro a terceira – índice 2 – a qual contém notas.

Em cada ramo existe um *loop* que recebe a linha respectiva. Em cada iteração, aplica-se a função '*Match Pattern*' no *string*, com o padrão buscado codificado. No caso, para nomes o padrão utilizado é o de uma letra maiúscula seguida de uma sequência de minúsculas. Para notas, por sua vez, possível sinal de menos ou ponto seguido de uma sequência de dígitos seguido de possível ponto e outra sequência de dígitos. Além disso, utilizaram-se *Shift Registers* para ganrantir que a busca se inicia sempre logo a seguir da ultima correspondência.

Quando não for possível encontrar correspondência, a função '*Match Pattern*' entrega o índice de offset -1. Quando isso ocorre, interrompe-se o funcionamento do *loop* e os *arrays* são liberados (com uso de *auto-indexing*). Finalmente, excluí-se o ultimo elemento de cada *array*, vazio e correspondente à ultima iteração, na qual não houve correspondência.

A leitura de arquivos de texto, como se vê, pode se mostrar relativamente desafiadora, especialmente se não se conhece bem a formatação do arquivo a ser lido. Nesse sentido, ressalta-se novamente o recurso de escrever arquivos no formato de '*spreadsheet*', de forma a ter-se formatação padronizada.

5.4.EMPREGANDO EXECUTÁVEIS

Por mais flexível que a plataforma LabVIEW seja, muitas vezes é necessário o emprego de programas executáveis desenvolvidos em outras linguagens e plataformas em uma rotina. É possível, dessa maneira, condiconar um diagrama de blocos a executar um programa qualquer de extensão '.exe'.

Para relaizar tal operação, basta aplicar um VI incluso na plataforma chamado '*System Exec*', conectando o *path* do executável ao SubVI. Dentre outras opções de customização, é possível escolher se o VI aguarda o fim das operações do executável para continuar sua própria execução, ou ainda definir se o executável rodará minimizado.

5.5.SIMPLE MAIL TRANFER PROTOCOLS

Segundo o documento mais atual de especificação do SMPT, o RFC5321 (2013), o Simple Mail Transfer Protocol tem como objetivo transferir mensagens eletrônicas de maneira confiável e eficiente. De fato, deve estabelecer-se um protocolo que seja capaz de enviar informação pela rede com o menor ruído e número de falhas possíveis, comunicando, ainda, a ocorrência desses erros, caso aconteçam.

Em um sistema computadorizado de monitoramento, a central que envia mensagens de alerta e acompanhamento configura-se no papel de 'Cliente', dentro da estrutura geral do SMTP. De fato, por definição do RFC5321 (2013), o cliente SMTP responsabiliza-se pela transferência de mensagens de e-mail para um ou múltiplos servidores SMTP, reportando falhas, caso existam.

Assim, no contexto de 'Cliente', para programar o envio de uma mensagem, é necessário montar uma mensagem de e-mail, identificando o servidor e os endereços da conta que envia e que recebe, bem como o conteúdo da mensagem. Adicionalmente, para alguns servidores, é necessário realizar autenticação. Finalmente, envia-se a mensagem ao servidor, que se responsabilizará pelo envio aos destinatários finais ou intermediários. Além disso, é necessário que o programa seja capaz de comunicar erros ocorridos nesse processo.

No caso, utilizou-se o servidor GMAIL, de sorte que a autenticação torne-se necessária. Segundo a plataforma de suporte ao usuário da National Instruments, no entanto, "*Quanto ao LabVIEW 8.6, não*

há suporte interno para enviar e-mail via servidores que requerem autenticação; os VIs SMTP inclusos só são compatíveis com servidores abertos. A plataforma Microsoft.NET fornece, no entanto, uma interface relativamente direta para o envio de e-mails com autenticação, com a qual o LabVIEW pode estabelecer interface."[5] (National Instruments, 2013, tradução livre). De fato, o envio de e-mails, nesse trabalho, será desenvolvido por tais vias.

Criou-se, assim, um VI relativamente genérico – podendo, assim, ser utilizado com bastante flexibilidade como SubVI – que envia e-mails pelo servidor GMAIL através de uma conta criada com apenas esse propósito. O VI é dividido basicamente em duas partes.

A primeira simplesmente verifica se é possível estabelecer conexão adequada com a internet.

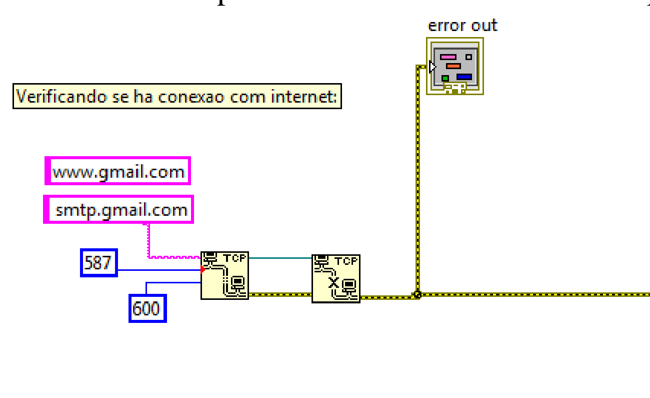


Figura 57 - Verificação de Conexão como Servidor

Aplica-se a função '*Transmission Control Protocol Open Connection*' para estabelecer uma conexão com o servidor SMTP GMAIL (smtp.gmail.com). Em seguida, para o *refnum* dessa conexão, aplica-se a função '*Transmission Control Protocol Close Connection*' para encerrar a conexão. O erro dessas operações é extraído. Se não resultarem erros, a segunda parte do VI é acionada por meio de uma estrutura de erros.

