Universidade Federal do Rio de Janeiro

Escola Politécnica

Departamento de Engenharia Eletrônica

Circuitos Elétricos II - 2015.2

Relatório

Autores: **Leonardo Mendes de Moura Carvalho**

Pedro Goñi Coelho

Avaliador: _____

Antônio Carlos M. de Queiroz

DEL

2015.2

Sumário

Capítulo 1

Introdução

1.1 Objetivo

O software de análise nodal temporal desenvolvido deve resolver circuitos utilizando os métodos implícitos de Adams-Moulton, de ordens 1 a 4 e aceitar os elementos:

- Fontes de corrente e de tensão independentes (DC, pulso, senoide);
- capacitores e indutores lineares invariantes no tempo;
- Resistores, possivelmente lineares por partes;
- As quatro fontes controladas, lineares;
- Chaves controladas a tensão;
- Amplificadores operacionais ideais, de 4 terminais.

1.2 Organização do Documento

O capítulo 2 descreve a estrutura do código.
O capítulo 3 mostra alguns resultados obtidos.

Capítulo 2

Código

2.1 Main.cpp

O arquivo main.cpp contém o loop principal do programa, que controla a iteração do Newton-Raphson, faz as chamadas às classes Netlist e LESolver e faz as principais operações de leitura e escrita de arquivos.

O primeiro passo é a abertura do arquivo indicado no argv[2]. Caso haja problemas na passagem de parâmetros, o programa apresenta uma mensagem de Uso. Caso o arquivo não exista, é exibida também uma mensagem de alerta. O programa aceita qualquer arquivo de texto, mas idealmente foi modelado para aceitar arquivos .net contendo o netlist do circuito a se analisar, nos padrões definidos no enunciado do trabalho.

Com o sucesso da abertura de arquivo, se inicializa um objeto *netlist* da classe Netlist e então acontece a leitura linha a linha do netlist, pulando a primeira linha, que por padrão contém o número de nós. Para cada componente lido, se é adicionado ao objeto *netlist* um objeto da classe correspondente. Foram desenvolvidas classes para cada componente aceito, discutidas mais à frente. Mensagens de erro eventualmente podem ser apresentadas no caso do netlist apresentar parâmetros errados.

As configurações específicas da análise, como tempo final, passo, número de passos internos e a ordem do método de Adams-Moulton também são passadas para o objeto *netlist* e caso haja algum erro ou ausência de configurações a análise é abortada com um erro.

Passada a fase da leitura, fecha-se o arquivo de entrada e chama-se o método de *sortNodeList()* do objeto *netlist*. Este organiza a lista interna de nós do objeto *netlist* o que por sua vez permite a nomeação dos nós com qualquer texto, mas garante que o nó de terra, obrigatoriamente nomeado como “0”, seja o primeiro nó, facilitando a exclusão da primeira linha e da primeira coluna do sistema de equações.

O próximo passo consiste em receber do objeto *netlist* a lista de incógnitas, guardadas em um objeto a ser impresso no final no arquivo de saída.

Finalmente chega-se à parte principal do programa, um loop que se repete enquanto uma variável de tempo, inicializada em 0, não ultrapassar o tempo final configurado, com incrementos definidos pelo passo dividido pelo número de passos internos. Dentro deste loop há um loop de while que é o controle do Newton-Raphson. Enquanto o resultado não convergir para a última aproximação, o programa continua os cálculos, até se atingir um limite definido no arquivo Utils.hpp, MAX_NR1. A primeira aproximação, gerada no momento que o main.cpp pede a lista de incógnitas, parte de 0 para todas as incógnitas menos as voltagens dos nós ligados a fontes de tensão, que já partem de seus valores no tempo 0. Caso o limite MAX_NR1 for atingido e esta for a análise para o tempo 0, um segundo contador é incrementado e gera-se uma nova aproximação a partir de números virtualmente aleatórios para as variáveis que não convergiram. O processo segue até que a convergência seja verdadeira ou que o segundo contador atinja a constante MAX_NR2, ou seja, que a convergência seja testada partindo de MAX_NR2's aproximações. Caso não haja convergência, o programa é abortado com erro de convergência.

O segundo passo dentro do loop principal, visto que o primeiro é checar e incrementar os contadores do Newton-Raphson, é chamar o método *mna()* do objeto *netlist*, passando o tempo e uma booleana que indica se esta é a primeira análise (análise do tempo 0). Esta booleana tem o objetivo de indicar se, no fim da análise para este tempo, os valores anteriores de correntes e tensões dos capacitores e indutores devem ser guardados em todos os campos ou apenas no último, com os valores anteriores atualizados. (Maiores detalhes adiante)

O método *mna()* é o que gera internamente as matrizes das equações. Estas matrizes foram implementadas como vector<vector<double>> e vector<double>, de forma que elas não tem limites pré-estabelecidos. Caso o trabalho não fosse puramente acadêmico, seria necessário um processo de teste com circuitos com grandes números de nós para determinar limites razoáveis para quantidades de memórias usuais em computadores modernos. Entretanto, não há limites, o que eventualmente poderá consumir uma quantidade significativa de memória em circuitos grandes.

O trabalho inicial do objeto *netlist* está feito quando ele retorna estas matrizes para o main.cpp. O que vem em seguida é a passagem destas para

um objeto da classe LESolver, através do método *cramer()*. Como o nome deste indica, ele recebe as duas matrizes e resolve através do método de Cramer, devolvendo a matriz resultado. Este método pode retornar um erro, caso o determinante da matriz seja 0. Isto causaria uma divisão por zero, o que é proibido. Neste caso, é tentada uma nova aproximação inicial para t = 0 ou a análise é abortada, indicando um sistema impossível.

Com a matriz resultado, chama-se novamente um método do objeto *netlist*, o *checkConvergence()*. Esta é a parte final do loop while que controla o Newton-Raphson. Quando a convergência é alcançada, o passo seguinte do loop exterior é o *netlist.getResult()*, o qual acumula, no caso do tempo atual ser um tempo múltiplo do passo configurado, os vetores de saída.

Por fim, se a análise não foi abortada em nenhum momento, o main.cpp imprime o vetor do nome das incógnitas e o vetor que contém os vetores de resposta para cada tempo em um arquivo de saída, localizado na pasta /output do projeto e nomeado pelo arquivo de entrada, com a extensão e separador definidos no arquivo Utils.hpp.

2.2 Utils.cpp e Utils.hpp

Estes arquivos basicamente contém constantes de configuração do programa e implementação de funções auxiliares.

Uma função auxiliar desenvolvida é a *split()*, comumente encontrada em outras linguagens de programção, como python, que basicamente recebe uma string e retorna vetor de strings contidas na string original e separadas por um caracter. Isto é usado principalmente para se ler os parâmetros da netlist, visto que estes são strings separadas pelo caracter de espaço, na string original que seria a linha do netlist.

A outra função, *pos()*, facilmente retorna a posição de uma string em um vetor de strings. Esta é muito utilizada para se determinar as posições de inserção dos parâmetros nas matrizes, pois cada objeto que representa um componente contém os nomes dos nós e o objeto *netlist* contém um vetor com os nomes dos nós.

