Universidade Federal do Rio Grande
Instituto de Matemática, Estatística e Física
Bacharelado em Física

Lucas Heck dos Santos

AUTOMAÇÃO DE UM SISTEMA DE MEDIDAS DE PROPRIEDADES ELÉTRICAS

Rio Grande
2015
LUCAS HECK DOS SANTOS

AUTOMAÇÃO DE UM SISTEMA DE MEDIDAS DE PROPRIEDADES ELÉTRICAS

Trabalho de Graduação (Física) II submetido ao Instituto de Matemática, Estatística e Física da Universidade Federal do Rio Grande – FURG.
Orientador: Prof. Dr. Jorge Luiz Pimentel Jr

Rio Grande
2015

LUCAS HECK DOS SANTOS

AUTOMAÇÃO DE UM SISTEMA
DE MEDIDAS DE PROPRIEDADES ELÉTRICAS

Trabalho de Graduação (Física) II submetido ao Instituto de Matemática, Estatística e Física da Universidade Federal do Rio Grande – FURG.

BANCA EXAMINADORA

_____

Prof. Dr. Jorge Luiz Pimentel Jr – Universidade Federal do Rio Grande – FURG (Orientador)

_____

Prof.ª Dr.ª Águeda Maria Turatti - Universidade Federal do Rio Grande – FURG

_____

Prof. Dr. Fabricio Ferrari – Universidade Federal do Rio Grande - FURG

LISTA DE FIGURAS

SUMÁRIO

RESUMO

O objetivo deste trabalho é automatizar uma montagem experimental para a obtenção de propriedades elétricas de amostras sólidas em função da temperatura. Este sistema será a primeira plataforma de medidas em ambiente criogênico do Laboratório de Supercondutividade em Magnetismo da FURG. Para garantir a eficiência e segurança na coleta de dados é necessário a utilização de um software, o qual também permita o controle da temperatura. A medida da resistividade elétrica permite estudar transições de fase e classificar materiais. O software de controle foi desenvolvido, utilizando-se a linguagem Python, para fazer a leitura dos equipamentos de medida, através de uma interface GPIB, e realizar a coleta e dados de forma automática. Para a determinação da temperatura, utilizou-se um termoresistor calibrado de platina. Para validar o software, utilizou-se como amostra a ser medida um termoresistor de platina não calibrado. Os resultados obtidos permitiram observar que o software permite monitorar a variação térmica da resistividade elétrica, medir a temperatura com precisão e fornecer dados experimentais com baixa latência. Desta forma, podemos iniciar a realização de medidas de transporte elétrico em outros tipos de materiais.

Palavras–chave: resistividade elétrica, automatização, interface gpib.

ABSTRACT

In this work, we automatize an experimental setup in order to obtain electrical proprieties of solid samples as a function of temperature. This will be the first platform for measurements in a cryogenic setting of the Laboratory of Superconductivity and Magnetism of FURG. To ensure the efficiency and safety in the capture of data it is necessary the use of a software, which also controls the temperature. Measurement of electrical resistivity allows the study of phase transitions and classification of materials. The software developed, using the language Python, to read the measurement from the devices, through a GPIB interface, and automatize the capture of data. To determine the temperature, a calibrated platinum thermistor was used. To verify the software, a non-calibrated platinum thermistor was used. The obtained results allowed us to see that the software could monitor the thermal variance of the electrical resistivity, calculate the temperature with precision and supply experimental data with small delay. At the laboratory, we are now able to start the measurement of electrical transport in other kinds of materials.

Keywords: electrical resistivity, automation, gpib interface.

INTRODUÇÃO

A automação de sistemas abrange uma grande área e permite que estudos físicos sejam realizados com maior precisão. Estes experimentos frequentemente são comparados e testados por modelos e teorias que visam explicar os fenômenos observados. A teoria de controle auxiliando o conhecimento físico permite que os experimentos sejam automatizados para maior eficiência. Dentre as diversas propriedades que caracterizam o material, a resistividade ρ se torna foco do trabalho, sendo intrínseca de todo material. Ela varia com a temperatura e sua caracterização permite o desenvolvimento de dispositivos elétricos. A resistividade elétrica é uma propriedade que pode ser obtida a partir da medida da resistência elétrica de uma amostra do material que pretendemos estudar.

Com o intuito de estudar esta propriedade elétrica, utilizamos uma placa de aquisição de dados GPIB [1], e desenvolvemos um software utilizando a linguagem Python [2] para comunicar os equipamentos com o computador para realização das medidas. Através de um termômetro (um termoresistor de platina calibrado) instalado na haste de forma a estar termicamente acoplado à amostra dentro do criostato, obtemos a temperatura com alta precisão, utilizando a técnica das quatro pontas [3], além da resistividade da própria amostra.

Neste trabalho apresentamos o desenvolvimento do software para a automatização das medidas e a coleta de dados. Além disso, mostramos um exemplo de utilização do software através da calibração de um termoresistor. Este trabalho está estruturado da seguinte forma:

No capítulo 1 é introduzido o conceito de resistividade elétrico e como é realizada a medida no laboratório.

No capítulo 2 apresenta-se os conceitos necessários ao desenvolvimento do software e ao entendimento da teoria de controle.

No capítulo 3 é apresentado o desenvolvimento do software e de seus módulos.

No capítulo 4 são apresentados os resultados obtidos, com o intuito verificar o funcionamento do software.

No capítulo 5 são apresentadas as conclusões e perspectivas futuras.

CAPÍTULO 1 – RESISTIVIDADE

1.1 TEORIA

As diversas propriedades elétricas que compõe os materiais são de vital importância quando se deseja realizar o estudo e/ou produzir novos materiais. Dentre essas propriedades, temos a *resistividade* ρ, que é intrínseca da matéria. Esta deve ser diferenciada da *resistência elétrica R*, que por sua vez consiste em uma propriedade de um dispositivo. Ambas representam a resistência da passagem de portadores de carga pelo material, entretanto a resistência se difere por ser dependente das propriedades geométricas.

Para metais, experimentalmente nós obtemos a relação da densidade de corrente elétrica $\vec{J}$ com o campo elétrico $\vec{E}$ dado pela equação:

$$\vec{E} = \rho \vec{J} \quad (1.1)$$

Integrando os dois lados da equação acima para o caso de uma amostra na forma de um paralelepípedo e considerando que estamos tratando de um campo elétrico constante, obtemos:

$$\rho = \frac{V}{I}\frac{A}{l} \quad (1.2)$$

onde *V* é a diferença de potencial, *I* a corrente, *A* a seção de choque e *l* o comprimento da amostra.

Assim conseguimos obter a resistividade através de propriedades mensuráveis, mas ainda podemos reescreve-la através de uma expressão matemática dada por:

$$R = \frac{V}{I} \quad (1.3)$$

A equação 1.3 corresponde a resistência, entretanto deve-se estar ciente de que esta relação descreve muitos materiais isotrópicos, mas não todos materiais. Assim temos:

$$\rho = R\frac{A}{l} \quad (1.4)$$

ou

$$R = \rho\frac{l}{A} \quad (1.5)$$

Esta última equação é comprovada experimentalmente quando utiliza-se um mesmo material para se obter dois dispositivos que variem em comprimento e seção transversal, ao se calcular a resistência se observa que os valores vão variar proporcionalmente ás propriedades geometrias que os diferem.

A resistividade, apesar de ser obtida através da equação (1.4), não é explicita sua

importância relacionada a variação de temperatura, esta importância surge com o interesse de classificar o tipo de material tratado e se ocorre alguma transição de fase. Podemos ainda escrever a resistividade através de uma variação da *Lei de Matthissen [4]*, postulada em 1860:

$$\rho_T = \rho_0 + \rho_{iT} \qquad (1.6)$$

Da equação 1.6, temos que o termo ρ_T é a resistividade total do material, ρ_0 é a resistividade residual devido a presença de falhas, impurezas e tensões na rede, e ρ_{iT} é a parte dependente da temperatura. Sua ideia era de que a inserção de impurezas não afetaria a resistividade dependente da temperatura. Na figura 1 temos o comportamento de um condutor, em temperaturas baixas apenas a resistividade residual é vista, como esperado por Matthissen, e a dependente da temperatura com a resposta linear esperada.

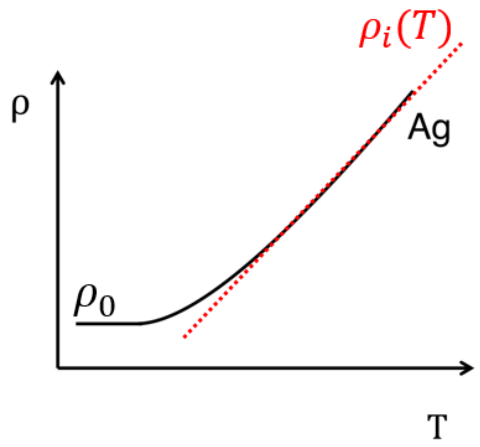


Figura 1: Resposta da resistividade de um condutor variando com a temperatura.

Em 1900, Drude propôs o chamado *Modelo de Drude [5]*, logo após a descoberta do elétron por J. J. Thomson [6], para explicar o comportamento do elétron em um sólido. Apesar deste ser um modelo clássico e conter diversas limitações, realiza diversas considerações válidas que permitem a observação da dependência da resistividade com a temperatura. Neste modelo, supunha-se que os elétrons se comportavam como um gás de partículas clássicas ao se moverem dentro de um sólido e sofriam várias colisões com os íons. Atualmente sabemos que as vibrações da rede e seus defeitos são mais importantes que essas colisões, mas ele ainda considerou que a cada colisão havia um *tempo de colisão médio* τ e que viajava em linha reta após as colisões, o que é correto. Neste modelo, quando ele considerava o caso de um campo elétrico constante, referente a *Lei de Ohm*, os cálculos através do momento linear da partícula antes e pós colisão permitiam que ele obtivesse a *velocidade de arraste* v da partícula, assim

obtinha a seguinte relação para a resistividade:

$$\rho = \frac{m_e}{e^2 n \tau} \qquad (1.7)$$

em que $\tau(T)$ é o tempo de colisão médio, m_eé a massa do elétron, $n(T)$ é a densidade de portadores de carga e e é a carga do elétron.

A diferença entre a resistividade do Ouro e do Silício é da ordem de 10^{20} Ωm, portanto havia de ter uma explicação de tal diferença com base na equação 1.7. O motivo é um ser metal condutor e o outro semicondutor, respectivamente, assim mesmo que a equação seja a mesma a importância dos termos diferem.

No caso de um condutor, a figura 2 (a) é utilizada para explicar. Temos que os elétrons se deslocando sobre o material sofrem constantes colisões com fônons e outros elétrons. A variação da temperatura causa mudanças na vibração dos íons da rede, portanto o fator importante da resistividade neste caso é o tempo de colisão médio, definido como τ na equação 1.7.

Para o caso de um semicondutor recorremos a teoria de bandas, representado pela figura 2 (b). Neste caso há um gap entre a banda de valência e a banda de condução. Isto significa que há uma dificuldade para os elétrons saírem da banda de valência, e em temperaturas baixas observaríamos uma ainda maior dificuldade pois o gap se tornaria maior. O número de elétrons que se encontram na banda de condução são definidos como a densidade de portador de carga do material, o *n* da equação 1.7. Para o metal há um *overlap* entre essas bandas, assim esse valor não varia suficientemente, comparado com o tempo de colisão médio, para causar uma mudança expressiva na resistividade.

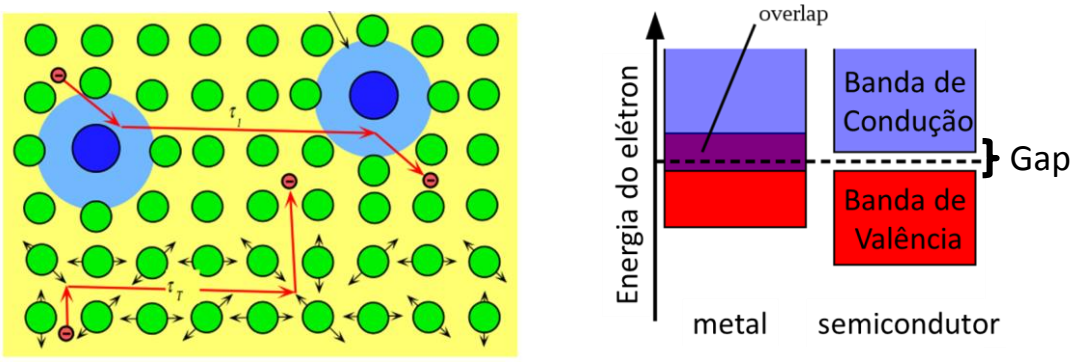


Figura 2: (a) descrição microscópica das colisões de um elétron em um material e (b) representação da teoria de bandas para um metal e um semicondutor. Adaptado de [21].

Entretanto, ambas proposições foram realizadas antes do surgimento de técnicas que permitissem a liquefação do Hélio[1]. Apenas 1908 Onnes [7] foi capaz de realiza-la, permitindo que fosse possível realizar medidas com temperatura da ordem de 4 K. Alguns anos à frente, decidiu estudar o comportamento da resistividade do mercúrio em tais temperaturas.