A segunda parte do VI consiste de propriamente montar e enviar mensagens de email pelo servidor SMTP GMAIL, por meio de interface com a plataforma Microsoft.NET.

[5] - As of LabVIEW 8.6, there is no native support for sending Email via servers that require authentication; the built-in SMTP vis only work with open servers.The Microsoft .NET platform does, however, provide a fairly straightforward interface for sending authenticated Email, which LabVIEW can interface with.

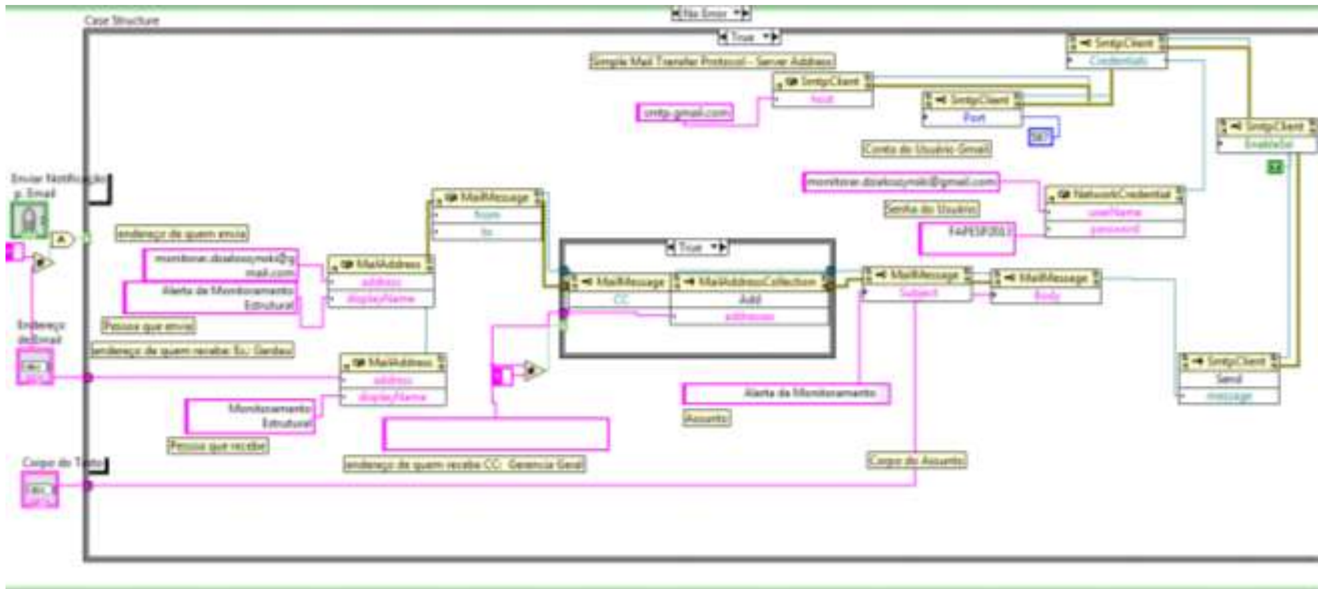


Figura 58 - Montagem da Mensagem de Email

A National Instruments oferece gratuitamente em seu site '*templates*' de rotinas SMTP. No caso, utilizou-se um deles como base, efetuando as customizações necessárias.

A estrutura pode aparentar complexa em primeira instância, mas as operações que realiza são relativamente simples. Basicamente, verifica-se se uma variável booleana, correspondente à confirmação do usuário de desejo de envio de e-mail retorna '*TRUE*'. Em seguida, uma série de '*Constructor Nodes*' e '*Property Nodes*' são aplicados para agregar a um *refnum* todas as informações do e-mail (destinatário, servidor, credenciais, etc.). Por fim, utiliza-se um '*Invoke Node*' para enviar a mensagem.

5.6. EXEMPLO DE CONSOLIDAÇÃO – CARACTERÍSTICAS GEOMÉTRICAS DE UMA POLIGONAL PLANA

Para consolidar os conceitos da macroárea de operações Input-Output, manipulação de arquivos de texto e SMTP foi desenvolvido o programa abaixo apresentado, intitulado 'Características Geométricas de uma Poligonal Plana'.

Figura 59 - Características Geométricas de Uma Poligonal Plana, Painel Frontal

O referido aplicativo recebe o nome e as coordenadas de uma poligonal, no sentido dextroso, apresentando instantâneamente sua representação gráfica. Em seguida recebe o path de um executável específico, que, recebendo dados em arquivos de texto, calcula as características geométricas da poligonal, entregando-as num arquivo de texto. Assim, quando a tecla 'ATIVAR' é pressionada, para a poligonal indicada pelo usuário, são calculadas as características geométricas, como coordenadas do baricentro, área, momentos de inércia, momentos de inércia nos eixos principais, produto de inércia, bem como a inclinação do eixo principal, apresentadas no painel frontal. Adicionalmente, a poligonal e um de seus eixos principais são representados graficamente. O programa faz, ainda, um acompanhamento dos processos realizados, indicando possíveis erros.

Para realizar tais operações foi criado um código gráfico, dividido básicamente em 4 operações. São elas: 'Inicializações', 'Escrita dos Dados de Entrada', 'Chamar o Executável' e 'Ler o Arquivo Gerado e Apresentar Resultados'. Todas essas operações constam dentro de um *loop* de funcionamento ininterrupto, para que o usuário possa utilizar sucessivamente o programa, sem interromper seu funcionamento.