Segue uma lista com as constantes criadas e seus valores padrão usados para gerar o programa:

- MY_DEBUG: usada para imprimir alguns valores em uma rotina de testes. False;
- DEBUG_APROX: usada para fazer debug especificamente da geração de novas aproximações. False;
- VERBOSE: usada para imprimir o progresso do programa na tela. True;
- INITIAL_STEP_DIVIDER: para t = 0, o programa divide o passo pelo valor desta constante. 10;
- ERROR_TOLERANCE: tolerância percentual usada para o método checkConvergence(). 10E-2;
- MAX_NR1: primeiro controle do Newton-Raphson. 50;
- MAX_NR2: segundo controle do Newton-Raphson. 50;
- RANDOM_RANGE: limite para valores aleatórios. 50;
- OUT_EXT: define a extensão do arquivo de saída. Unix: csv, Windows: tab;
- SEPARATOR: define o separador do arquivo de saída. Unix: “,”, Windows: “ ”;
- WINDOWS: define se o target será um executável Windows, principalmente usado para definir o separador do path dos arquivos, entre “/“ e “\”. Unix: true, Windows: false.

2.3 Classes

2.3.1 Netlist

Esta classe, após o arquivo main.cpp, contém as principais rotinas do programa. Ela contém campos para vetores de nós, nós internos, e componentes além de métodos para populá-los. Com estes vetores, as matrizes são construídas em seu método *mna()*.

O primeiro método importante é o *getUnknownsList()*. O retorno deste método é uma lista com o nome de todas as incógnitas, o que inclui as voltagens dos nós (nomeados “e” + o nome do nó) e correntes de fontes de tensão, indutores e capacitores (nomeados “j” + nome do componente). A implementação trata de percorrer os vetores de elementos, um a um, e

adicionar os nós internos para os cálculos das correntes necessárias, criando também o vetor da aproximação inicial. Por último, é feita a equivalência de nós causada pela futura eliminação de Amplificadores Operacionais, que mapeia as colunas e linhas a serem eliminadas em campos específicos para tal.

O segundo método fundamental é o *mna()*. Como já mencionado, este percorre os vetores de componentes e os insere nas matrizes da análise nodal modificada, inicialmente zeradas. O método também implementa a eliminação de Amplificadores Operacionais, somando as colunas e linhas equivalentes e as retirando, de trás para frente, retirando por fim a linha e a coluna 0, que representam o nó de terra.

Por fim, o método *checkConvergence()* recebe o vetor resultado e o compara com o vetor de aproximação, que contém ou a aproximação inicial ou o último resultado, guardando em um vetor de booleanas quais incógnitas estão convergindo e quais estão divergindo. A convergência é atingida para uma variável se o erro relativo dela estiver dentro de um percentual configurado na Utils.hpp. No final, o vetor de aproximação recebe o resultado, já preparando para o próximo loop do programa principal.

Outros métodos que merecem citação são o newAprox(), que gera novos valores para a aproximação das incógnitas que divergiram, o *updateValues()*, que percorre a lista de indutores e a de capacitores, chamando seus métodos de atualização de valores passados, além dos métodos que adicionam novos componentes e nós às respectivas listas.

2.3.2 Resistor

Uma classe simples, que contém o nome, os nós e o valor de um resistor. O principal método retorna o valor da condutância para montagem da matriz na classe netlist.

Cabe observar que um campo foi reservado para guardar se o resistor está em um ramo controlado por corrente. Porém, depois entendeu-se melhor que o controle a corrente sempre está em um curto, definindo um nó a mais. Isto facilitou bastante a montagem automatizada do netlist, apesar de aumentar o sistema.

2.3.3 NLResitor

A implementação do resistor linear por partes guarda o nome, os nós e os pares de valores que determinam os pontos das retas que formam a curva de condutância do componente.

O modelo de um resistor não linear linearizado é um resistor paralelo a uma fonte de corrente. Assim, a classe tem um método para obter a condutância, que é a inclinação da reta, e um para obter o valor da fonte de corrente, que é o ponto em que a reta cruza o eixo y. A reta a ser selecionada é a que compreende o valor de tensão a qual o componente está submetido, calculado passando os valores da última aproximação para o objeto.

2.3.4 Switch

De forma semelhante ao resistor não linear, a implementação da chave controlada a tensão depende dos valores da última aproximação, porém o cálculo da condutância é bem mais simples: se a tensão sobre a chave for maior do que a tensão de referência, a condutância é Gon. Caso contrário, Goff.

2.3.5 Inductor

Além dos campos normais, o componente possui campos para guardar os valores passados de corrente e tensão. Assim, recebendo a ordem de Adams-Moulton a se utilizar, e o Δt, a classe retorna um cálculo diferente em cima de uma quantidade diferente de valores passados. Seu modelo é um resistor em série com uma fonte de tensão. O cálculo de sua corrente é automática com a análise nodal modificada. No final da análise, O método updateValues() é chamado para atualizar os valores passados. Para a primeira análise, todos os valores passados são atualizados, de forma a garantir que o valor inicial seja usado nos tempos iniciais em ordens

maiores, como especificado. Para as análises seguintes, ocorre a “rotação” dos valores.

2.3.6 Capacitor

A implementação do capacitor é semelhante a do indutor, mantidas as devidas diferenças entre fórmulas para os valores e campos passados relevantes. A outra grande diferença é que para se atualizar os valores passados, é necessário primeiro calcular a corrente sobre o capacitor, já que esta não é calculada automaticamente. Seu modelo é um resistor em paralelo com uma fonte de corrente.

2.3.7 Fontes Controladas

São quatro classes que implementam fontes controladas: VoltAmp (fonte de tensão controlada a tensão, ou amplificador de tensão), CurrAmp (fonte de corrente controlada a corrente, ou amplificador de corrente), TransCondAmp (fonte de corrente controlada a tensão, ou amplificador de transcondutância) e TransResAmp (fonte de tensão controlada a corrente, ou amplificador de transindutância). Suas implementações são basicamente idênticas, pos a diferença no tratamento das fontes controladas está no posicionamento dos valores nas matrizes, realizado pela classe Netlist.

Vale ressaltar, porém, que fontes de tensão criam nós internos no sistema, o que aumenta a ordem do sistema e calcula a corrente no ramo do componente. Controles a corrente também criam nós internos. De forma que, por exemplo, um amplificador de transindutância cria dois nós internos. Um Amplificador de transcondutância, porém, não cria nenhum, e seu valor entra diretamente na parte da matriz destinada a condutâncias.

O entendimento da elaboração dos modelos destes componentes e do conceito dos nós internos foi fundamental para a realização destas implementações.

2.3.8 Fontes DC

Fontes DC são as fontes mais simples. As classes IDC e VDC tem apenas que guardar nomes, nós + e - e o valor da fonte. A classe Netlist trata de colocar o valor no lugar correto.

2.3.9 Fontes SIN

Fontes SIN são similares às DC, só que contém todos campos necessários para guardar seus parâmetros. São geradores de senóides que podem ser deslocadas por um valor DC, atrasadas no tempo e/ou na fase, amortecidas po um fator exponencial configurável e com número limitado de ciclos. Seu valor é calculado dependendo do tempo alvo da análise. Além disso, para manter a continuidade da função, em casos de amortecimento ou atraso em fase e número de ciclos limitado simultâneos, a fonte mantém, no final dos ciclos, o valor final constante.

O parâmetro de número de ciclos, quando configurado em 0, equivale a uma senóide infinita.