1.2 DETALHES EXPERIMENTAIS

Onnes iniciou o estudo e observou que a medida era de um comportamento que contrapunha a lei de Matthissen, quando a temperatura do mercúrio atingiu *4,2 K* a resistividade passava por uma queda abrupta para um valor nulo, não exibindo a resistividade residual. Na figura 3 podemos observar a existência de uma temperatura crítica T_c representando a fronteira entre o estado normal, em que $T > T_c$, e o estado denominado de supercondutor, em que $T < T_c$.

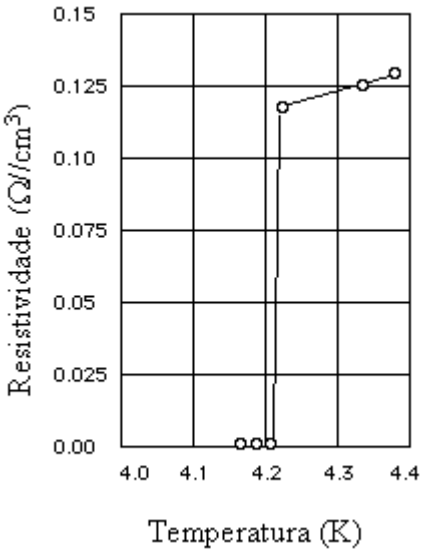


Figura 3: Gráfico da medida obtida por Onnes, representando a resistividade em função da temperatura e a transição para a fase supercondutora. Adaptado de [8]

Além de necessitar de temperaturas baixas, é necessário possuir um criostato que permite tais medidas. Este por sua vez, pode possuir diversas montagens, mas todas elas consistem sempre em manter um ambiente em baixa temperatura, no qual se encontrará a

[1] A liquefação do hidrogênio, de temperatura 20.28 K, já era obtida desde 1885 mas ainda não era suficiente para exibir o estado supercondutor nos materiais conhecidos.

amostra, para realizar a medida, aplicar um campo magnético, ou para outra necessidade.

Na figura 4 podemos observar um exemplo de criostato. Neste caso, trata-se de um que suporta Nitrogênio e Hélio líquidos. Podemos ver que a haste da amostra tem duas extremidades importantes, a superior possui a saída dos cabos para os equipamentos, assim os fios passam atravessando o criostato até a extremidade inferior. Chegando no porta amostra os fios são soldados na amostra, no termômetro e no aquecedor, que, como é visto, estão isolados por vácuo para não manter o contato direto com o hélio líquido. Assim, é possível então realizar medidas na amostra em temperaturas baixas sem danificá-la devido ao toque direto com o hélio ou nitrogênio líquido.

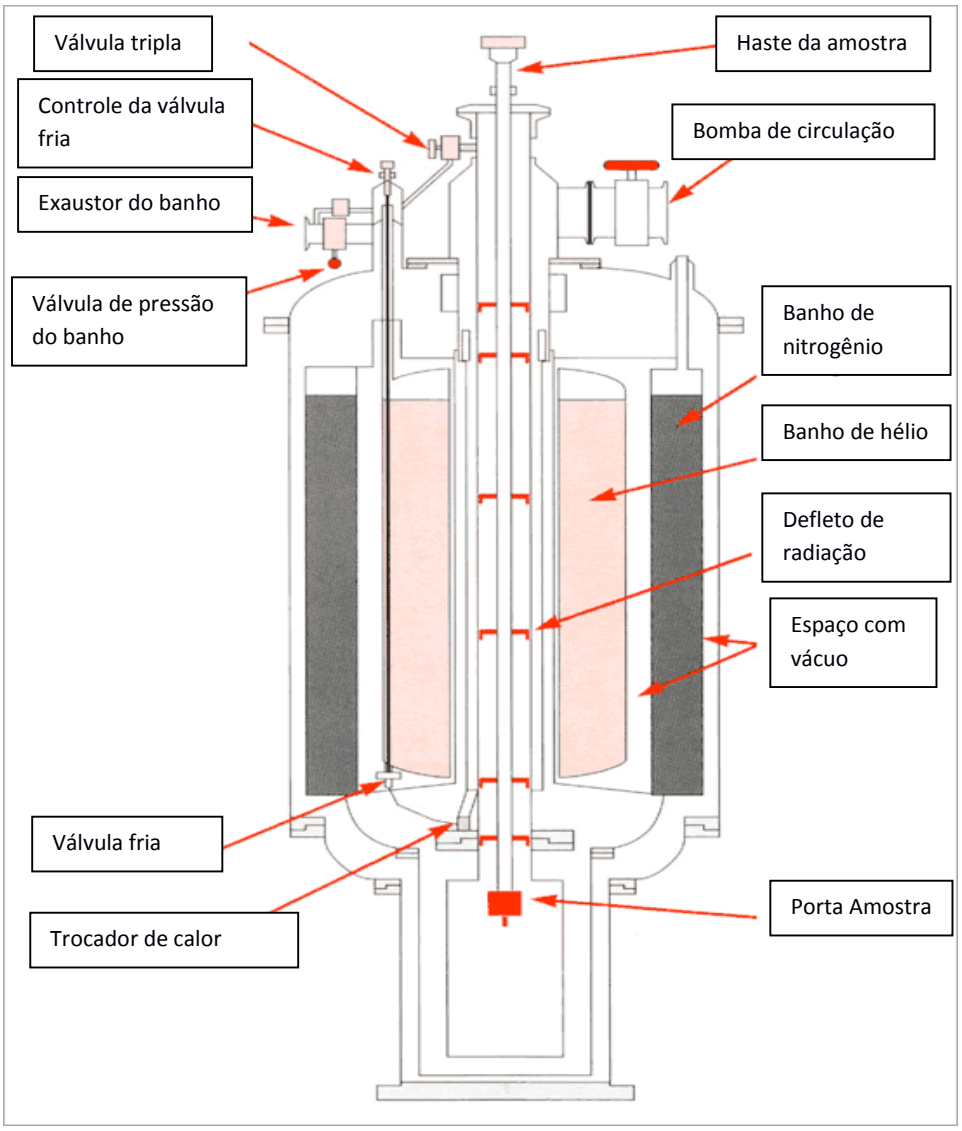


Figura 4: Representação de um criostato da marca X. adaptado de [9]

Com o criostato e a cana confeccionados, realiza-se a ligação com multímetros, fontes

de corrente e outros equipamentos eletrônicos conforme a necessidade da medida. Para medirmos a resistência elétrica, por exemplo, utilizamos a técnica das quatro pontas, que consiste em aplicar uma corrente elétrica e medir a tensão utilizando um multímetro e utilizamos a equação 1.3 para calcular a resistência, como pode ser visto na figura 5. Dependendo do caso, podemos operar com correntes e tensões alternadas ou contínuas.

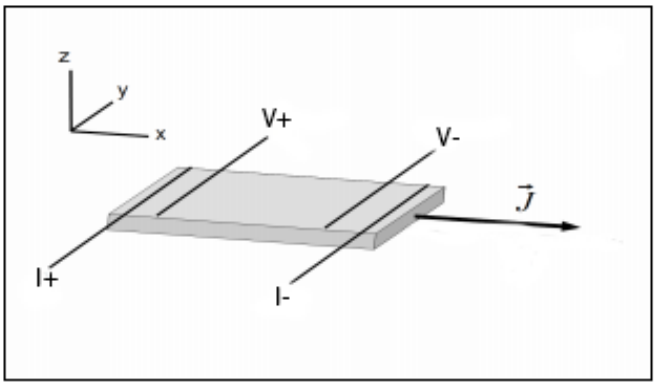


Figura 5: Aplicação da técnica das quatro pontas em uma amostra. Adaptado de [10].

A temperatura, assim como a resistividade, não é uma propriedade mensurável diretamente, por isso precisamos utilizar um termoresistor. Este é exatamente um resistor, com o fato de que sua variação com relação a temperatura já é a priori devidamente calibrada pelo fabricante.

Com todas essas ferramentas disponíveis é possível realizar as medidas, entretanto para obter uma precisão satisfatória é necessário que sejam feitas lentamente. Como esse processo pode chegar a horas, já que alguns sistemas conseguem ter uma taxa de controle de variação de $0{,}01\ K/min$ até $10\ K/min$, não podemos depender de um usuário escrevendo os valores em papel a cada medida, pois será um trabalho exaustivo e certamente ocorrerão erros humanos devido a isto. Para resolver isto é necessário que seja realizado o desenvolvimento de um software que automatize este processo, assim os dados podem até ser observados em tempo real e sem erros de cansaço devido a repetição.

CAPÍTULO 2 – CONTROLE, AUTOMAÇÃO E AQUISIÇÃO DE DADOS

2.1 INTRODUÇÃO

Neste capítulo serão descritos os conceitos fundamentais para o desenvolvimento de um software para controle, automação e aquisição de dados de um sistema de propriedades elétricas.

2.2 SISTEMAS DE CONTROLE

Sistema é um conjunto de equipamentos com o intuito de obter um certo valor, naturalmente aplicamos essa ideia a diversos ramos, seja abstrato ou não. Entretanto, quando tratamos de um sistema de controle, nos focamos no tipo de informação que queremos controlar e obter.

Para simplificar o estudo de tais sistemas, sem que seja necessária uma análise da montagem experimental, utilizam-se modelos como: diagrama de blocos, diagramas de fluxo de dados, equações, maquetes e diversos outros. Para visualização da montagem do aparato experimental utiliza-se o diagrama de blocos, assim somos capazes de entender melhor sobre o que se trata um sistema de controle. Na figura 6 temos o caso mais simples para o qual podemos descrever um sistema, dele definimos:

- **Entrada**: variável controlada, de referência;
- **Processo**: operação através de uma série de ações para se atingir um resultado final;
- **Saída**: valor obtido através do processo.

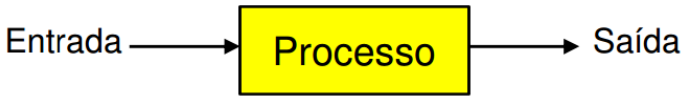


Figura 6: Modelo de um sistema simples usando diagrama de blocos. Adaptado de [11].

Através deste tipo de análise, a teoria de controle classifica sistemas de controle em dois tipos: de malha aberta e de malha fechada. Os sistemas de malha aberta, figura 7, tendem a ser mais simples, possuindo aplicações diretas já que sua intenção não é tratar o resultado obtido,

mas realizar uma operação assim que obter.

Devido a não possuir realimentação, possuem maior imprecisão mas custo baixo, assim é utilizado em situações rudimentares. Um exemplo são as máquinas de lavar roupa, ao iniciar o processo é entrada a intensidade da lavagem, mas não há controle para verificar se há necessidade de mais ciclos para obter um resultado mais próximo do ideal, apenas mantem o ciclo definido. E ainda tiramos algumas importantes nomenclaturas:

- **Controlador**: dispositivo que trata o valor real do sistema e define a sequência de operações do resto do sistema;
- **Atuador**: dispositivo que gera a ação do controlador ao processo;

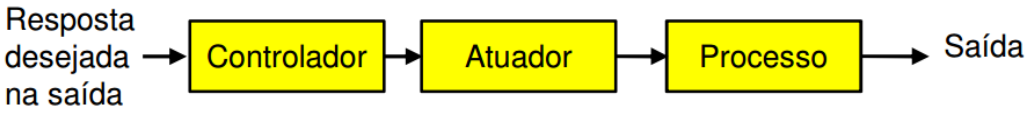


Figura 7: Modelo de um sistema de malha aberta usando diagrama de blocos. Adaptado de [11].

Para o caso de um sistema de controle em malha fechada, figura 8, há utilização nos casos onde é necessário controle dos erros. Sendo essa uma das diferenças básicas com relação ao anterior, de tal modo que agora podemos observar que possuímos um erro no diagrama, este é devido a uma relação entre o sinal de entrada e o de saída a realimentação, Este ciclo gera-o, para que no próximo, o valor obtido no processo não esteja mais sobre o efeito do erro, mas, caso ainda possua-o, o sistema continua do mesmo modo com o ciclo sempre reduzindo-o. Este tipo de sistema, em função dessas características, também é denominado sistema de controle realimentado e abaixo temos algumas outras nomenclaturas novas de grande importância para o entendimento:

- **Erro**: diferença entre o sinal de entrada e o de saída ou o de realimentação;
- **Realimentação**: passagem do sinal do processo para um sensor que vai reiniciar o ciclo;
- **Sensor**: objeto responsável por obter o elemento físico;
- **Medida de saída**: valor obtido pelo sensor em forma de sinal;

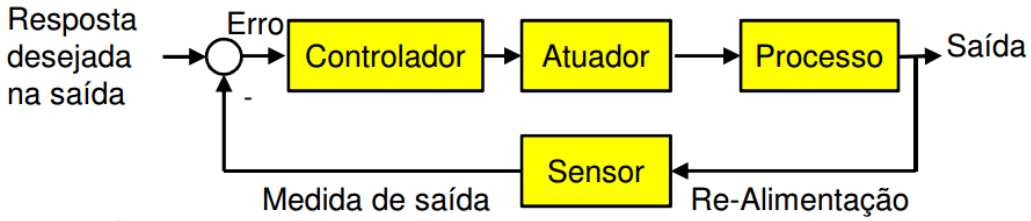


Figura 8: Modelo de um sistema de malha fechada usando diagrama de blocos. Adaptado de [11].