5.6.1. INICIALIZAÇÕES

As inicializações são as primeiras operações a serem realizadas no loop. Isso é garantido pela lógica de fluxo de dados, devido à dependências naturais.

Uma das operações a ser realizada nas inicializações é a plotagem da poligonal.

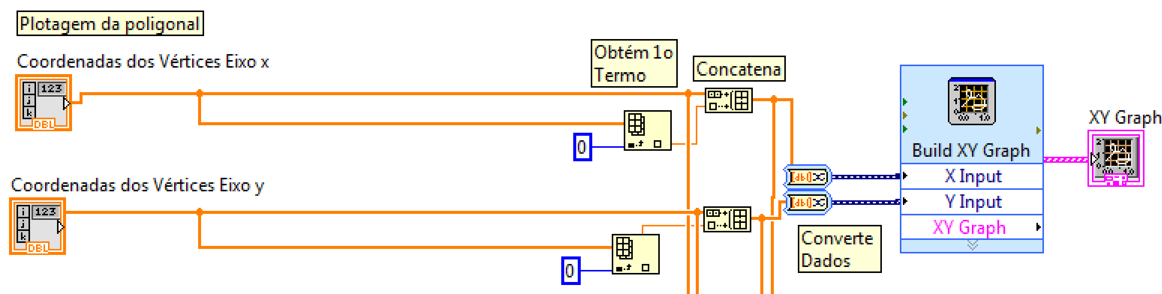


Figura 60 - Plotagem dos Dados

As coordenadas da poligonal, definidas pelo usuário tomam dois caminhos possíveis. Existem cabos que seguem para etapas posteriores do programa, como apresentação de dados, ou escrita do arquivo de texto do executável. Outros cabos prestam-se propriamente à plotagem instantênea da poligonal.

Como se vê, toma-se o array das coordenadas x e y, concatenando uma cópia do primeiro termo de cada um na ultima posição dos respectivos *arrays*. Dessa forma, os *arrays* representam uma poligonal devidamente fechada. Em seguida convertem-se os dados para *'Dynamic Data*', para aplicá-los à função '*Build XY Graph'*, que entrega o gráfico montado ao respetivo indicador.

Outra operação é a obtenção do diretório do executável.

Figura 61 - Obtenção do Diretório do Executável

Nessa etapa, a partir do *path* do executável, dado pelo usuário, aplica-se a função '*Search and Replace String*' (com '*replace all*' em '*TRUE*') para encontrar o *offset* da ultima barra, isso é, onde se inicia o nome do arquivo em si. Com esse *offset*, a função '*Replace Substring*' obtém o *string* equivalente ao *path* do diretório onde o executável está salvo. Finalmente, o *string* é convertido em *path*.

Como se verá adiante, para efeito de controle, o executável, ao final de seu funcionamento, cria um arquivo de texto de nome FILEND.txt em seu diretório. Dessa maneira, para garantir o funcionamento adequado do programa, esse arquivo deve ser inicialmente deletado, caso exista.

Figura 62 - Deletar FILEND

No caso, recebendo o diretório do executável préviamente obtido, com a função '*Build Path*' monta-se o *path* do arquivo a ser buscado. Em seguida, aplica-se o SubVI 'Não Achou Arquivo True', cujo diagrama de blocos consta abaixo.

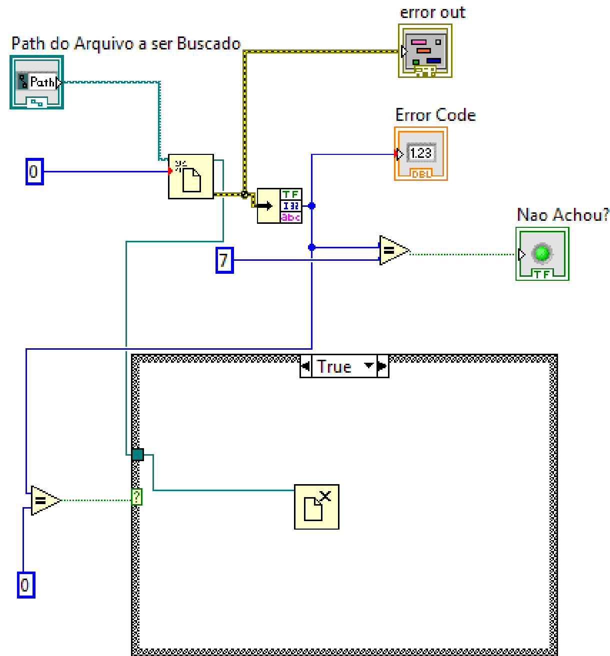


Figura 63 – SubVI Não Achou Arquivo True

O SubVI, recebendo o *path* de um arquivo aplica uma função para abrí-lo, recebendo seu erro de saída. O *cluster* desse erro é dividido em seus elementos, dos quais se extraí o código de erro, entregue ao usuário. Se esse valor é 7, quer dizer que o arquivo não existe, o que é comunicado por uma saída booleana. Se o valor for 0 então não houve erro e o arquivo foi aberto normalmente e então uma estrutura de casos fecha o arquivo. Assim, no VI principal, se o arquivo foi aberto normalmente, o mesmo é deletado em uma estrutura de casos.

Finalmente, implementou-se uma estrutura de casos para garantir que, quando ativado, o VI restaure seus controles e indicadores para valores padrão.