2.3.10 Fontes Pulse

São fontes versáteis que podem formar desde funções Gate até dentes-de-serra. Seus parâmetros são tempo até a subida (atraso), tempo de subida, tempo de descida, tempo ligada, tempo total de um período, número de ciclos, além das amplitudes do tempo desligado e do tempo ligado.

Aqui, o número de ciclos em 0 também equivale a ciclos infinitos.

2.3.11 OPAMP

A implementação dos Amplificadores Operacionais é muito simples. A classe apenas guarda o nome e os nós. Foi escolhido realizar o tratamento simplificado através da eliminação de amplificadores operacionais. Com isso, a corrente de saída não é calculada mas os nós equivalentes são eliminados, somando-se e eliminando-se equações e colunas, o que reduz a

ordem do sistema e agiliza a resolução. A eliminação em si está implementada na classe Netlist.

2.3.12 LESolver

A classe solucionadora de equações lineares tem dois métodos principais: um privado, que calcula o determinante de uma matriz, e um público, que executa o método de Cramer para resolver o sistema.

O determinante é calculado recursivamente, multiplicando cada elemento da primeira coluna pelo seu cofator e pelo determinante da matriz de ordem diretamente menor, causada pela eliminação da respectiva linha e coluna, e somando-se todos os resultados.

O método de Cramer utiliza o determinante para achar o divisor de cada resultado. Caso este seja 0, a análise é abortada. Caso contrário, o método substitui a matriz resultado em cada coluna da matriz original para achar o numerador do resultado de cada incógnita.

Capítulo 3

Resultados

A metodologia utilizada para testar o programa foi rodá-lo no mesmo ambiente em que foi desenvolvido, MacOS X, que, sendo um sistema Unix, é compatível com builds para Linux. O formato do arquivo de saída foi configurado para csv, assim, os resultados eram abertos no programa Numbers, equivalente Apple para o Excel, onde o resultado pode ser plotado com gráficos de linha. Para gerar as curvas de condutância de alguns componentes, foi necessário criar gráficos de dispersão, e como o Numbers não permite a escolha de duas colunas para este, foi necessário abrir o csv em Excel e gerar este gráfico.

Para comparação, os mesmos circuitos foram rodados no programa mnae.exe fornecido pelo professor, em Windows, com os gráficos gerados no próprio.

A versão compilada para Windows também foi testada comparando os arquivos de saída, e notou-se que o comportamento é idêntico.

a) artefato.net

```
4
R0100 1 0 10
L0203 2 3 1E-3
N0301 3 1 -1000 0 0 0 0.1 1000 1 10000
E0400 4 0 2 3 1
V0200 2 0 SIN 0 10 1000 0 0 0 10
.TRAN 2E-3 10E-6 ADMO2 1 UIC
```

Figura 4.1: mnae.exe - tensões

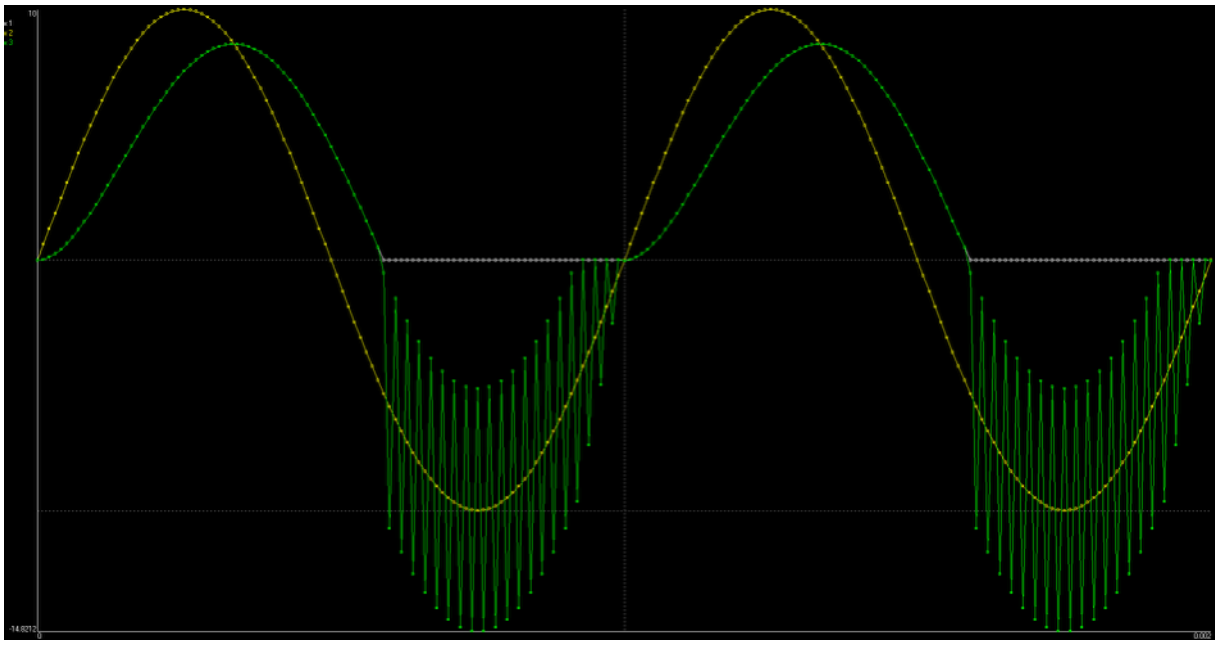
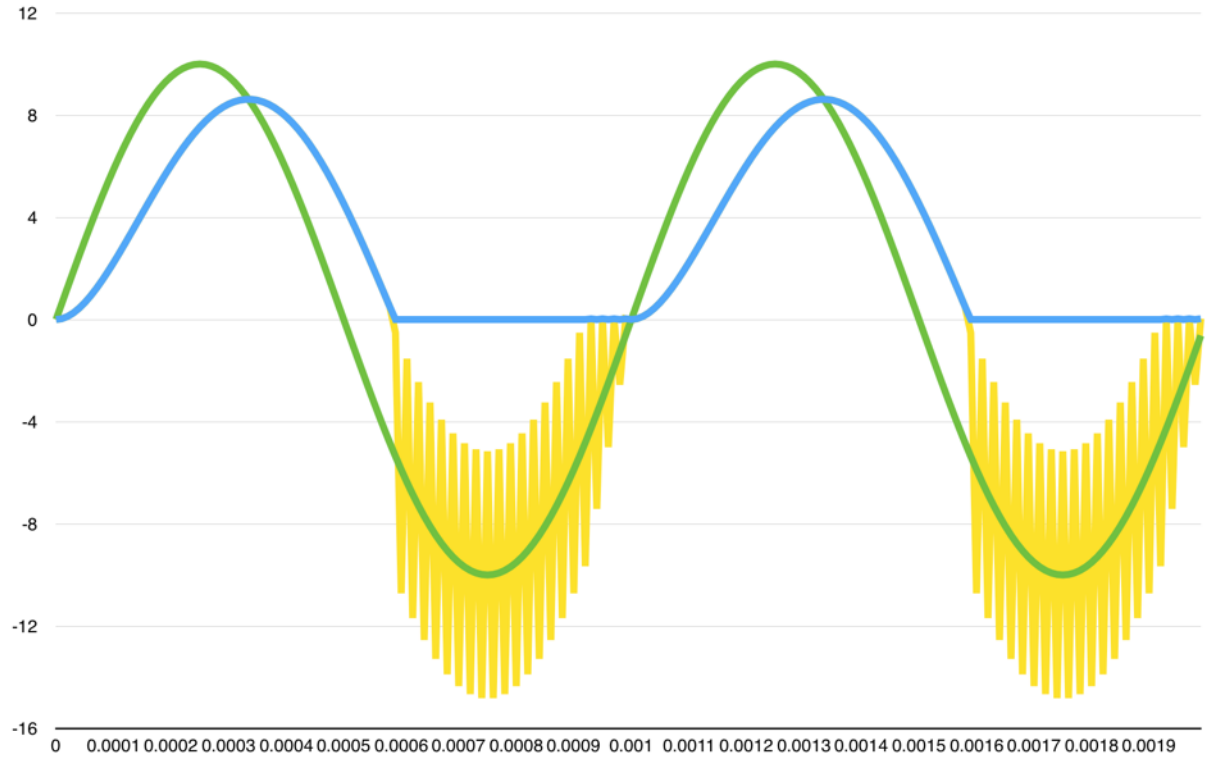