Através destes sistemas é possível construir montagens experimentais com diversas finalidades e com melhor taxa de precisão. Entretanto, se a princípio considerarmos que todos esses sistemas são manuais, mesmo que esteja utilizando um sistema de malha fechada, é possível que a precisão nunca seja suficiente para um determinado experimento. Isto porque dentre as possibilidades de erro, quando o ser humano realiza uma tarefa ele está exposto a diversas falhas naturais, como devido ao cansaço, em que pode anotar um valor errado ou se esquecer de ajustar alguma variável, dentre diversas outras possibilidades.

Na figura 9 (a) podemos ver o que acontece em um sistema de controle manual onde se há o interesse de manter uma taxa constante em um determinado valor(set-point). Neste caso, podemos ver pela curva que há uma resposta rápida do sistema, assim o usuário não consegue realizar um ajuste no experimento suficientemente rápido e sua taxa permanece com um grande erro em relação ao set-point.

Devido a esse tipo de problema, sistemas controlados passaram a ser automatizados com uso de computadores, assim o usuário poderia definir o set-point, como na figura 9 (b). Devido a sua realimentação, velocidade de processamento e recebimento de dados, há uma resposta mais ágil do estado em que se encontra o experimento, reduzindo a taxa com maior precisão até atingir o set-point. Alguns sistemas não utilizam computadores para automatização, o que continua sendo definitivamente viável, entretanto, a tecnologia permite que com computadores seja possível controlar o experimento de forma mais simples, alterando parâmetros no *software* e não no *hardware*.

Com a automatização, não apenas o sistema passa a resolver o problema sem causar cansaço ao usuário, em função de experimentos onde é realizada medida durante horas, mas também obtêm-se uma maior taxa de precisão dos valores desejados.

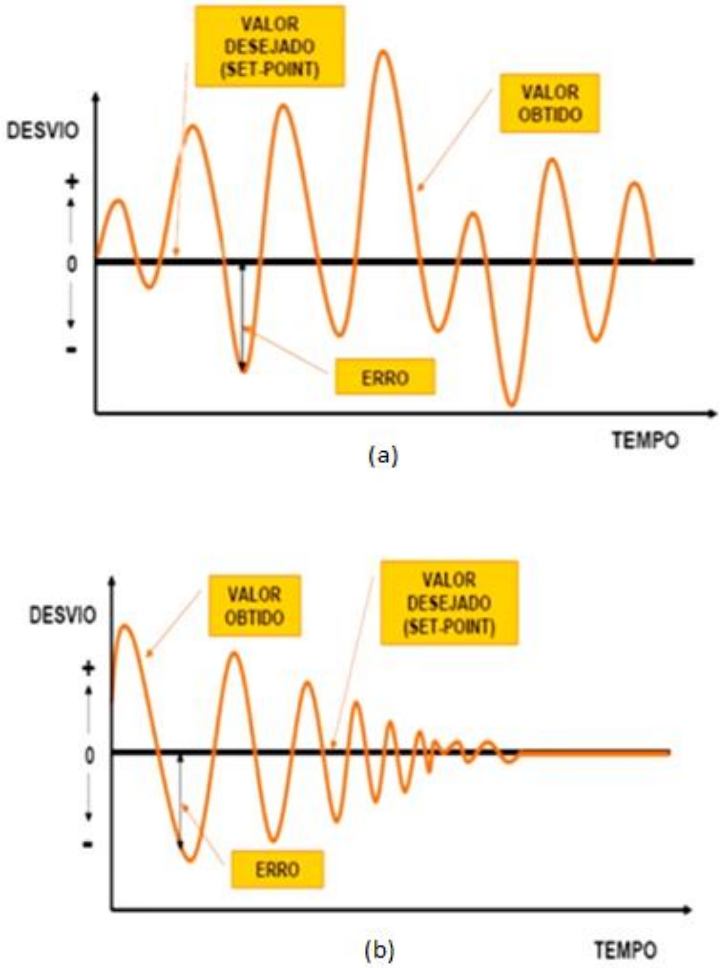


Figura 9: Controle de um mesmo sistema para o caso (a) manual e (b) automático. Adaptado de [12].

2.3 AQUISIÇÃO DE DADOS

O intuito dos experimentos em sua maioria se trata de obter dados que serão analisados com um determinado fim, para isso é necessário que eles sejam armazenados em algum lugar. Com computadores isso se tornou mais simples, basta possuir um *software* obtendo e armazenando os valores obtidos por um *hardware,* mas o problema é que valores reais e valores digitais são, naturalmente, diferentes.

Para os computadores, valores são dados em Bits na sua memória, que são discretos, possuindo apenas dois valores: 0 ou 1, verdadeiro ou falso. Para obter um valor maior, esses bits são ordenados em outros tamanhos, assim um Byte, possui na verdade 8 bits e permite que seja obtido valor de 0 até 255, ou $2^8 - 1$. Com isso, quantos mais bits forem rearranjados temos

que maior é o valor possível de se obter, até que seja atingido o limite do computador. O limite padrão está se tornando 64 bits, assim teríamos de 0 até $2^{64} - 1$ valores possíveis.

Entretanto, o *hardware* realiza medida de valores reais, o que significa que ele está obtendo dados analógicos, que por sua vez são contínuos e não quantizados, passando a ter uma inconsistência com o digital. Isto é, entre o valor 0 e 1 para um sinal analógico teríamos infinitos valores, como 0,1, 0,0000052 e etc., entretanto para o sinal digital estamos limitados a 0 e 1 discretos. Para resolver este problema foi-se determinado o padrão IEEE 754 [13], assim o computador consegue obter valores com ponto flutuante (com vírgula) e negativos, mesmo utilizando bits. E para obter o sinal é utilizado um transdutor, este realiza uma conversão de analógico para digital e vice-versa. Essa conversão permite que o *software* seja capaz de realizar a leitura dos dados obtidos pelo *hardware* e tratá-los.

Na figura 10 estão os diversos tipos de aquisição de dados que são possíveis de serem realizados, seja diretamente ou indiretamente. Como vemos, o software não consegue interagir diretamente com o processo, primeiramente ele passa por um *hardware* de aquisição, Este, como veremos a seguir, faz a conversão para realizar o controle e obtenção de dados. Temos ainda a aquisição de dados indireta, que consiste no armazenamento dos dados em um banco de dados ou na nuvem. Pode vir a ser muito útil no caso em que experimentos são realizados durante várias horas e o usuário não tem necessidade de supervisionar diretamente o software que realiza a aquisição de dados no local do experimento, podendo fazer isso a partir de outras localidades.

O gerenciador de experimento da figura consiste no *software*, através dele o experimento é organizado e controlado. Alguns possuem interface gráfica (GUI) para facilitar o uso, assim como outras funções adicionais que não são comuns ao usuário, como exibição em tempo real dos dados já tratados.

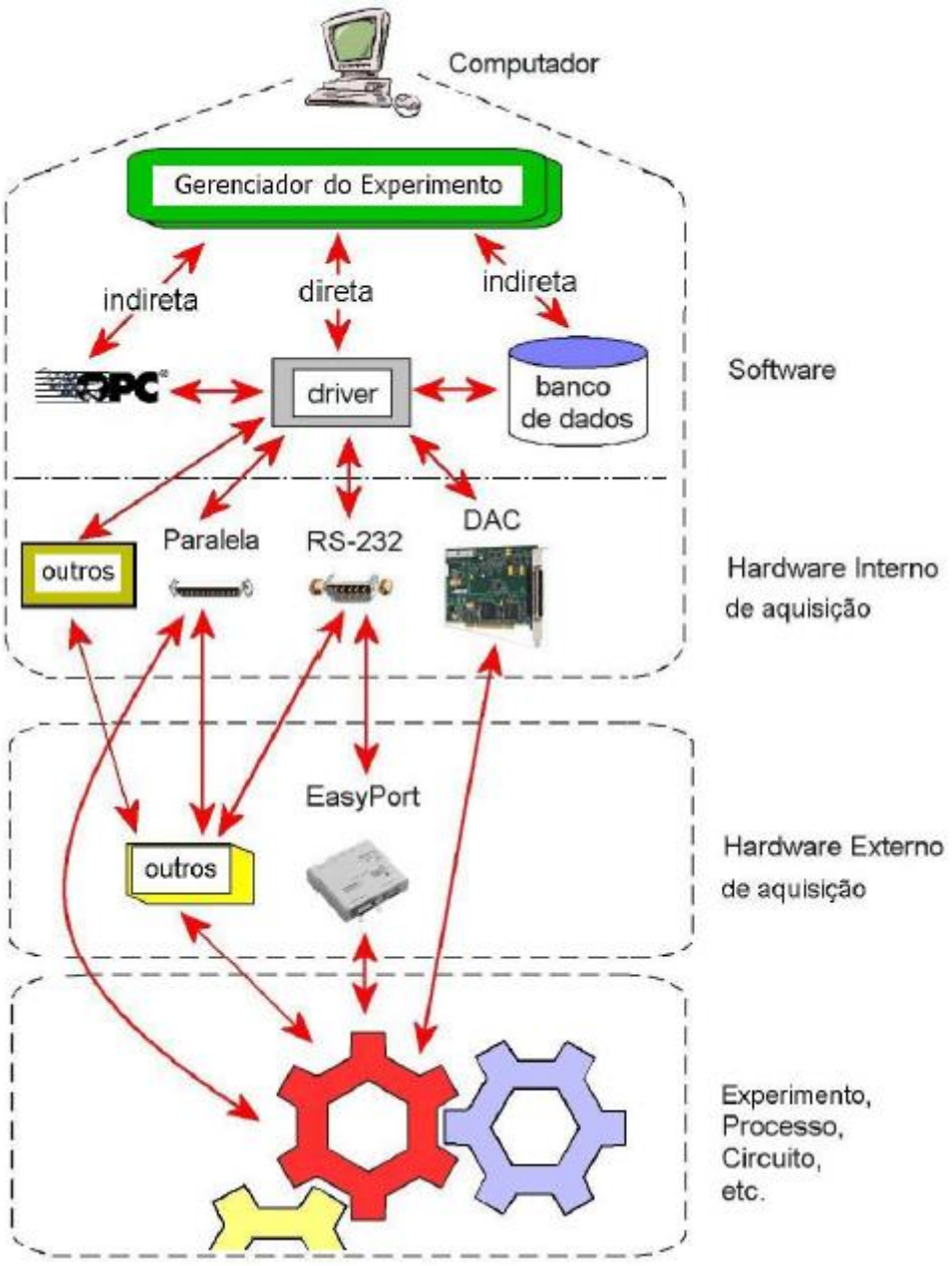
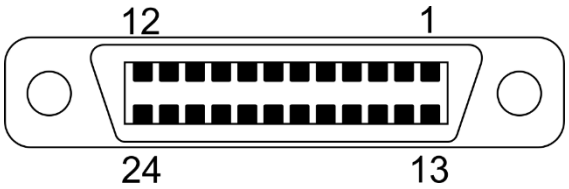


Figura 10: Diversos modos de aquisição de dados. Adaptado de [14]

2.4 INTERFACE GPIB

Existem diversos protocolos de comunicação que permitem que equipamentos se comuniquem com o computador, como: TCP/IP, USB, RS-232, GPIB e outros, em que cada fabricante desenvolve *drivers* que permitem através da leitura do barramento obter dados e controla-los. Com o intuito de se facilitar essa comunicação, foi determinado o padrão SCPI [15], que é uma série de comandos para programar instrumentos, assim é possível desenvolver *softwares* mais gerais.

A interface GPIB em particular, possui barramento paralelo de 8 bits, visto na figura 11, que possui 24 linhas de sinal sendo elas: 8 para transferência de dados, 3 para *handskake* e 5 para controle, além de 8 para o terra.



Sinal	*Pino*	*Função*
DIO1	*1*	*Dado/commando*
DIO2	*2*	*Dado/commando*
DIO3	*3*	*Dado/commando*
DIO4	*4*	*Dado/commando*
EOI	*5*	*Fim ou identidade*
DAV	*6*	*Dado inválido*
NRFD	*7*	*Sem leitura pro dado*
NDAC	*8*	*Sem dado aceito*
IFC	*9*	*Interface limpa*
SRQ	*10*	*Requisição de serviço*
ATN	*11*	*Atenção*
Shield	*12*	*Cable shield*
DIO5	*13*	*Dado/commando*
DIO6	*14*	*Dado/commando*
DIO7	*15*	*Dado/commando*
DIO8	*16*	*Dado/commando*
REN	*17*	*Controle remoto*
Terra	*18...24*	*Terra*

Figura 11: Barramento IEEE-488 com os 24 pinos nomeados. Adaptado de [16]

Os 8 bits para transferência de dados podem ser manipulados diretamente, entretanto isso requer uma programação abstrata, mas utilizando a interface de programação VISA [15], que será discutida mais à frente, é possível interagir através dos comandos definidos pela SCPI. Os 3 bits para *handshake* servem para definir que a conexão entre o equipamento e o *software* foi estabelecida com sucesso ou falhou. Os 5 para controle estão presentes e bem definidos em

todos equipamentos, pois eles possuem a sua identificação (ID), variando de 0 a 31 que são definidos no equipamento e em seguida no *software* para localizá-lo.