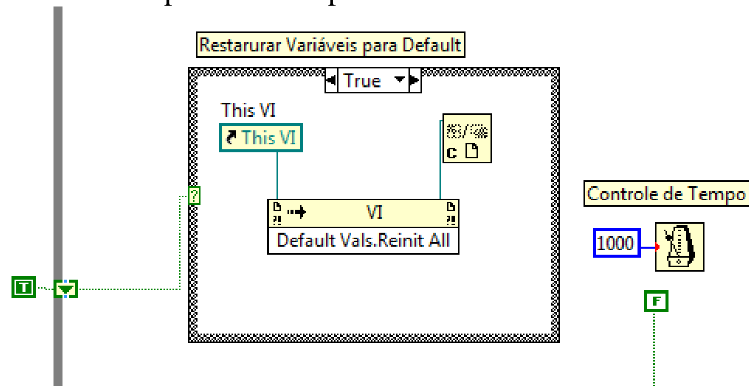


Figura 64 - Restauração das Variáveis

A estrutura de casos recebe controle de um *shift register* inicializado em '*TRUE*', mas que recebe '*FALSE*' em todas as demais iterações. Assim, garante-se que os comandos da estrutura, que restauram as variáveis só funcionarão uma vez, imediatamente após a ativação do programa.

Adicionalmente existe o controle do tempo de 1 segundo para cada iteração do *loop*.

As próximas etapas do código do programa operam dentro de uma estrutura de casos, controlada pelo botão 'ATIVAR' do painel de controle. No caso '*FALSE*', nada ocorre, e no caso '*TRUE*', os procedimentos, que utiliazam os dados de entrada e as inicializações, são efetuados. Como tal botão

foi programado para operar como ‘campainha’, quando o usuário o pressiona, os processos são realizados apenas uma vez, entregando as saídas do programa, visíveis nos inidicadores do painel frontal.

5.6.2. ESCRITA DOS DADOS DE ENTRADA

Uma vez que ‘ATIVAR’ é pressionado, iniciam-se as operações de escrita dos dados.

O executável utilizado recebe suas entradas em arquivos de texto. Dessa maneira, para aplicá-lo, é necessário converter as informações fornecidas pelo usuário para esse formato. Basicamente, dois arquivos devem ser criados. Um deles leva o nome GEOM.txt, contendo o nome da poligonal. O outro leva o nome da poligonal e contém as coordenadas dos vértices.

Para realizar as funções necessárias, foram criados 3 SubVIs auxiliares.

Um deles, chamado ‘Cria Arquivo Nome Tipo’, recebe o nome de um arquivo, sua extensão, o path do diretório e o cria ou substituí.

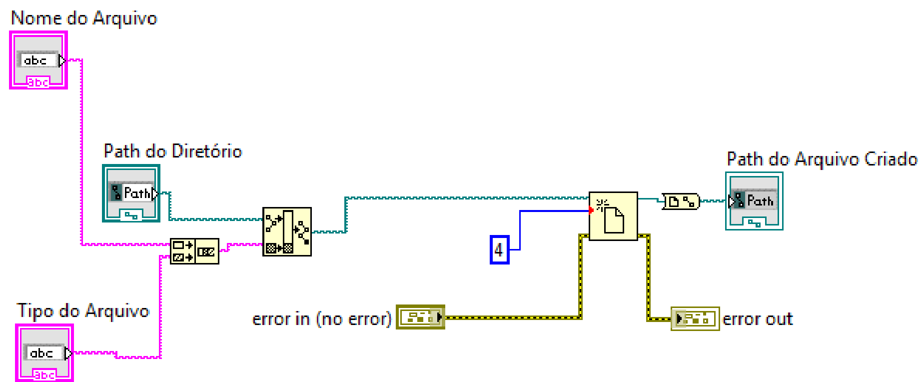


Figura 65 - SubVI Cria Arquivo Nome Tipo

Aplicam-se ainda os SubVIs ‘Escreve Linha’, ou sua versão análoga para arrays de duas dimensões ‘Escreve Tabela’. Para ambos os casos, recebido um *array* de *strings* e o *path* de um arquivo, monta-se uma *spreadsheet* do *array*, a qual é escrita no arquivo.

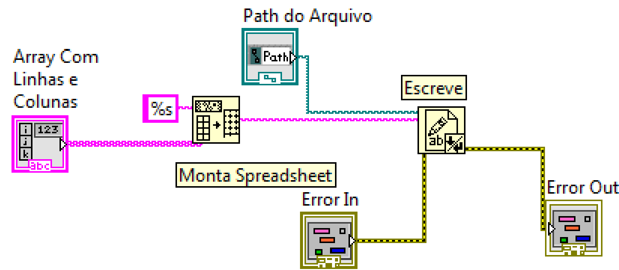


Figura 66 - SubVI Escreve Tabela

Assim, em primeiro lugar, o programa cria e escreve o arquivo GEOM.txt, com o nome da poligonal mais extensão ‘.txt’, no diretório do executável.