Figura 4.2: admo - tensões



b) chuaam.net

```
5
R0102 1 2 1.9
R0304 3 4 1.9
L0300 3 0 1
C0200 2 0 0.31 IC=0.1
C0100 1 0 1 IC=0.1
C0500 5 0 1
C0400 4 0 0.31 IC=0.1
C0300 3 0 1 IC=0.1
N0200 2 0 -2 1.1 -1 0.7 1 -0.7 2 -1.1
*H1 6 0 7 0 1
N0400 4 0 -2 1.1 -1 0.7 1 -0.7 2 -1.1
G0001 0 1 5 0 1
G0500 5 0 1 0 1
.TRAN 200 100E-3 ADMO3 1 UIC
```

Figura 4.3: mnae.exe - tensão no nó 1

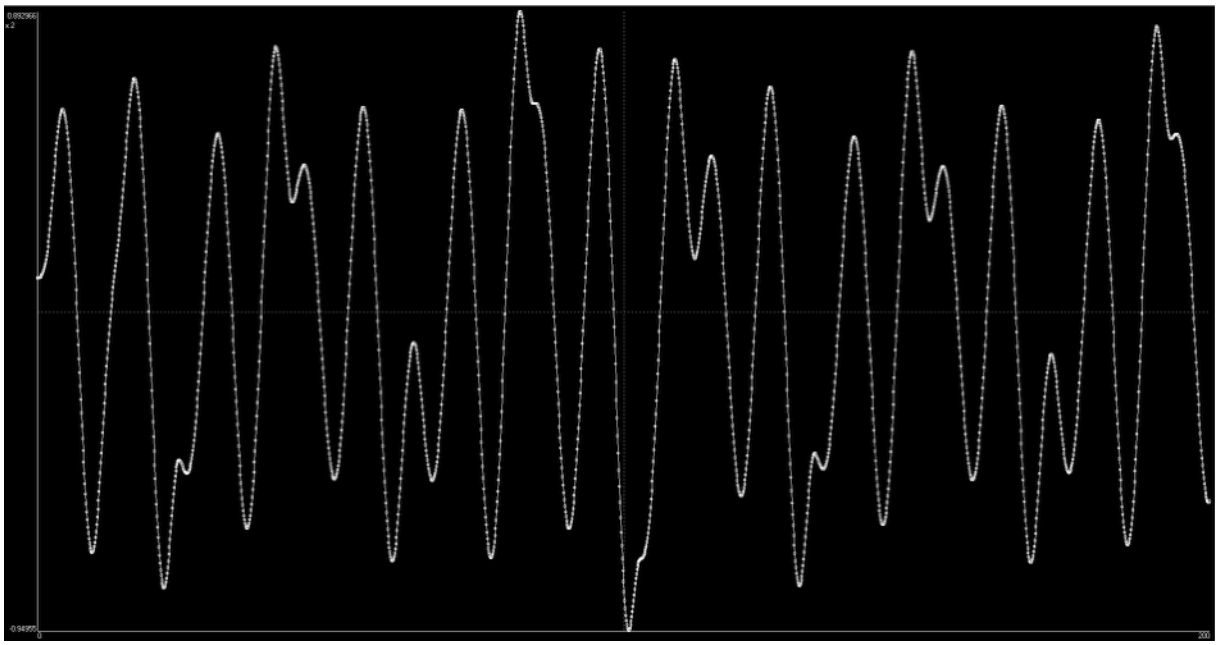


Figura 4.4: mnae.exe - comportamento caótico

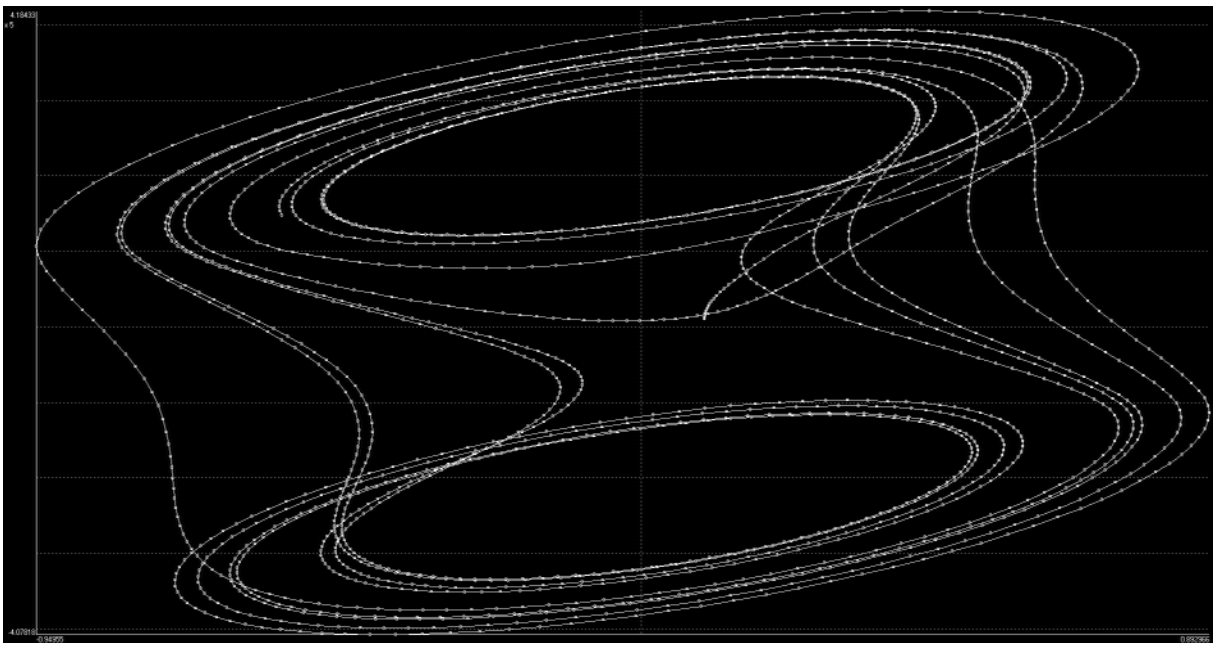


Figura 4.5: admo - tensão no nó 1

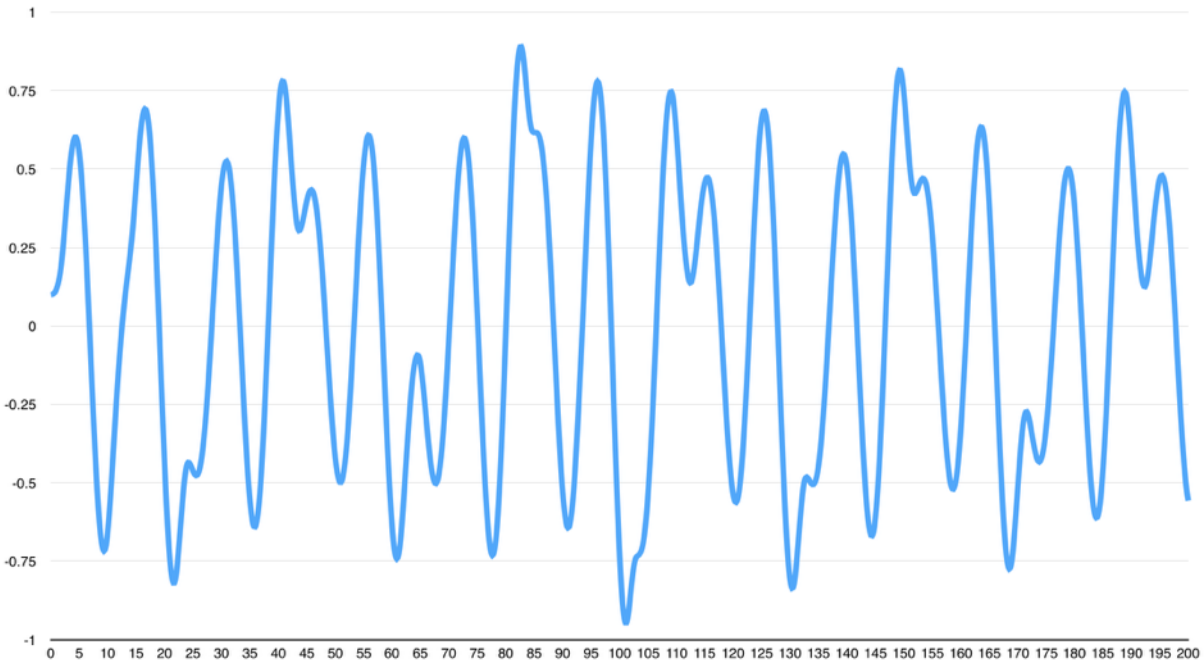


Figura 4.6: admo - comportamento caótico

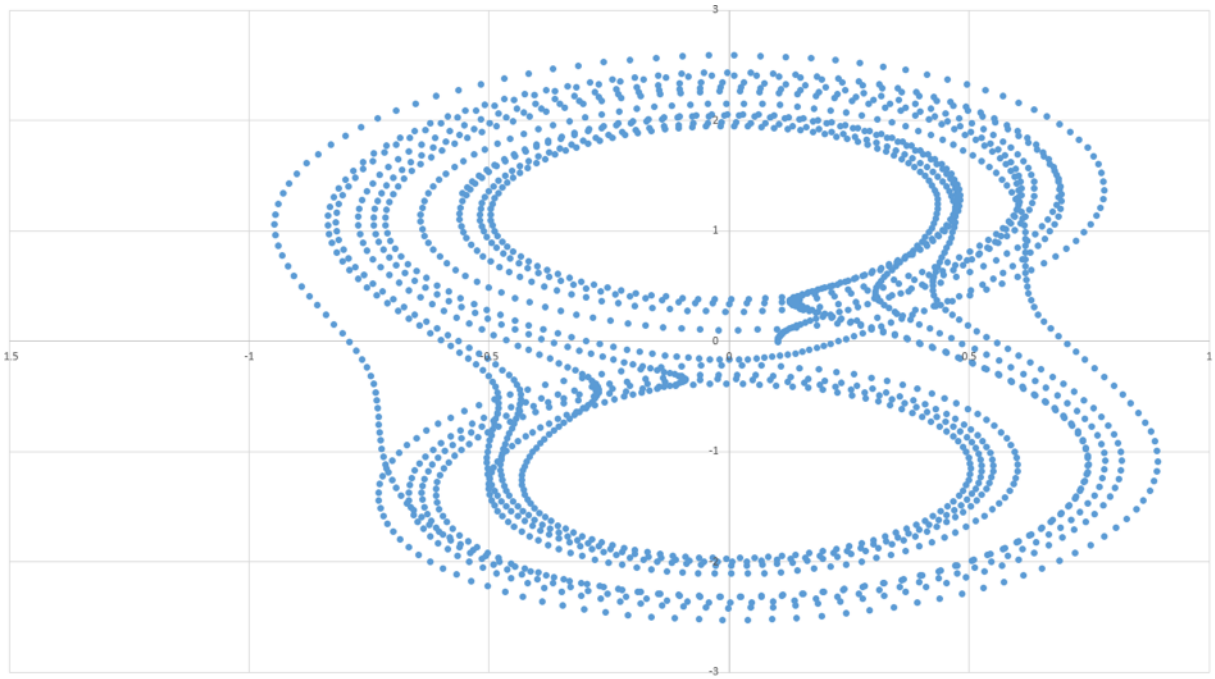
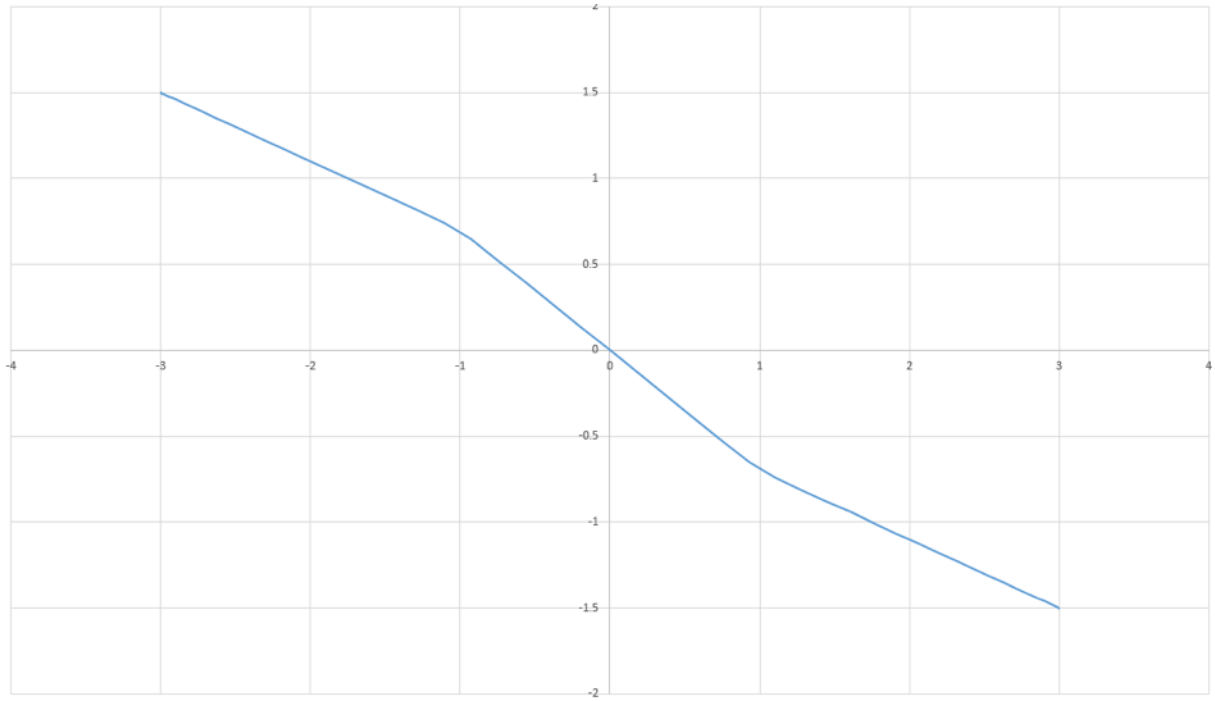