Neste tipo de interface, o *software* torna-se o controlador, através dele executamos os comandos que passam pelo barramento até o equipamento. Possuindo alguns comandos básicos, mas de vital importância, como "Interface Clear" (IFC) que limpa os dados que podem estar a espera para serem transferidos, ou "Remote Enable" (REN) que torna o equipamento acessível remotamente. Se esse barramento em específico não tiver definido não é possível executar os comandos do *software* resultando em erros, além de outros.

A interface GPIB que possuímos no laboratório é conectada ao computador através da interface USB e é capaz de permitir o controle de até quinze (15) equipamentos. Podemos então realizar medida em diversos equipamentos simultaneamente com baixa latência e, caso ocorram valores mais altos, o motivo deve estar diretamente ligado ao equipamento, mas não a interface em si. A interface vai determinar a taxa de transferência com base no equipamento mais lento que estiver conectado, independente dos outros.

A vantagem de ser via USB, consiste na praticidade de poder movê-la se necessário para outro laboratório, visto que as mais antigas eram conectadas no slot PCI. Mesmo que o barramento PCI seja mais rápido que o USB, é possível atingir com a GPIB via USB uma transferência de 8 MB/s, portanto mesmo que sejam utilizados vários equipamentos simultaneamente ela consegue satisfazer todas as necessidades.

Para a utilização desta interface, podemos conectá-la ao computador como exibido na figura 12. Este é o modo em pilha, existem outras formas mas isso não implica uma significativa menor velocidade de transferência, assim a decisão pode se basear apenas na questão de praticidade, facilidade ou até estética.

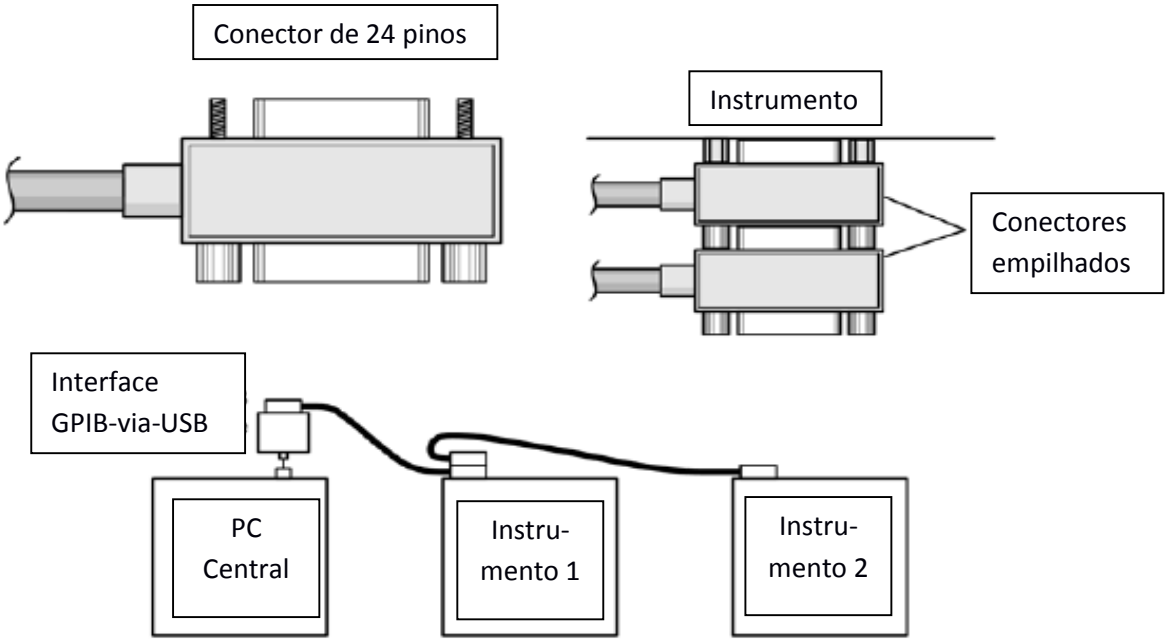


Figura 12: GPIB-via-USB ligada ao equipamento em pilha. Adaptado de [15].

2.5 LINGUAGEM DE PROGRAMAÇÃO

Para o desenvolvimento do *software*, utilizamos a linguagem Python. Foi desenvolvida em 1991 é uma linguagem de alto nível que possuí uma sintaxe clara e organizada, o que a torna ideal para o desenvolvimento de um *software* que pode vir a ser alterado futuramente. Além disso, é uma linguagem que quando bem desenvolvida consegue ter um processamento rápido. A utilização de bibliotecas como Numpy e Scipy, que em conjunto permitem a utilização de várias funções matemáticas, torna simples o tratamento dos dados em tempo real.

Diversas outras linguagens poderiam ser utilizadas para o desenvolvimento do *software*, como C, C++, Visual Basic e outras, mas Python permite que o mesmo trecho de código das outras linguagens seja escrito de forma mais simples. A forma como sua sintaxe é definida torna-a mais legível, principalmente em códigos mais extensos, pois tem um guia de estilo padrão que recomenda-se ser seguido, a PEP 8 [16], o que limita a quantidade de variações que o código pode ter em relação a outros. Em linguagens como C, a liberdade para o desenvolvimento do código é maior, do ponto de vista de sintaxe. Entretanto, isto abre margem para a utilização de comandos obsoletos como o "goto", que permite uma movimentação no código sem uma regra especifica, quebrando fluxo de controle que deveria ser seguido. Obviamente, isso não significa que sejam linguagens ruins, aliás, em termos de performance C++, por exemplo, é superior a Python e possui uma grande comunidade de utilizadores, porém

a curva de aprendizado de Python é muito menor, facilitando a modificação em códigos por um novo usuário.

Poderíamos ao invés do Python, utilizar algum *software* proprietário, como o LabView da National Instrument que permite o desenvolvimento da rotina desejada através de uma interface gráfica, figura 13. Podemos citar pontos positivos para esta abordagem como a de que não iria falhar devido a pequenos detalhes, o que pode acontecer em linguagens de programação, além de compilar rapidamente sem necessidade de configurações adicionais. Porém, esse *software* precisaria primeiramente ser comprado e tem um alto custo, além disso a correção de erros é complicada, pois, quanto maior o número de funções, o estilo de "teia" em que cria as rotinas se torna confuso e estaríamos nos limitando a uma plataforma apenas, *Windows*. Desenvolvendo o *software* com o Python temos a dificuldade de escrever todo o código para a interface gráfica e para os equipamentos, o que é relativamente trabalhoso, com erros básicos de escrita podendo causar erros e a depuração para encontrar problemas requisitando muito tempo. Mas apesar disto tudo, não temos que pagar pelo software, o funcionamento independe da plataforma, a documentação dele é completa, não vai requisitar tanto poder de processamento do computador e observamos em detalhes o funcionamento da interface GPIB se comunicando com os equipamentos.

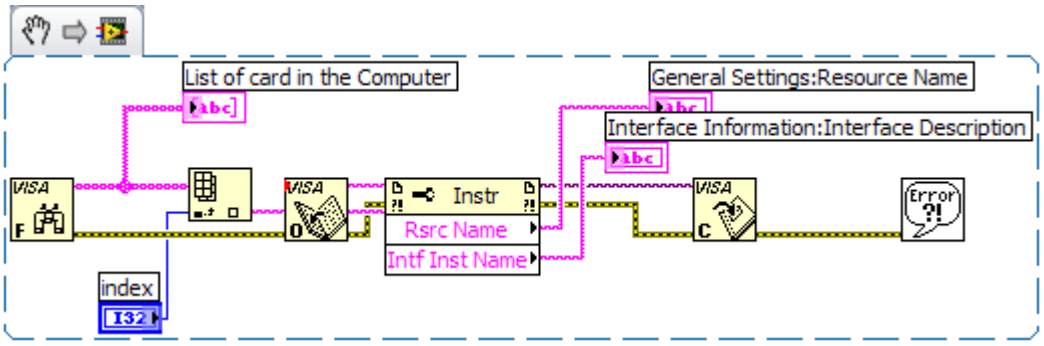


Figura 13: Exemplo de funcionamento de uma rotina no LabView. Adaptado de [17].

2.6 INTERAÇÃO SOFTWARE-EQUIPAMENTOS VIA GPIB

Com o passar do tempo, foi-se percebendo que com o aumento do número de equipamentos, o controle se tornava cada vez mais complexo, pois cada um possuía uma determinada interface e seu determinado modo de operação, o que não tinha necessidade, pois a base de todos era sempre enviar e receber informações. Deste modo, diversos protocolos passaram a ser criados para simplificar isso, e de modo geral, não surgiram muitos a ponto de fazer o problema voltar a origem. Um desses protocolos foi então o IVI [15] (Interchangeable Virtual Instrument), composto pela SCPI que definia um conjunto de comandos padrão e a VISA que é a interface para interação com interfaces tipo GPIB.

O SCPI tem uma grande importância, pois além de ter definido os comandos disponíveis, ainda classificou os equipamentos que podem ser utilizados em sistemas em oito classes:

- Chassis Dynamometers,
  - aparelhos destinado a medir a dinanometria (rpm) do motor.
- Digital Meters,
  - equipamentos de medição, como termômetros, multímetros e outros.
- Digitizers,
  - conversor de desenhos feitos a mão em dados ao computador.
- Emissions Benches,
  - analisadores de gás.
- Emission Test Cell,
  - equipamentos de teste de emisão.
- Power Supplies,
  - fontes de alimentação.
- RF & Microwave Sources,
  - emissores de ondas.
- Signal Switchers.
  - conversores de sinal.

Possuindo então um padrão de comandos que nos permitem realizar a interação com os equipamentos através da GPIB, utilizamos uma biblioteca desenvolvida com esse propósito, denominada PyVisa [18], que converte a base da IVI (visa32.dll) em uma série de comandos. Com ela, a interação se torna mais simples. Isto porque a PyVisa funciona em três níveis, que são:

- **Baixo nível:** neste nível a biblioteca define os tipos, respostas e conversões necessárias de cada função.
- **Médio nível:** funções que acionem as de baixo nível, mas que são necessárias apenas para realizar controle específico da biblioteca Visa.

- **Alto nível:** funções referentes ao controle de recursos, está diretamente ligada ao controle dos equipamentos e é parte essencial do *software*.

Assim, a maior parte do tempo restringimos as funções apenas as de alto nível, pois neste ponto temos todas as funções que nos interessam, isto é, de requisição e leitura de dados. As outras se tornam interessantes quando possuímos alguma opção básica como acesso remoto do equipamento que pode estar falhando, então acessamos funções de nível médio e nos comunicamos com o equipamento para modifica-lo "internamente".

Através das funções de alto nível, conseguimos no código observar todos os equipamentos ligados a interface, escolher qual desejamos controlar e realizar alguma requisição, como no exemplo 1, em que através de uma simples rotina obtemos uma medida de tensão.

```
>>> import visa
>>> rm = visa.ResourceManager()
>>> eq = rm.get_instrument("GPIB0::20::INSTR")
>>> print(mt.ask("MEAS:VOLT:DC?"))
```

Exemplo 1: Definição do controle de um equipamento e requisição e leitura de um dado.

Temos que, na linha de comando um, selecionamos que queremos usar a biblioteca PyVisa que é chamada apenas de *visa*. Na linha dois iniciamos o controle dos recursos para na linha três obtermos o controle do equipamento que possuir o ID de número vinte, que esteja ligado a GPIB. Agora já podemos fazer requisições, como a da linha quatro, em que enviamos o comando "MEAS:VOLT:DC?" para medir uma tensão em corrente contínua. O comando utilizado é um exemplo de como a SCPI funciona, ela define que se mudarmos apenas o "VOLT:DC?" para "RES?" já passaríamos a realizar outra operação. No caso, em vez de medirmos tensão contínua seria medido uma resistência.

2.7 LATÊNCIA

Quando se pensa em desenvolver um sistema que precisará medidas em diversos equipamentos, além da preocupação de controla-los ainda há uma maior, que consiste em quão rápida a interação é. Quanto mais demorada é a interação computador-equipamento, pode ser que isso impeça a realização de medida em outros equipamentos. Este tempo de resposta é chamado de latência (ou *delay* em inglês).

Planejando o sistema é necessário observar a latência causada por cada componente. A nossa GPIB via USB, possui uma latência de $1000\,\mu s$ e concluímos que ela satisfaz as necessidades, entretanto ainda poderíamos optar por uma GPIB via PCI/PCI-E que tem latência de $0{,}7\,\mu s$, mas que possuí as dificuldades citadas na subseção 2.4. Para o multímetro Agilent 34001A, que utilizamos, essa preocupação é definida através do *tempo de integração* conforme dado pela tabela 1. Este é o período que o multímetro converte o valor da medida de sinal analógico para digital, dado em *NPLC* (Number of Power Line Cycles) que indica o número de vezes que o sinal de entrada é integrado para obter uma única medida.