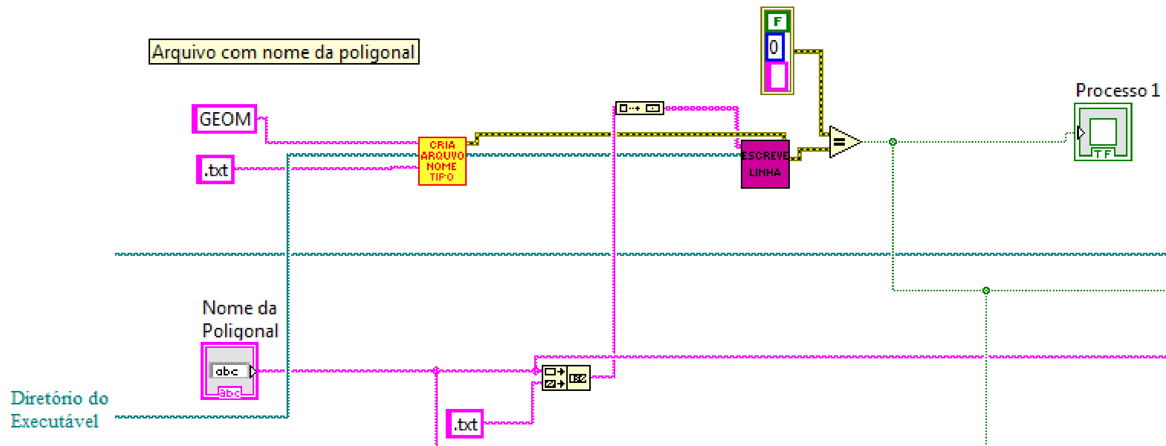


Figura 67 - Escrita do Arquivo com Nome da Poligonal

De fato, simplesmente concatenam-se strings, convertidos em *array* 1D para obter o texto a ser escrito e aplicam-se os SubVIs supracitados. Adicionalmente verifica-se se houve erro na criação ou escrita do arquivo, havendo aviso no indicador. Como se verá, o resultado dessa comparação será utilizado ainda para outros propósitos.

Em segundo lugar, deve-se criar e escrever o arquivo com os dados da poligonal. Ainda que a construção do texto a ser escrito seja mais complexa, o código eh absolutamente análogo, exceto pela conversão de variáveis numéricas em strings, a qual pode ser visualizada pela mudança nos tipos de cabos.

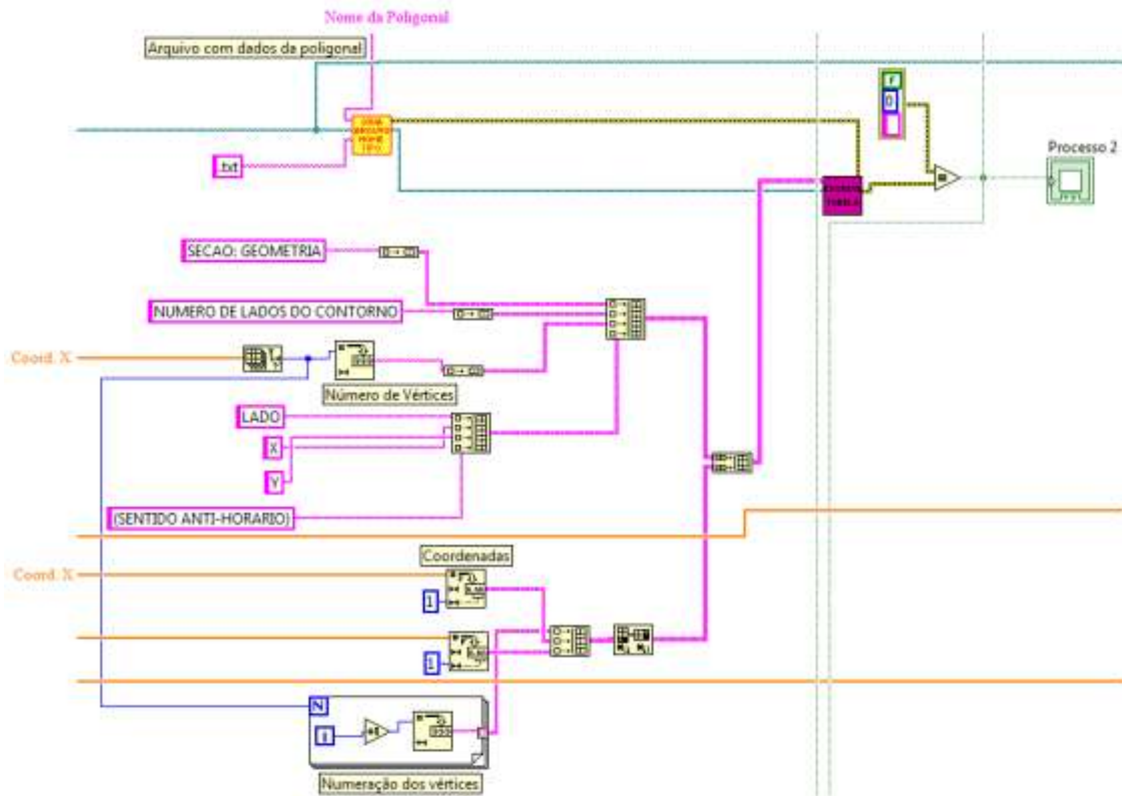
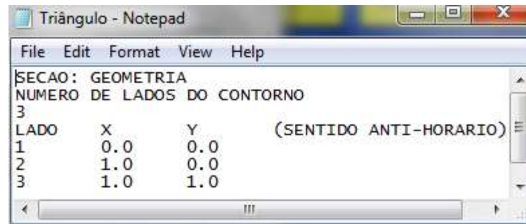


Figura 68 - Escrita do Arquivo com Dados da Poligonal

Para um triângulo, por exemplo, obter-se-ia a seguinte saída:



```
SECAO: GEOMETRIA
NUMERO DE LADOS DO CONTORNO
3
LADO    X       Y       (SENTIDO ANTI-HORARIO)
1       0.0     0.0
2       1.0     0.0
3       1.0     1.0
```

Figura 69 - Exemplo de ArquivoGerado

Após realizadas ambas as tarefas, o programa faz uma verificação booleana para conferir se os processos aconteceram sem erros. Em caso positivo, uma outra estrutura de casos é ativada, dando continuidade ao funcionamento do programa.