Figura 4.7: admo - diodo de chua



c) el5

```
4
R0100 1 0 1
R0203 2 3 1
L0402 4 2 0.586082142617872940
L0104 1 4 0.881627694904598500
C0200 2 0 1.415174707426466310
C0400 4 0 2.130674656969229800
C0100 1 0 1.844218243238877410
C0402 4 2 1.085374806121910160
C0104 1 4 0.364397675632797990
V0300 3 0 PULSE 0 1 0 1 1 49 100 5
.TRAN 15 10000E-5 ADMO3 1 UIC
```

Figura 4.8: mnae.exe - tensões nos nós

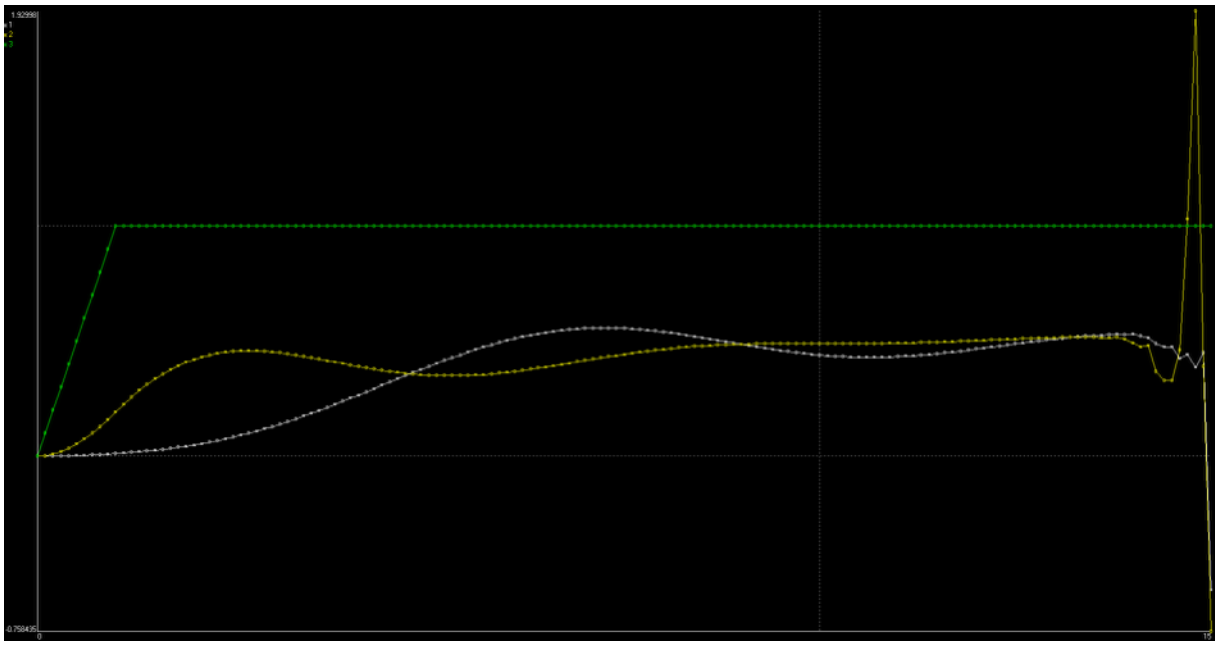
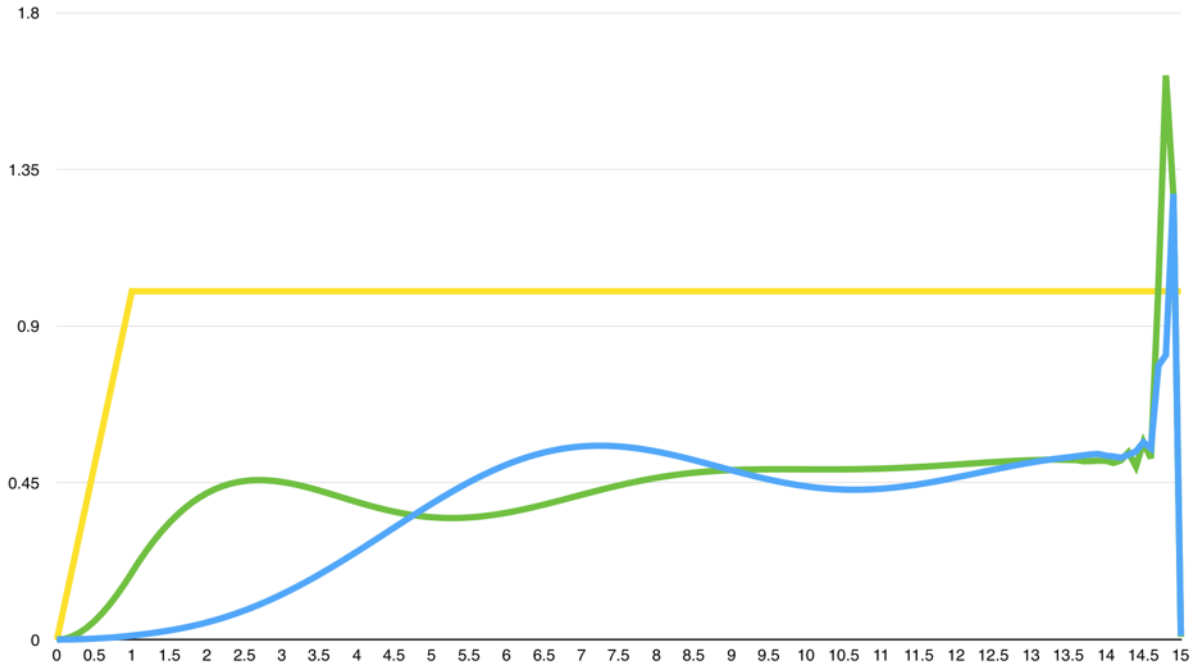


Figura 4.9: admo - tensões nos nós



d) lc.net

```
6
L0100 1 0 1E-3 IC=0
L0200 2 0 0.25E-3
L0300 3 0 0.1111111111E-3
C0100 1 0 1E-6 IC=1
C0200 2 0 1E-6 IC=1
C0300 3 0 1E-6 IC=1
E0400 4 0 3 0 1
E0504 5 4 2 0 1
E0605 6 5 1 0 1
.TRAN 0.003 0.0003E-3 ADMO2 1 UIC
```