Tempo de Integração	Resolução	Digitos	Leituras por segundo
0,02 NPLC	0,0001 x Range	4 ½	3000
0,2 NPLC	0,00001 x Range	5 ½	300
1 NPLC	0,000003 x Range	5 ½	60
10 NPLC	0,000001 x Range	6 ½	6
100 NPLC	0,0000003 x Range	6 ½	0.6

Tabela 1: Tempo de integração e seus seguintes resultados.

Assim vemos que, se a resolução não fosse um fator crucial, poderíamos utilizar um NPLC de até 0,02 e mesmo com uma grande quantidade de componentes não teríamos uma latência alta. Entretanto, precisamos da maior precisão possível e que sejam assíncronas, para isso introduziremos o conceito de *Threads [19]*, ou *Encadeamento de execução*.

CAPÍTULO 3 – O SOFTWARE

3.1 OBJETIVO

O objetivo do *software* consiste em monitorar uma propriedade Física em função da temperatura. Através disto os dados são salvos, podendo em sequência serem analisados e retiradas informações sobre a amostra estudada. O monitoramento deve ocorrer de maneira que a latência apresente o menor valor possível, assim garantindo que as medidas correspondem a um mesmo instante, ou seja, sejam síncronas. Os dados obtidos devem ser salvos em um arquivo *ASCII*, para serem abertos em um programa como o *OriginLab* em forma de colunas, permitindo sua análise.

No caso deste estudo, a figura 14 representa a montagem experimental utilizada. Nela podemos ver os equipamentos eletrônico utilizados na medida de transporte elétrico em função da temperatura. Temos um termoresistor calibrado que medimos a tensão e somos capazes de obter a temperatura com alta precisão. No lugar da amostra, utilizamos um termoresistor não calibrado de platina, assim também medimos sua tensão, mas calculamos apenas a resistência. Utilizando a temperatura do calibrado e sabendo a resistência do não calibrado, seremos capazes de calibrá-lo.

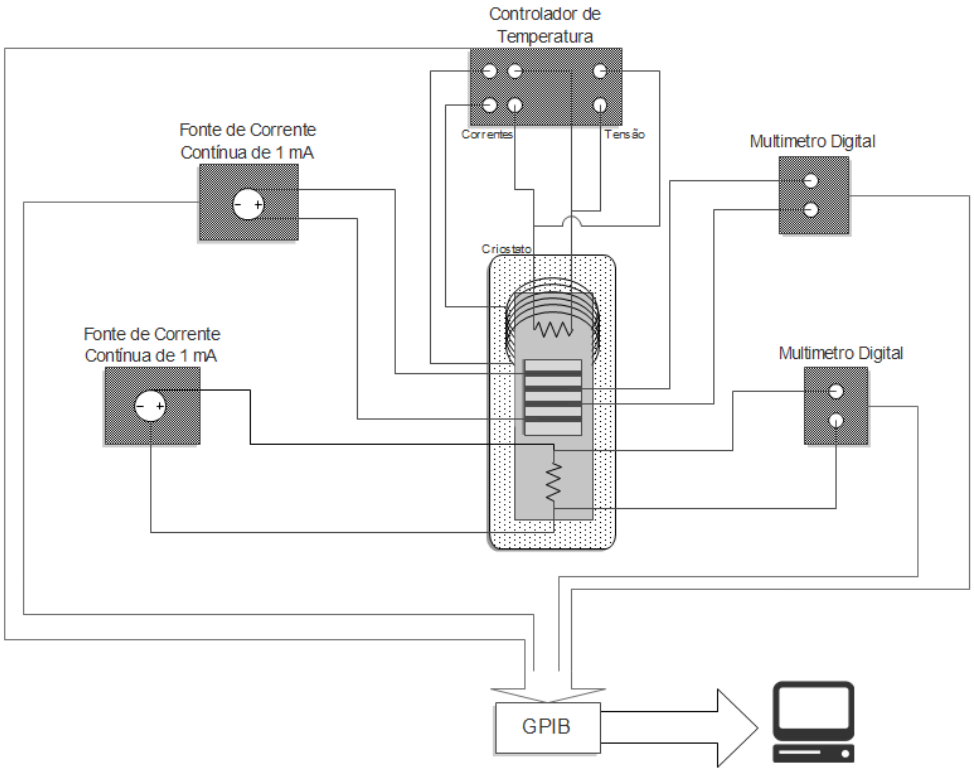


Figura 14: Diagrama representando a montagem experimental utilizada.

3.2 DESENVOLVIMENTO

Para iniciar o desenvolvimento, definimos que o *software* fosse modular, sendo prático para criar e modificar novas rotinas. Isto é interessante, pois as necessidades da medida vão variar para cada experimento, além de permitir que o *software* seja até mesmo utilizado em outros laboratórios. Como podemos ver, a figura 15 é a interface inicial ao iniciar o *software*, nela podemos ver as rotinas já criadas, abri-las, edita-las ou criar novas.

Figura 15: Interface inicial do software.

Utilizamos o software *QtDesigner* [20] para criar as interfaces, pois é um processo trabalhoso se feito manualmente, com a ferramente podemos através de uma interface gráfica criar a interface do nosso programa, sendo necessário apenas desenvolver em código as funções. Para as funções de cada botão da interface foi desenvolvido o código *main.py* (anexo x), responsável por todo o controle de exibição de dados até a interação com os módulos.

Para criação de um novo módulo o usuário deve seguir um padrão, isto é, o código deve ter a seguinte estrutura:

```
import visa
from collections import OrderedDict
def run(constantes):
  ...
  dados = OrderedDict()
  …
  return dados
def equips(ID):
  …
  return equipamento
def main():
  requisitar = []
  return requisitar
```

As três funções foram determinadas para funcionar em conjunto com os botões básicos da interface. Ao iniciar o módulo a função *main* é executada e se houver algum valor em *requisitar* então aparecerá uma caixa de texto na interface.

Ao selecionar o ID do equipamento na interface geral, figura 16, a função *equips* é executada para realizar a primeira interação módulo-usuário. Para o nosso caso, com os multímetro, obtemos a localização dele via GPIB e limpamos todos os dados que podem estar a espera para serem lidos de alguma outra medida anterior, a fim de não gerar erros as novas medidas.

A função *run* é iniciada quando se decide iniciar a medida e será repetidamente executada até o usuário decidir parar através do botão da interface. O argumento *constantes* consiste dos dados que foram requisitados pelo módulo via função *main* e inserido pelo usuário na interface. No nosso caso, esta função é onde se realiza a medida, calcula os valores de interesse e retorna os dados para serem exibidos via gráfico pela interface.

Os módulos desenvolvidos até o momento são o *TensaoTemperaturaDupla.py* (apêndice B) e o *TensaoTemperaturaDuplaComThread.py* (apêndice C), ambos com o propósito de medir duas tensões, calcular uma temperatura, exibir e salvar os dados.

Na figura 16 podemos ver como é a interface para os dois módulos, já que possuem as mesmas funções em questão de visualização, ainda temos a escolha de onde serão salvos os dados que forem obtidos e calculados.

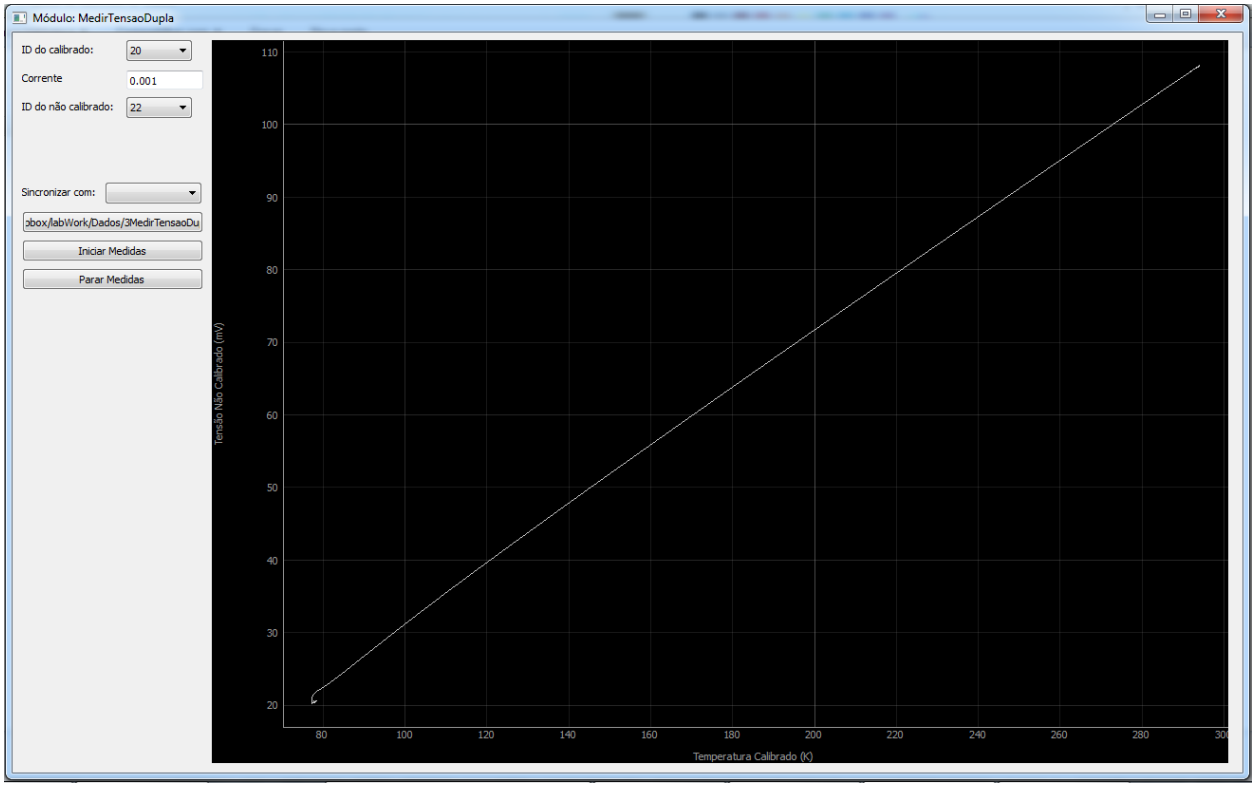


Figura 16: Interface dos módulos com a curva obtida pela medida.

Conforme as necessidades surgirem, uma maior quantidade de módulos serão desenvolvidos e estarão disponíveis. Deste modo, com o *software* generalizado, em um determinado ponto espera-se que não seja mais necessário o desenvolvimento de novos módulos, devido a já estarem disponíveis. Mas se for preciso, uma noção de Python básica e o seguimento do padrão definido tornará o processo simples.

3.2.1 TensaoTemperaturaDupla.py

Este primeiro módulo possui as funções básicas necessárias para a medida, mas com alta latência. A função dele consiste primordialmente em ler a tensão de dois multímetros, selecionados na interface. Um dos termoresistores que utilizamos é um PT-103-14L, cuja resistência em resposta a variação de temperatura é descrita pela curva da figura 17, devidamente calibrado. Ao realizar a compra deste, o fabricante nos fornece um polinômio, no caso de Tchebychev, como explicito no apêndice A, que permite obter a temperatura sabendo a resistência. O código dos módulos consistem em grande parte da conversão deste polinômio para linguagem Python.

Na função *main* do módulo, há a requisição de corrente que irá aparecer na interface,

assim na função *run* ocorre a interação com os multímetros para obter as tensões e calcula-se a resistência, com esses valores ele passa pelo polinômio e calcula a temperatura do ambiente. A cada uma dessas interações ele ainda informa o tempo que o módulo levou, permitindo, assim que gerar a saída de dados, que se saiba o que está gerando a latência.

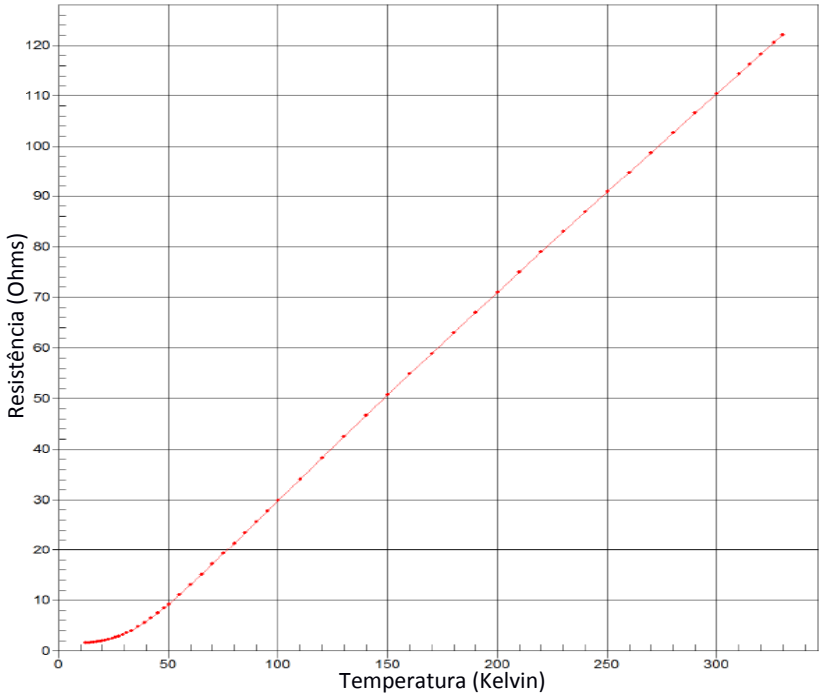


Figura 17: Resposta do termoresistor calibrado ao variar a temperatura. Adaptado de [21].