5.6.3. CHAMADA DO EXECUTÁVEL

Em posse dos dados em arquivo de texto, o executável pode ser utilizado para calcular as grandezas objetivadas. A chamada do executável se dá de maneira muito símples – basta aplicar a função '*System Exec*', entrando com o path do executável - optando-se por fazer o VI aguardar o fim da execução e com janela minimizada. Realiza-se, novamente, verificação da ocorrência de erros.

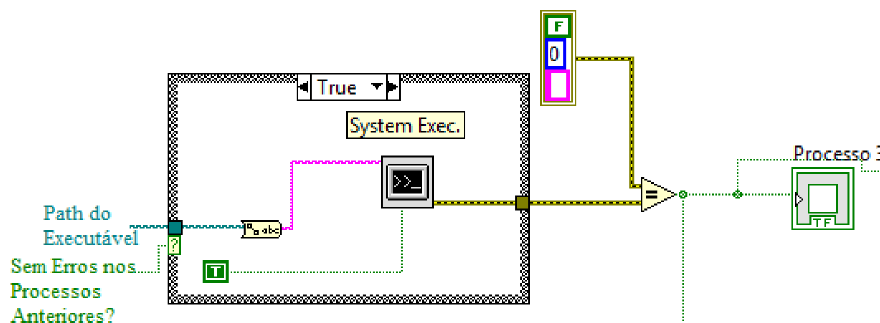


Figura 70 - Chamada do Executávle

Se não houverem erros, a estrutura de casos da próxima etapa executa, em seguida.

5.6.4. LEITURA DO ARQUIVO GERADO E APRESENTAÇÃO DE RESULTADOS

O executável é programado para gerar um arquivo de texto com as características geométricas da poligonal, no diretório do executável, com o nome da poligonal e extensão '.lst'. O padrão de formatação, por sua vez, é conhecido.

```
Características Geométricas da seção do arquivo Triângulo.txt
Hora:   12:  27:  23   Data: 20131107
   X_CG      Y_CG      AREA      JXCG      JYCG      JXYCG     JMAX      JMIN      TETA
   0.6667    0.3333    0.5000    0.0278    0.0278    0.0139    0.0417    0.0139    45.0000
FIM
```

Figura 71 - Arquivo a ser Lido

Como se percebe, os dados de fato encontram-se na quarta linha, em determinada ordem. Assim, utiliza-se, para ler e converter os dados, uma estrutura semelhante à apresentada no ítem 4.3.3..

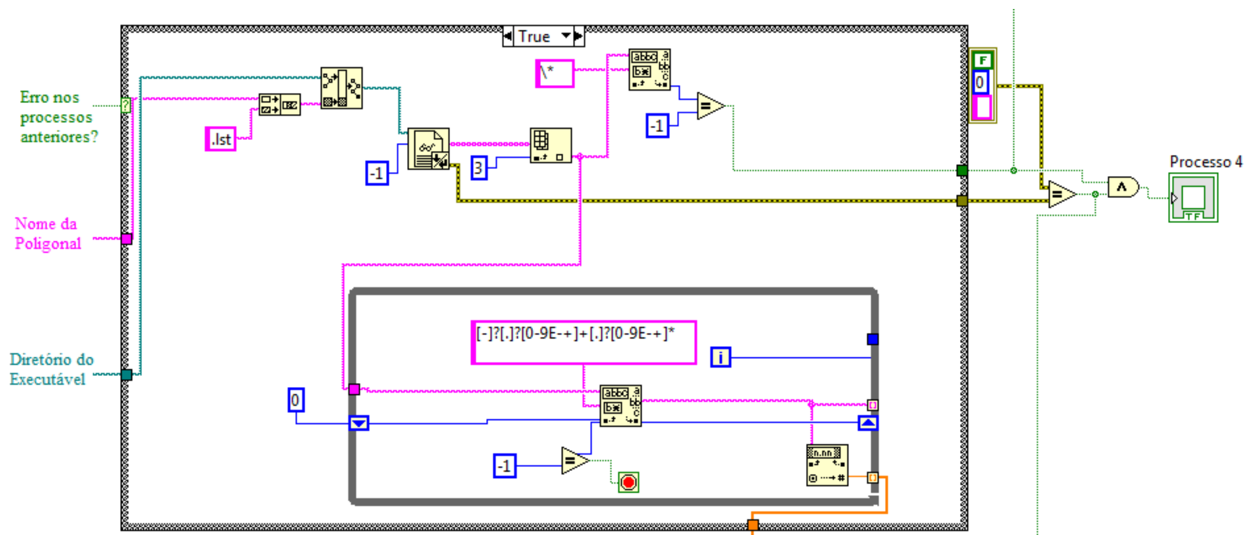


Figura 72 - Tradução dos Arquivos de Texto

Assim, contrói-se o *path* do arquivo, o qual é lido, em seguida. Toma-se a quarta linha (ínice 3), na qual são procurados e armazenados num *array*, após convertidos, todos os valores numéricos. Para tal, busca-se evetual sinal negativo, uma sequância de números, seguida de possível ponto e outra sequência de números.

Em seguida, dois tipos de erros são verificados. O primeiro deles é na leitura. O segundo é relativo a erros de cálculo no próprio executável, geralmente devidos a entradas inadequadas. Quando isso ocorre, o executável libera no arquivo de texto de saída, para os valores que resultaram erros, os caracteres '*'. Assim, simplesmente buscam-se os referiods caracteres no arquivo de saída.