Figura 4.10: mnae.exe - tensões

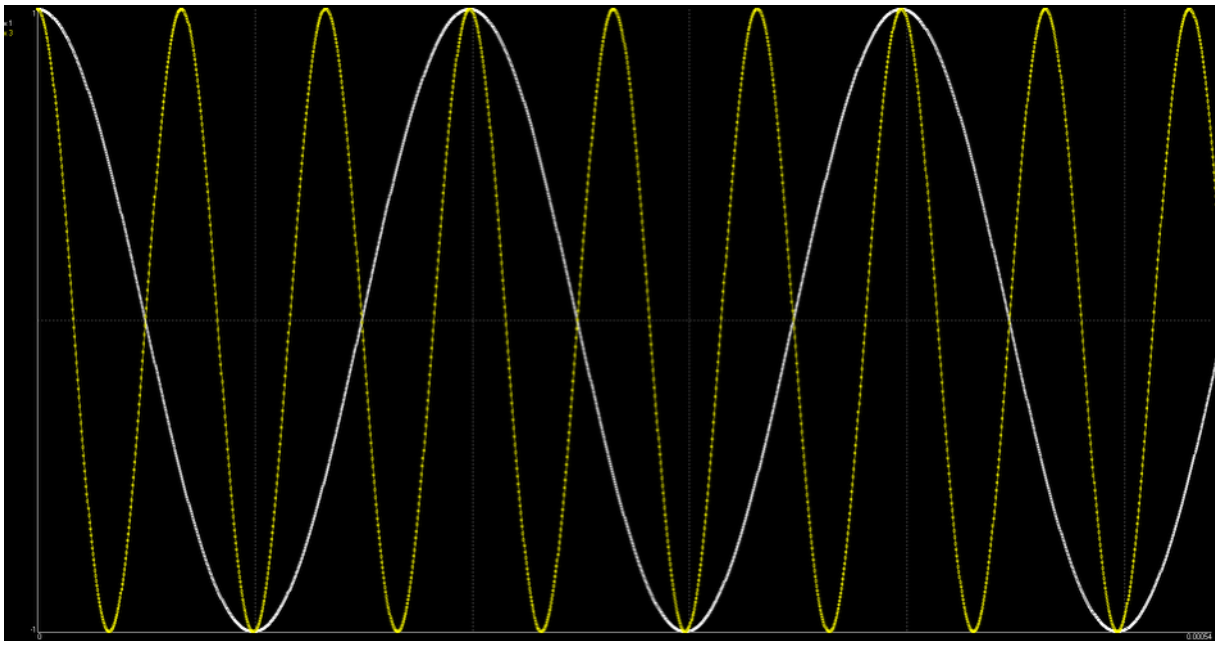
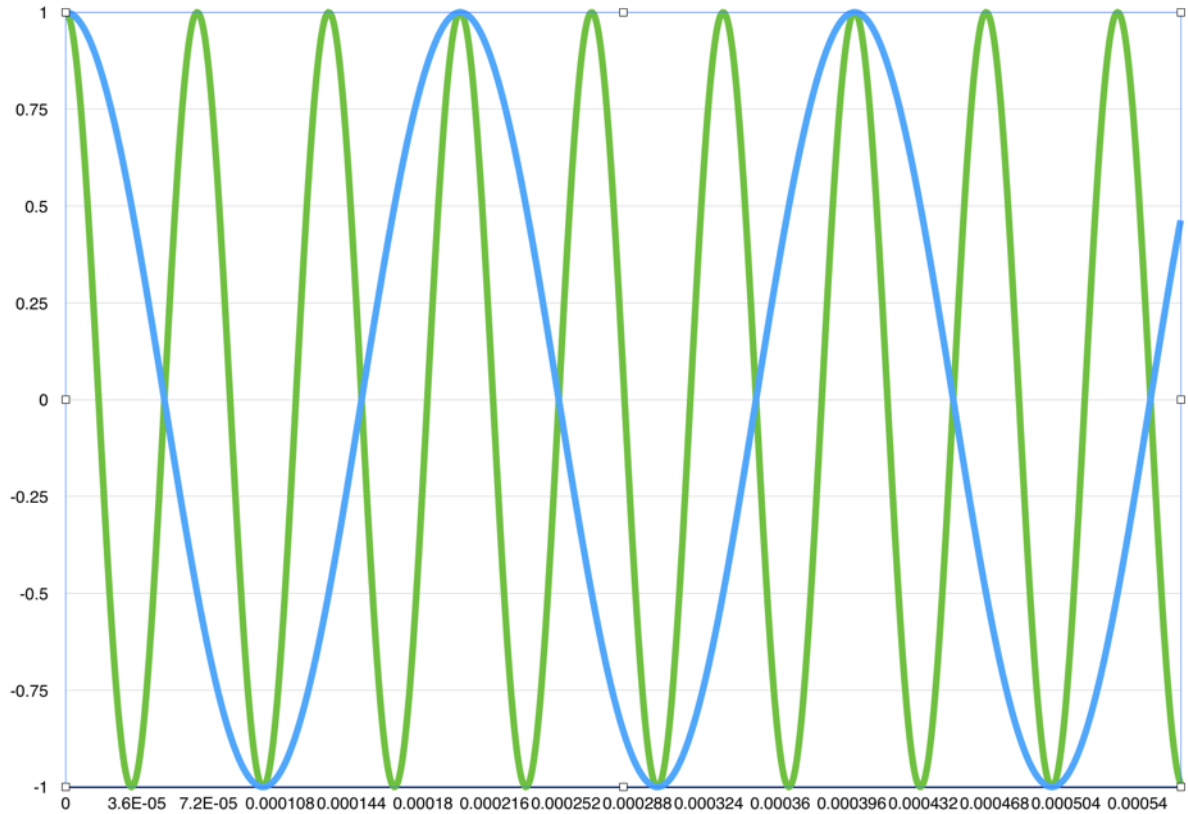


Figura 4.11: admo - tensões



Observação: aqui, o programa demorou 202 segundos para fazer os cálculos, enquanto o modelo fornecido pelo professor demorou apenas 1 segundo. Uma clara demonstração da ineficiência do método de Cramer.

e) mres.net

```
4
L0100 1 0 5.14532930107527E-2
L0201 2 1 5.15539843865076E-2
L0302 3 2 7.84557907845579E-2
L0403 4 3 1.81818181818182E-1
C0100 1 0 7.06040816326531 IC=1
C0200 2 0 3.38349206349206
C0300 3 0 1.92063492063492
C0400 4 0 1
.TRAN 40 8E-2 ADMO3 1 UIC
```