3.2.2 TensaoTemperaturaDuplaComThread.py

Este módulo funciona exatamente como o anterior, com a diferença que se passa a utilizar a biblioteca *threading*, como dito na subseção 2.7. No apêndice B e C, podemos observar a simples diferença, que consiste em realizar a requisição de medida dos dois multímetros em sequência, mas apenas faz a requisição de dados após ambos terem sido informados. Como veremos, isso permite que a latência da medida decaia para valores desprezíveis.

3.3 ARQUIVO DE SAÍDA

O interesse de uma medida não é apenas observar um gráfico na interface, mas sim tratar os dados de um certo modo. Com isso, é necessário que o *software* retorne os valores ao usuário em um arquivo de saída. No *software* desenvolvido, o arquivo de saída se trata de um *ASCII* com os valores em forma de colunas. Os valores a serem salvos são determinados pela variável *dados* que a função *run* retorna para a interface, deste modo o arquivo gerado é limitado ao que o usuário necessita, sem retornar valores desnecessários. Por exemplo:

Se determinarmos o seguinte retorno na função *run:*

```
def run(constantes):
  ...
  dados = OrderedDict()
  dados[“Temperatura”] = temperatura
  dados[“Tensao”]  = tensão
  dados[“Corrente”] = corrente
  return dados
```

Exemplo 2: Definição de quais valores serão salvos no arquivo de saída.

Teríamos como arquivo de saída um *ASCII* na forma:

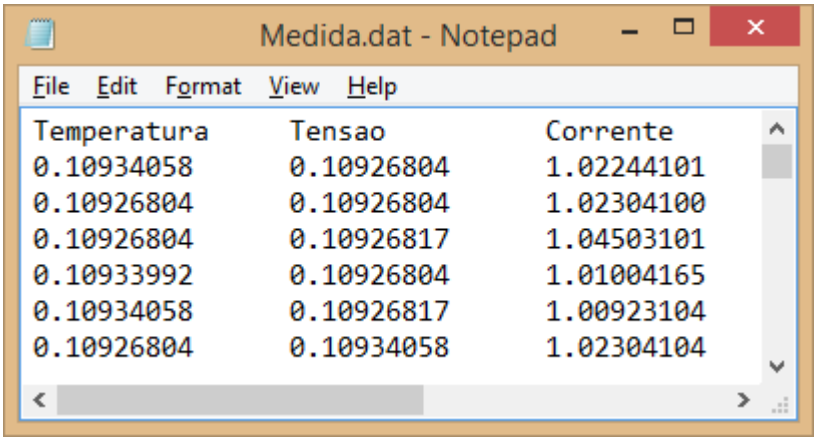


Figura 18: Exemplo de um arquivo de saída.

CAPÍTULO 4 – RESULTADOS

Para obter os resultados, foi-se realizado o seguinte procedimento:

1º A cana com dois termoresistores de platina, sendo um calibrado e outro não, inicialmente em equilíbrio térmico com o ambiente, foi inserida em um recipiente que possuía nitrogênio líquido;

2º Começamos as medidas pelo *software* e submergimos a extremidade da cana;

3º Assim que observamos que atingiu a temperatura de aproximadamente $77\,K$, sabemos que entrou em equilíbrio com o nitrogênio e então removemos a cana;

4º Deixamos a cana entrar em equilíbrio térmico com o ambiente, novamente não interagimos com a variação, apenas esperamos enquanto o software capta os dados.

Na primeira medida realizada, o objetivo foi a obtenção dos valores das tensões e o cálculo da temperatura. Através desse procedimento, pudemos obter os dados conforme a figura 19. Como podemos ver, o arquivo *ASCII* retornou os dados obtidos separados em colunas.

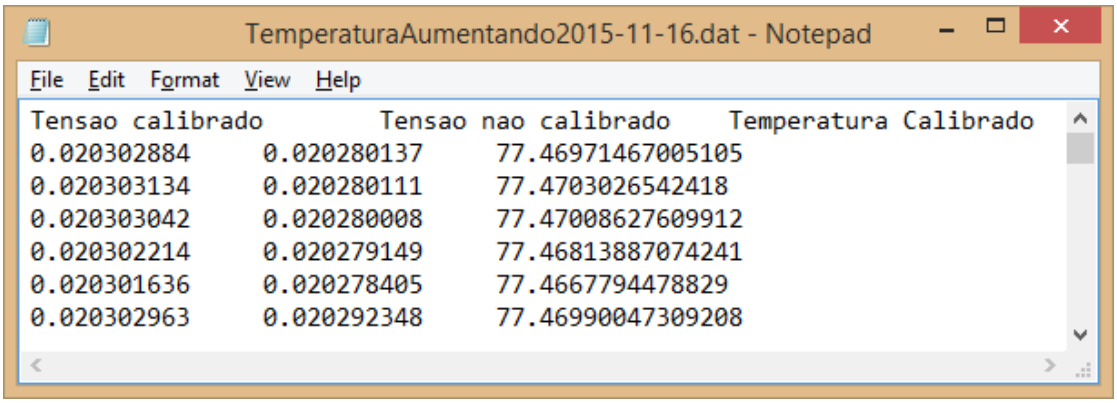


Figura 19: Arquivo de saída da primeira medida realizada.

Após o êxito na obtenção dos valores das tensões e do cálculo da temperatura, passamos a nos preocupar com a latência das medidas. Constatamos que era necessário saber quanto tempo estava levando, para verificar se havia necessidade de mudanças. Assim, fizemos uma mudança no módula, utilizando a biblioteca *time* do Python. Inserimos no módulo uma função que requisitava o tempo entre a requisição e a medida, com isso pudemos gerar um novo arquivo de saída com a estrutura de uma matriz, para registrar a latência da medida, como podemos ver na figura 20.

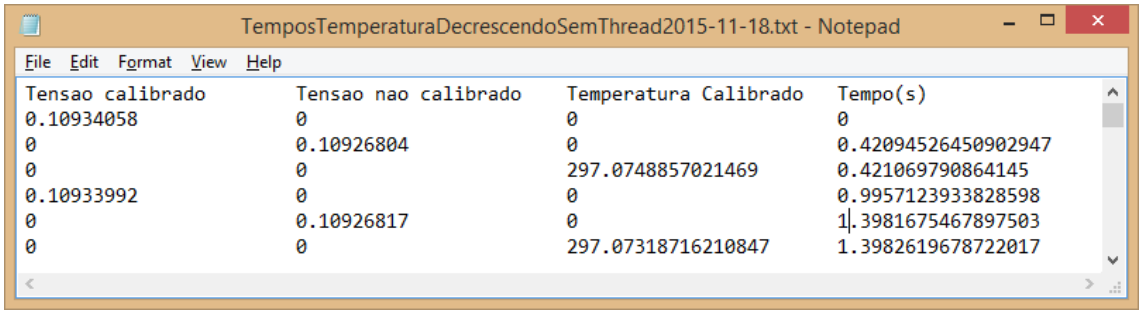


Figura 20: Arquivo de saída em forma de matriz para o módulo sem Thread.

Nesta saída de dados temos quatro colunas, onde cada uma tem o respectivo tempo que levou para ser realizada a medida, com a referência sendo a primeira medida de tensão. Percebemos com isso que o tempo de uma medida para outra teve uma latência de *420,9 ms*, gerando uma alta latência. Portanto, não podíamos ter confiança que ambas estivessem em uma mesma temperatura. Para resolver este problema de latência, como citado anteriormente, utilizamos *threads*. Esta é uma forma de fazer com que o *software* seja separado em mais de uma tarefa, assim ele pode realizar mais de uma requisição sem que haja dependência. Com isso, não é necessário que espere um multímetro retornar o valor para realizar a requisição de outro multímetro. Para isso, foi realizada a seguinte alteração no código:

Antes	Depois
import visa ... def run(constantes): equipamento1 = constantes['equipamento1'] equipamento2 = constantes['equipamento2'] tensao1 = equipamento1.ask('MEAS:VOLT:DC?') tensao2 = equipamento2.ask('MEAS:VOLT:DC?')	import visa ... from multiprocessing.pool import ThreadPool def measureDC(equipamento): return equipamento.ask('MEAS:VOLT:DC?') def run(constantes): pool = ThreadPool(processes=2) async_result1 = pool.apply_async(measureDC, (constantes['equipamento1'],)) async_result2 = pool.apply_async(measureDC, (constantes['equipamento2'],)) tensao1 = async_result1.get() tensao2 = async_result2.get() …

Utilizando *thread* fazemos com que a iniciação da medida dos dois seja realizada ao

mesmo tempo, obtendo o valor apenas quando elas terminarem. Diferente do outro modo, em que o *software* permanecia inativo até receber o valor de um e então faria a outra requisição. Com isso fomos capazes de obter um novo arquivo de saída:

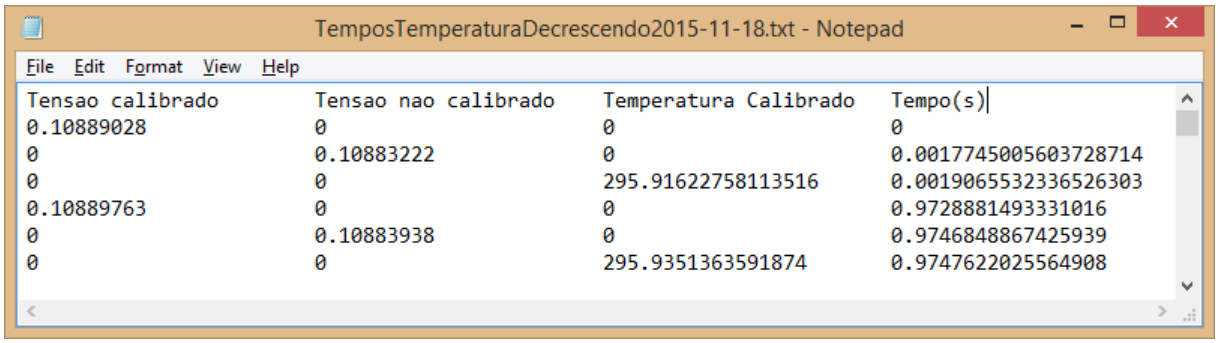


Figura 21: Arquivo de saída em forma de matriz para o módulo com Thread.

Podemos observar, figura 21, que a diferença de tempo entre a medida das tensões foi de *1,7 ms* e o cálculo da temperatura ambiente se torna mais preciso. Esse novo módulo permite que tenhamos mais confiança nas medidas, pois é vital ao experimento que as medidas sejam realizadas simultaneamente. Além de que podemos permitir com *thread* que a medida tome o mesmo tempo que a normal para realizar a medida total. Na figura 22 é explicito como as medidas sem *thread* não podem ser utilizadas devido a latência.

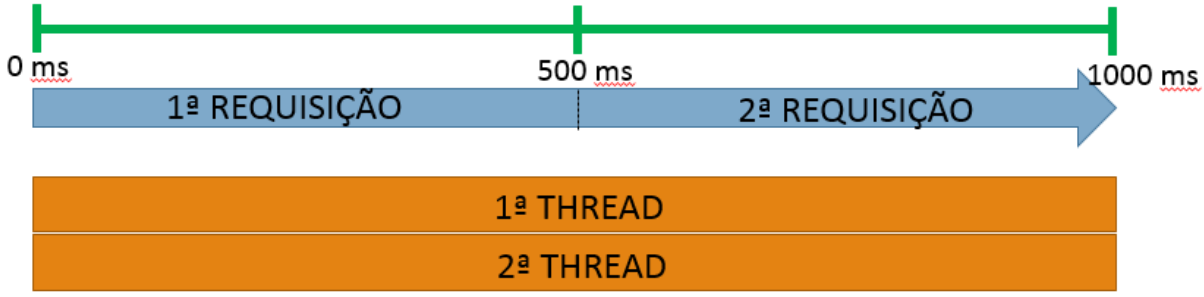


Figura 22: Diferença de sincrona com utilização de Threads.

Sabendo a eficácia da rotina com *thread*, realizamos a medida a partir da qual mostraremos os resultados. Na figura 23, podemos observar um resultado esperado, pois ambos são de platina, então suas tensões possuem valores muito próximos e tem resposta linear.