Finalmente, o programa apresenta os dados obtidos. Nesse sentido, deseja-se escrever todos os valores numéricos obtidos, em indicadores, bem como plotar a poligonal, e um de seus eixos principais, sendo o baricentro a origem.

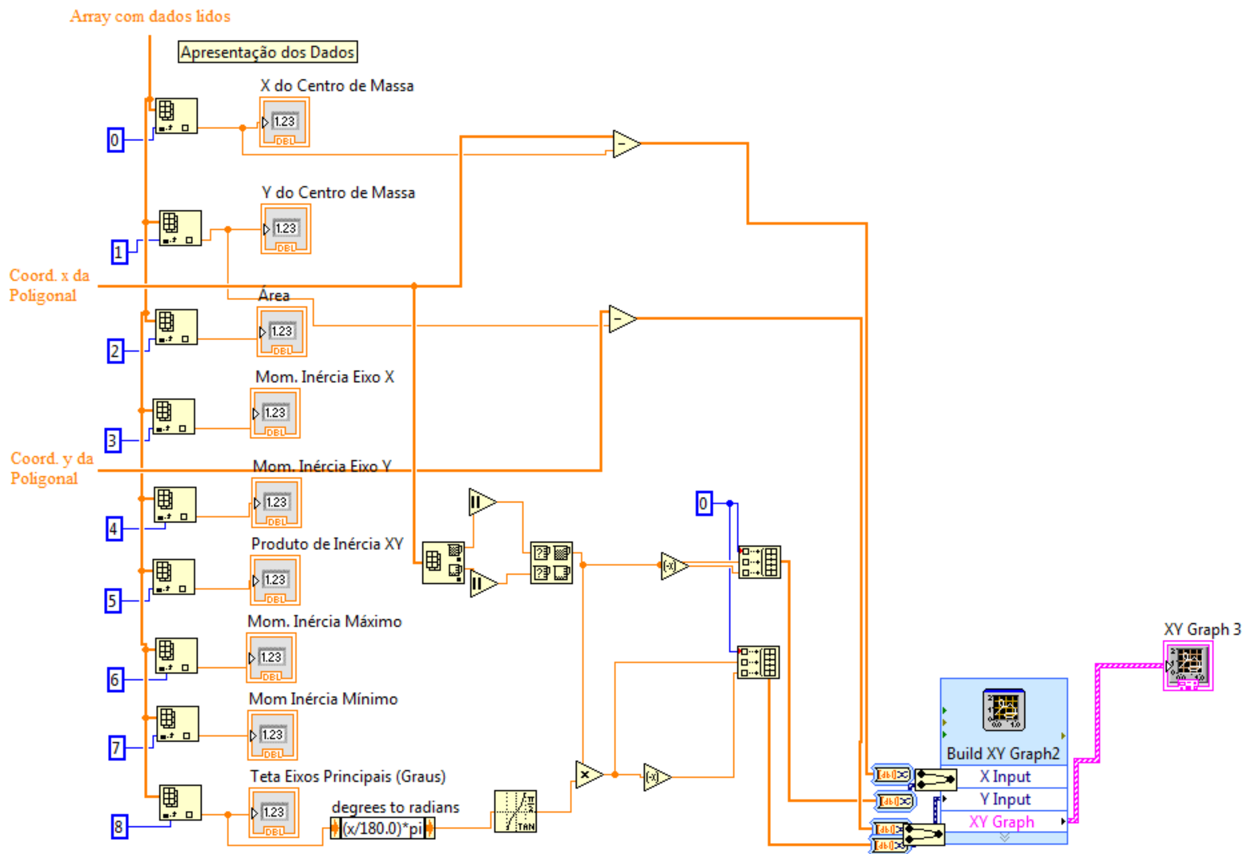


Figura 73 - Apresentação dos Dados

Como se observa, primeiramente, os dados do array lido são devidademnte alocados, por seus índices, nos indicares adequados. Em segundo lugar, as coordenadas da poligonal são deslocadas para o novo referencial, simplesmente subtraíndo o valor das coordenadas do baricentro. Finalmente, para traçar o eixo principal de inércia, tomam-se as coordenadas 'x' do array, já convertidas, da qual se pega o maior valor absoluto. Isso é feito para que a reta plotada ocupe toda a área vizualizada do gráfico. Assim, para as coordenadas x, cria-se um *array* com o valor negativo do obtido, zero, e o valor obtido propriamente. Para as coordenadas y utlizam-se os respectivos valores, mas multiplicados pela tangente de *theta*, obtido do executável. Assim, as coordenadas da poligonal e da reta são convertidas em '*Dynamica Data*'. Por fim, os sinais são unidos e plotados num gráfico XY.

O programa possuí ainda a capacidade de alertar os usuários sobre a natureza de possíveis erros. Para cada verificação de efetuação adequada dos processos, o programa possuí uma pequena estrtura de casos para lidar com os erros.

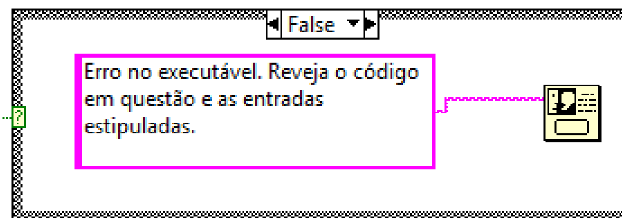


Figura 74 - Tratamento dos Erros

Por exemplo, se a chamada do executável retornar erro, a estrutura de casos é acionada em 'false' e uma caixa de diálogo com um texto explicativo é mostrada.