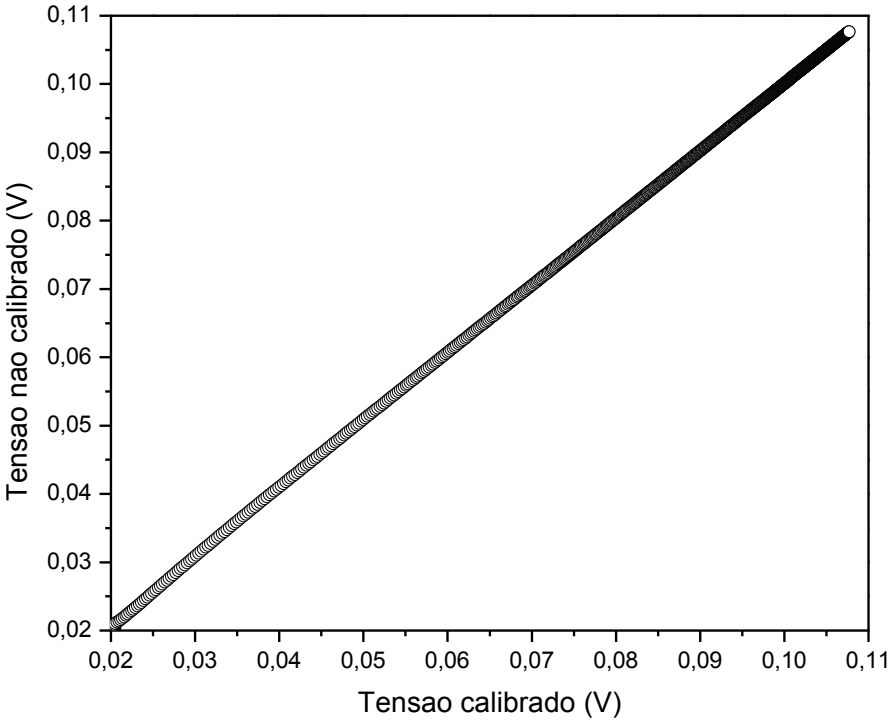


Figura 23: Medida da tensão nos dois termoresistores.

Como aplicamos uma corrente de *1 mA*, de uma fonte estável, através da equação 1.3 e dos dados obtidos, pudemos calcular a resistência do termômetro calibrado obtido. Sabemos que ele possui uma equação na qual a temperatura, dada em kelvin, é dependente da resistência. Obtemos com isso a figura 24, onde podemos observar que em temperaturas baixas a curva está mais clara, pois este é o momento de maior gradiente em relação a temperatura ambiente.

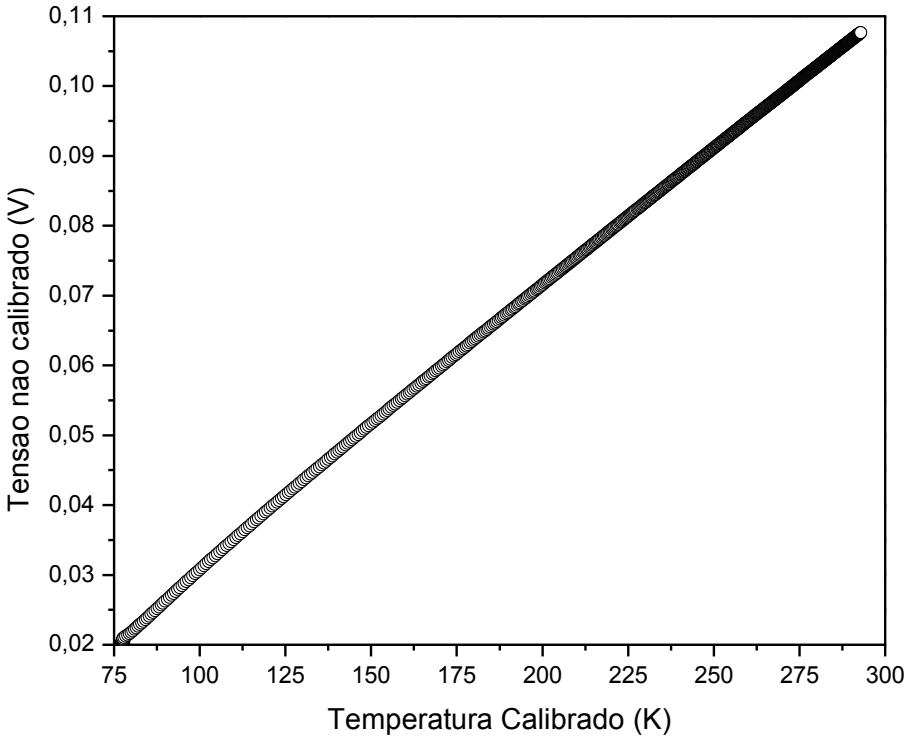


Figura 24: Variação da tensão medida do termoresistor não calibrado pela temperatura calculada com o calibrado.

Podemos agora tratar do termômetro não calibrado, pois aplicamos a mesma corrente nele. Fazendo isto, conseguimos obter a figura 25, que consiste em um gráfico aproximado do que obtêm-se do fabricante para o termoresistor calibrado. A diferença foi que a nossa temperatura não foi baixa o suficiente para vermos a curva para a resistividade residual.

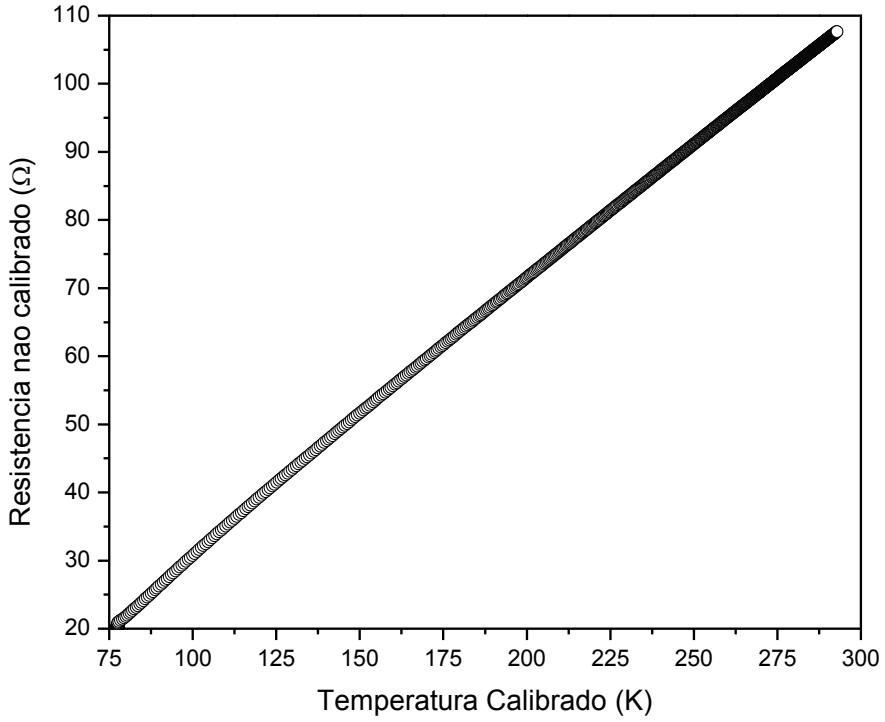


Figura 25: Variação da resistência do termoresistor não calibrado pela temperatura obtida pelo calibrado.

Mesmo o termoresistor que utilizamos como amostra para validar o *software* não sendo calibrado, em temperatura ambiente sua resistência é de aproximadamente $100\ \Omega$,. Para calibrá-lo, podemos ajustar a curva da figura 25 com algum polinômio, para obtermos a temperatura em função da resistência do não calibrado. O primeiro ajuste foi realizado com o *software* OriginLab, visto na figura 26, com a reta determinada pela equação $y = -9.6153 + 0.40174 * x$.

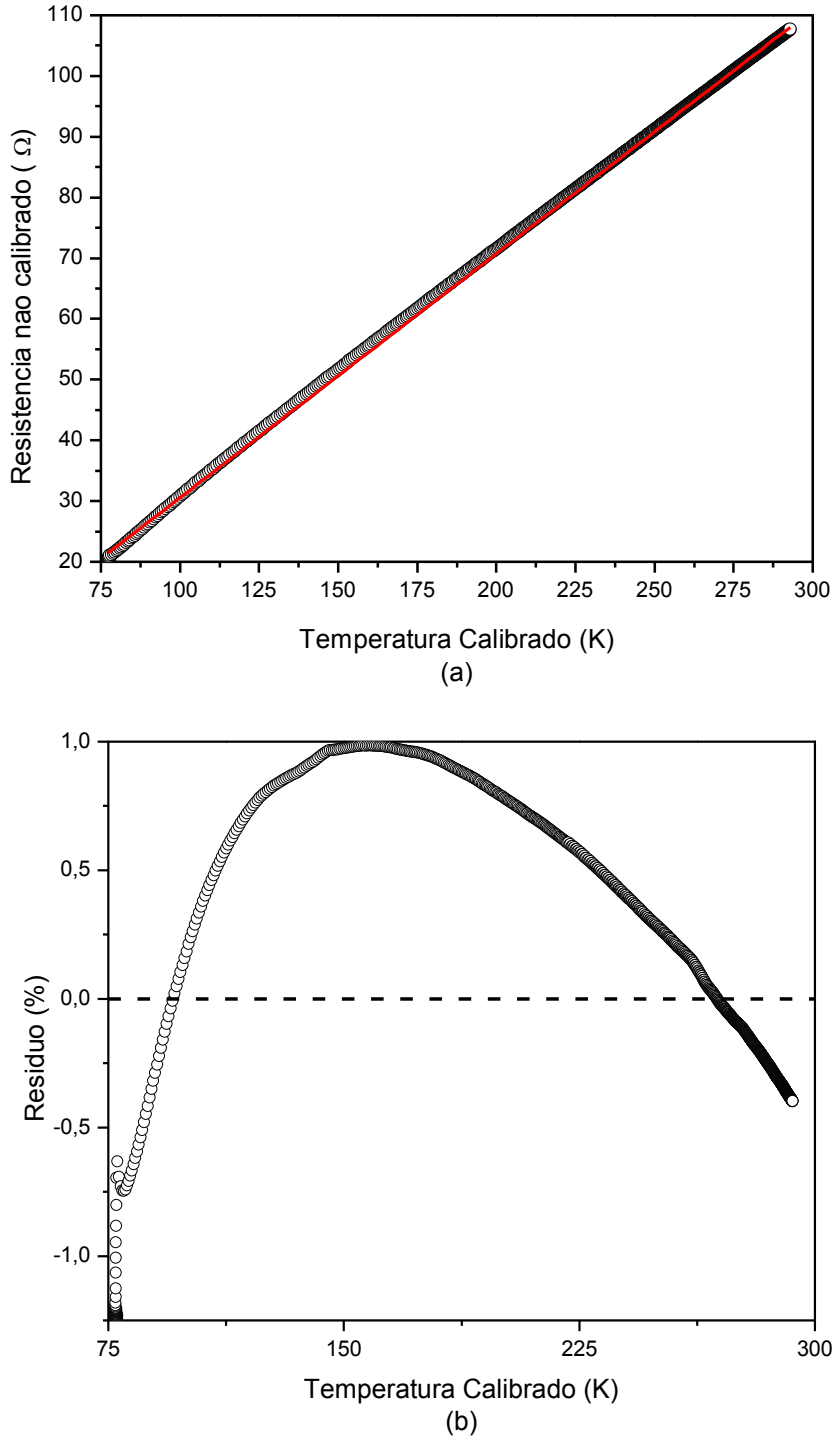


Figura 26: Ajusta da curva utilizando a equação de uma reta. (a) Medida completa. (b) Resíduo do ajuste em relação aos dados medidos.

No painel (a) da Figura 26 a visualização em escala de medide completa, o ajuste a príncipio parece ser razoável. No entanto, no painél (b) vemos que o resíduo não tende a zero como deveria. Para toda a variação da temperuta o resíduo costuma variar de 1% á 0.5%. Vemos com isso que apesar de aparentemente satisfazer a necessidade, há vários problemas que tornam

inprescindível o ajuste com um polinômio mais complexo. Com o intuito de resolver esse fato, fazemos então o ajuste na mesma curva utilizando um polinômio, da seguinte forma:

$$y = -23.65005 + 0.69049 * x - 0.00201 * x^2 + 6.06606 * 10^{-6} * x^3 - 6.92503 * 10^{-9} * x^4 \quad (4.1)$$

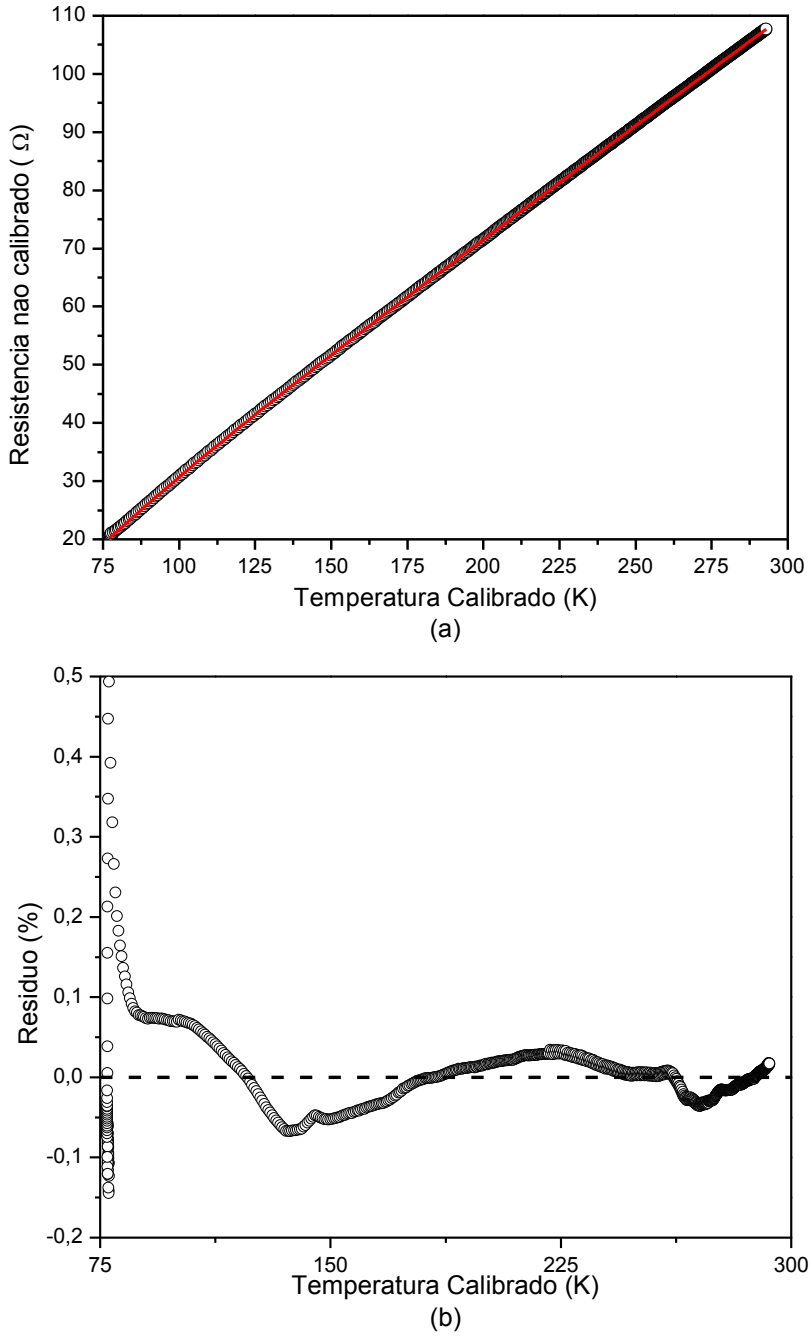


Figura 27: Ajusta da curva utilizando um polinômio de quarta ordem. (a) Medida completa. (b) Resíduo do ajuste em relação aos dados medidos.

Novamente o painel da figura 27 (a) deixa o ajuste aparentemente razoável. Olhando no painel (b) vemos que o resíduo possuí valores altos em baixas temperaturas, mas em seguida passa a ter valores na vizinhança de 0%, sendo assim aceitável. Deste forma, podemos, a partir de agora, utilizá-lo para a determinação da temperatura, pois este encontra-se calibrado.

CAPÍTULO 5 – CONCLUSÕES

O *software* permite que sejam realizadas medidas com diversos módulos, através da interface gráfica, com registro dos dados em baixa latência, permitindo a consideração de uma boa sincronia das medidas realizadas. Ao utilizar um termoresistor não calibrado como amostra, pudemos calibrá-lo através dos dados obtidos, e da temperatura calculada pelo termoresistor calibrado.

Como perspectivas futuras, pretende-se automatizar um controlador de temperatura LakeShore e implementar a eletrônica necessária de um resistômetro diferencial. Com isso, será possível realizar medidas de transporte elétrico com corrente alternada, permitindo controlar a variação da temperatura e assim realizar experimentos em amostras de materiais com interesse em pesquisa básica e aplicada.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] **“Manual Agilent 82357B USB/GPIB Interface.”** Agilent Technologies (2014).

[2] **“Python 3.5.1rc1 documentation.”** Python Software Foundation (2015). Acessado 1 de Dezembro, 2015, https://docs.python.org/3/ .

[3] e [10] Girotto, Emerson M., and Ivair A. Santos. **"Medidas de resistividade elétrica dc em sólidos: como efetuá-las corretamente."** Química Nova 25.4 (2002): 639-647.

[4] Meaden, G T. **“Electrical resistance of metals.”** Springer, 1965.

[5] Ashcroft, Neil W., and N. David Mermin. **"Solid state phys."** Saunders, Philadelphia (1976): 293.

[6] Falconer, Isobel. **"Corpuscles, Electrons and Cathode Rays: JJ Thomson and the ‘Discovery of the Electron’."** The British Journal for the History of Science 20.03 (1987): 241-276.

[7] e [8] Van Delft, Dirk, and Peter Kes. **"The discovery of superconductivity."** Physics Today 63.9 (2010): 38-43.

[9] **“Cryostats Introduction.”** Scientific Products. Acessado 1 de Dezembro, 2015, http://www.asscientific.com/products/cryostats/ .

[11] **“Fundamentos de Controle de Processos.”** Senai (1999). Acessado 1 de Dezembro, 2015, http://sistemas.eel.usp.br/docentes/arquivos/5817066/157/Controle.pdf .

[12] Negri, Victor Juliano. De. **“Introdução aos Sistemas para Automação e Controle Industrial.”** Florianópolis (2014).

[13] Viana, Gerardo Valdisio Rodrigues. **"Padrão IEEE 754 para Aritmética Binária de Ponto Flutuante."** Revista CT 1.1 (1999): 29-33.

[14] Schaf, Frederico Menine**. "Arquitetura para ambiente de ensino de controle e automação utilizando experimentos remotos de realidade mista."** (2006).

[15] Hughes, John M. **“Real World Instrumentation with Python: Automated Data Acquisition and Control Systems."** O'Reilly Media, Inc.", 2010.

[16] **“GPIB-232CV-A User Manual.”** National Instruments (1999).

[17] **“Tutoriais para sua avaliação do LabVIEW: Monitore e registre dados com o LabVIEW Real-Time.”** National Instruments (2014) http://www.ni.com/tutorial/52028/pt/ .

[18] Tantra, Juki Wirawan. **"Experiences in Building Python Automation Framework for Verification and Data Collections."** Python Papers Monograph2 (2010).

[19] Borges, Luiz Eduardo. **“Python para desenvolvedores.”** Novatec Editora, 2014.

[20] Pavan, Theo Zeferino, et al. **"Software for subjective visual vertical assessment: an observational cross-sectional study."** Brazilian journal of otorhinolaryngology 78.5 (2012): 51-58.

[21] Fairfield, Jessamyn. “I’m With The Band Theory”. Letstalkaboutscience (2012).

APÊNDICE A – POLINÔMIO DO TERMORESISTOR PT-103

Para o caso do PT-103, a temperatura foi calibrada para valores de 14K a 325K. Na figura 17 podemos ver o comportamento em todo esse intervalo. Entretanto, para ajustar uma equação a esta curva, o polinômio de Tchebychev foi definido pelo fabricante (LakeShore®) em três faixas: de extrema baixa temperatura, onde vemos praticamente só a residual, 14K a 20,1K; de baixas, onde há uma curva, 20K a 100K; de temperaturas mais elevadas, onde o comportamento se torna quase linear, 100K a 325K.

O polinômio tem portanto a forma:

$$X = \frac{[(Z - ZL) - (ZU - Z)]}{ZU - ZL}$$

$$Temperatura(K) = \sum_i A_i \cos[i * \arccos(X)]$$

Cada termo dessas equações são definidos devido à complexidade da faixa que pertencem, como dado na tabela abaixo.

14K a 20,1K		**20K a 100 K**		**100K a 325K**	
Z = log(resistência)		Z = log(resistência)		Z = resistência	
ZL = 0,19401543303		ZL = 0,26365650889		ZL = 25,68694949357	
ZU = 0,35417036478		ZU = 1,58398501241		ZU = 123,48691646842	
Ordem	**Coeficiente**	**Ordem**	**Coeficiente**	**Ordem**	**Coeficiente**
0	18,103999	**0**	57, 88530	**0**	210,344545
1	5,110398	**1**	46,829291	**1**	121,928768
2	-0,639407	**2**	10,659042	**2**	1,525781
3	0,188277	**3**	3,897883	**3**	-0,134913
4	-0,057209	**4**	0,567510	**4**	0,03418
5	0,027824	**5**	0,247909	**5**	0,002210
		6	-0,027843		
		7	0,019215		
		8	-0,011171		
		9	0,005712		
		10	-0,002581		
		11	0,001796		

APÊNDICE B – *TensaoTemperaturaDupla.py*

Este primeiro módulo consiste em realizar a medida em dois multímetros, em sequência, calcular a temperatura de um dos termoresistor e retornar os valores para a interface gráfica.

```
import visa
import numpy as np
import time
from collections import OrderedDict

def run(constantes):
    equipamento1 = constantes['equipamento1']
    equipamento2 = constantes['equipamento2']
    tensao1 = equipamento1.ask('MEAS:VOLT:DC?')
    tensao2 = equipamento2.ask('MEAS:VOLT:DC?')

    resistencia = float(tensao1)/float(constantes["Corrente"])

    if 1.623 <= resistencia < 1.990:
        ZL = 0.19401543303
        ZU = 0.35417036478
        Z = np.log10(resistencia)
        A = [18.103999, 5.110398, -0.639407, 0.188277, -0.057209, 0.027824]
    elif 1.990 <= resistencia < 29.94:
        ZL = 0.26365650889
        ZU = 1.58398501241
        Z = np.log10(resistencia)
        A = [57.888530, 46.829291, 10.659042, 3.897883, 0.567510, 0.247909, -
            0.027843, 0.019215, -0.011171, 0.005712, -0.002581, 0.001796]
    elif 29.94 <= resistencia <= 120.1:
        ZL = 25.68694949357
        ZU = 123.48691646842
        Z = resistencia
        A = [210.344545, 121.928768,
            1.525781, -0.134913, 0.033418, 0.002210]
    X = ((Z - ZL) - (ZU - Z)) / (ZU - ZL)
    temperatura = 0
    for i in range(0, len(A)):
        temperatura += A[i] * (np.cos(i * (np.arccos(X))))

    dados = OrderedDict()
    dados["Tensao1"] = float(tensao1)
    dados["Tensao2"] = float(tensao2)
```

```
    dados["Temperatura1"] = float(temperatura)
    dados["Tempo1"] = tempo1
    dados["Tempo2"] = tempo2
    dados["Tempo3"] = tempo3

    return dados

def equips(ID):
    rm = visa.ResourceManager()

    equipamento = rm.get_instrument('GPIB::'+str(ID))
    equipamento.control_ren(rm.session)
    equipamento.write('*RST; *CLS')

    return equipamento

def main():
    requisitar = ["Corrente"]
    return requisitar
```

APÊNDICE C – *TensaoTemperaturaDuplaComThread.py*

Este segundo módulo consiste em uma modificação do módulo anterior com a adição de Threads para a requisição das medidas.

```
from datetime import datetime
import visa
import numpy as np
import time
from collections import OrderedDict

from multiprocessing.pool import ThreadPool

tempoInicial = 0

def measureDC(equipamento):
    return equipamento.ask('MEAS:VOLT:DC?')

def run(constantes):
    global tempoInicial
    pool = ThreadPool(processes=2)
    #TENSAO 1 = CALIBRADO

    async_result1 = pool.apply_async(measureDC, (constantes['equipamento1'],))
    async_result2 = pool.apply_async(measureDC, (constantes['equipamento2'],))
    tensao1 = async_result1.get()
    if tempoInicial == 0:
        tempoInicial = time.perf_counter()
        tempo1 = 0
    else:
        tempo1 = time.perf_counter() - tempoInicial
    tensao2 = async_result2.get()
    tempo2 = time.perf_counter() - tempoInicial

    resistencia = float(tensao1)/float(constantes["Corrente"])

    if 1.623 <= resistencia < 1.990:
        ZL = 0.19401543303
        ZU = 0.35417036478
```

```
        Z = np.log10(resistencia)
        A = [18.103999, 5.110398, -0.639407, 0.188277, -0.057209, 0.027824]
    elif 1.990 <= resistencia < 29.94:
        ZL = 0.26365650889
        ZU = 1.58398501241
        Z = np.log10(resistencia)
        A = [57.888530, 46.829291, 10.659042, 3.897883, 0.567510, 0.247909, -
            0.027843, 0.019215, -0.011171, 0.005712, -0.002581, 0.001796]
    elif 29.94 <= resistencia <= 120.1:
        ZL = 25.68694949357
        ZU = 123.48691646842
        Z = resistencia
        A = [210.344545, 121.928768,
            1.525781, -0.134913, 0.033418, 0.002210]
    X = ((Z - ZL) - (ZU - Z)) / (ZU - ZL)
    temperatura = 0
    for i in range(0, len(A)):
        temperatura += A[i] * (np.cos(i * (np.arccos(X))))
    tempo3 = time.perf_counter() - tempoInicial

    dados = OrderedDict()
    dados["Tensao1"] = float(tensao1)
    dados["Tensao2"] = float(tensao2)
    dados["Temperatura1"] = float(temperatura)
    dados["Tempo1"] = tempo1
    dados["Tempo2"] = tempo2
    dados["Tempo3"] = tempo3
    return dados

def equips(ID):
    rm = visa.ResourceManager()
    equipamento = rm.get_instrument('GPIB::'+str(ID))
    rm.visalib.gpib_control_ren(equipamento.session, 6)
    equipamento.write('*RST; *CLS')
    return equipamento

def main():
    requisitar = ["Corrente"]
    return requisitar
```