6. BIBLIOGRAFIA

Agilent Technologies, **application Note 290-1 – Pratical Strain Gage Measurements**, 1999.

Bischoff, Reinhard; Meyer, Jonas; Feltrin , Glauco. Wireless Sensor Network Platforms. **Encyclopedia of Structural Health Monitoring.** Edited by Christian Boller, Fu-Kuo Chang and Yozo Fujino, John Wiley & Sons, pp. 1-10, 2009.

Camargo, Valter Luís Arlindo de, **Desenvolvimento e Implementação de uma Plataforma para Monitoramento Estrutural Utilizando Sensores Extensométricos Conectados em Rede**, Tese (Mestrado em Engenharia Elétrica), Universidade Estadual de Londrina, Londrina, 2008.

Camelo, H. N.; de Maria, P. H. S.; Carvalho, P. C. M.; Pereira, T. B. **Métodos de Extrapolação de Velocidade do Vento para Regiões Litorâneas do Nordeste Brasileiro,** 2010.

Drummond, J. (1998). **PHY 406 - Microprocessor Interfacing Techniques.** http://www.upscale.utoronto.ca/GeneralInterest/Drummond/LabVIEW/

Farrar, C. R.; Park, G.; Allen, D. W.; Todd , M. 2006. **Sensor Network Paradigms for Structural Health Monitoring,** Struc. Ctrl. Health Mon., 13:210-255.

Hill, Jason Lester, **System Architecture for Wireless Sensor Networks**, Tese (Doutorado em Ciência da Computação), Universidade da Califórnia, Berkeley, 2003.

Housner, G. et al., **Structural Control: Past, Present and Future**, ASCE., J. Engrg. Mech.,, vol. 123, pp. 897-971, 1997.

Kimura, A., **Informática Aplicada em Estruturas de Concreto Armado**, PINI: 2007.

LabVIEW: Data Acquisition Basics Manual (1998). National Instruments Corporate Headquarters. ftp://ftp.ni.com/support/manuals/320997e.pdf

LabVIEW User Manual (2003). National Instruments Corporate Headquarters. http://www.ni.com/pdf/manuals/320999e.pdf

Linux Information Project. **Path Definition.** Disponível em: http://www.linfo.org/path.html Acesso em: 19 de nov. 2013

Lui, S.C.; Tomizuka, M., **"Strategic Research for Sensors and Smart Structures Technology" First International Conference on Structural Health Monitoring and Intelligent Infrastructure (SHMII-1'2003),** Tokyo, Japan, November 13-15, 2003.

Nagayama, T,; Ruiz-Sandoval, M.;Spencer Jr., B. F.; Mechitov, K. A,; Agha, G., **Wireless Strain Sensor Development for Civil Infrastructure**, 2004.

National Instruments. **Manuais. Enabling Auto-Indexing For Loops.** Disponível em: http://zone.ni.com/reference/en-XX/help/371361G-01/lvhowto/auto_indexing_1/Acesso em: 19 de nov. 2013.

National Instruments. **Manuais. Handling Errors.** Disponível em: http://www.ni.com/gettingstarted/labviewbasics/handlingerrors.htm Acesso em: 19 de nov. 2013.

National Instruments. **Manuais. Index Array Function.** Disponível em: http://zone.ni.com/reference/en-XX/help/371361J-01/glang/index_array/. Acesso em: 19 de nov. 2013.

National Instruments. **Manuais. Multitasking, Multithreading and Multiprocessing.** Disponível em: http://zone.ni.com/reference/en-XX/help/371361J-01/lvconcepts/multitask_multithread_multip/. Acesso em: 19 de nov. 2013.

National Instruments. **Manuais. Refnum Controls.** Disponível em: http://zone.ni.com/reference/en-XX/help/371361H-01/lvhowto/refnum_controls_and_indica/ Acesso em: 19 de nov. 2013.

National Instruments. **Manuais. Wait Until Next ms Multiple Function.** Disponível em: http://zone.ni.com/reference/en-XX/help/371361H-01/glang/wait_till_next_ms_multiple/ Acesso em: 19 de nov. 2013.

National Instruments. **Manuais. Write to Text File Function**. Disponível em: http://zone.ni.com/reference/en-XX/help/371361H-01/glang/write_characters_to_file/ Acesso em: 19 de nov. 2013.

National Instruments, **NI LabVIEW for CompactRIO Developer's Guide**, National Instruments: 2013

National Instruments. **Produtos e Serviços. CompactDAQ.** Disponível em: http://www.ni.com/compactdaq/whatis/pt/ Acesso em: 19 de nov. 2013.

National Instruments. **Suporte. How Can I Send Email From LabVIEW Via a Secure SMTP Server Such as GMAIL.** Disponível em: http://digital.ni.com/public.nsf/allkb/484272384C2960AA8625749E006512EE Acesso em: 19 de nov. 2013.

Nery, Roberta T. C., **Introdução ao Labview**, Universidade Federal do Pará. (2013)

RFC5321 – Simple Mail Transfer Protocol. Disponível em: http://www.ni.com/compactdaq/whatis/pt/ Acesso em: 19 de nov. 2013.

Spencer, B. F. Jr; Ruiz-Sandoval, M.; Krata, N. **Smart sensing technology: opportunities and challenges.** Journal of Structural Control and Health Monitoring 2004; 11:349–368