Figura 4.12: mnae.exe - tensão no nó 1 com ordem 1

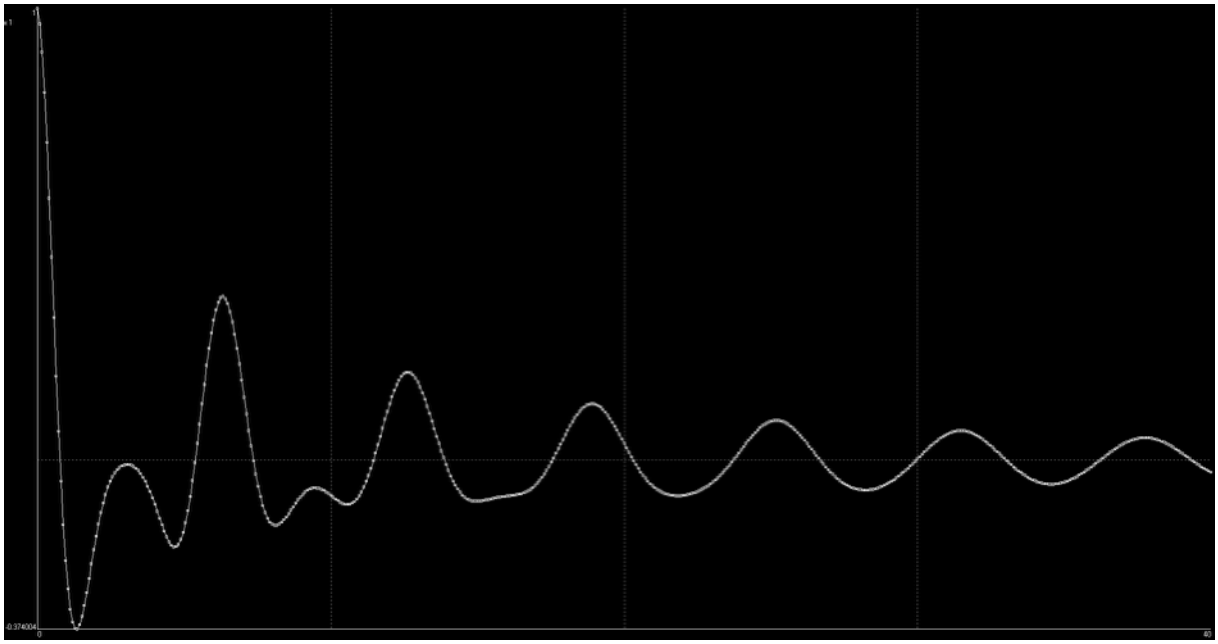


Figura 4.13: mnae.exe - tensão no nó 1 com ordem 2

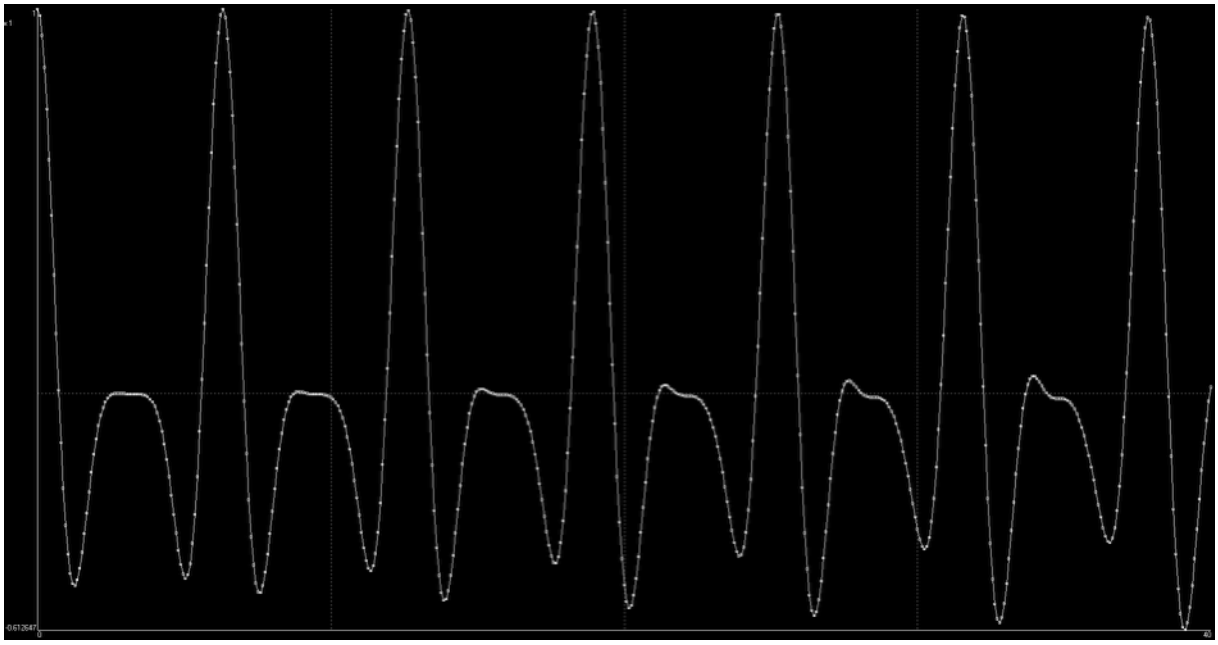


Figura 4.14: mnae.exe - tensão no nó 1 com ordem 3

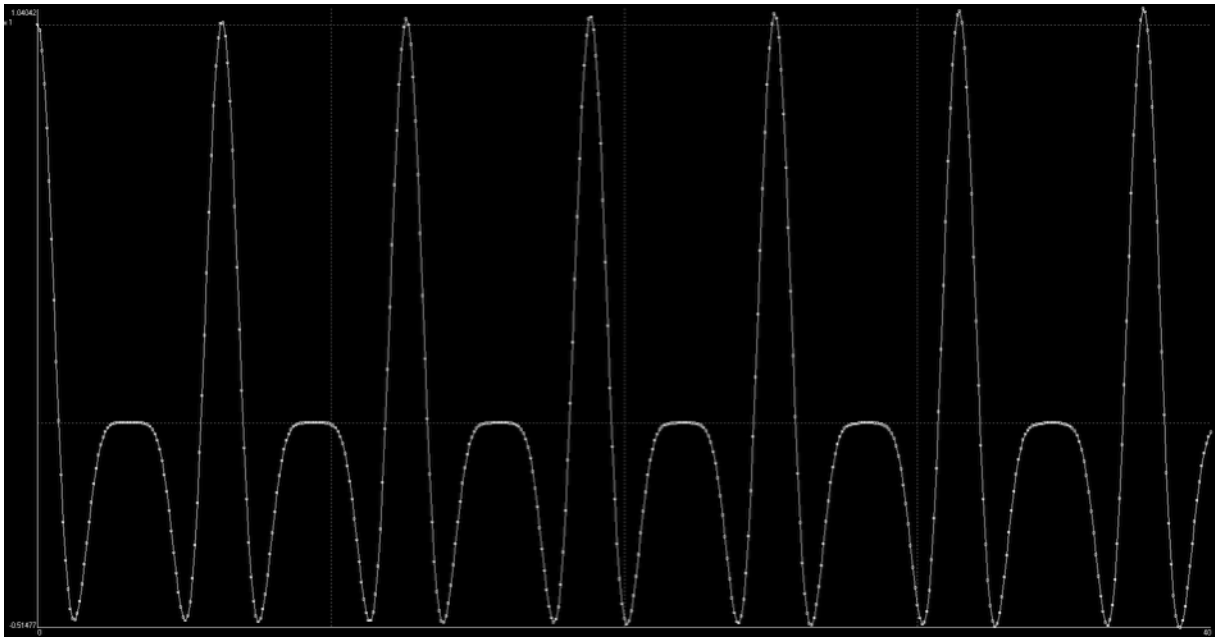


Figura 4.15: mnae.exe - tensão no nó 1 com ordem 4

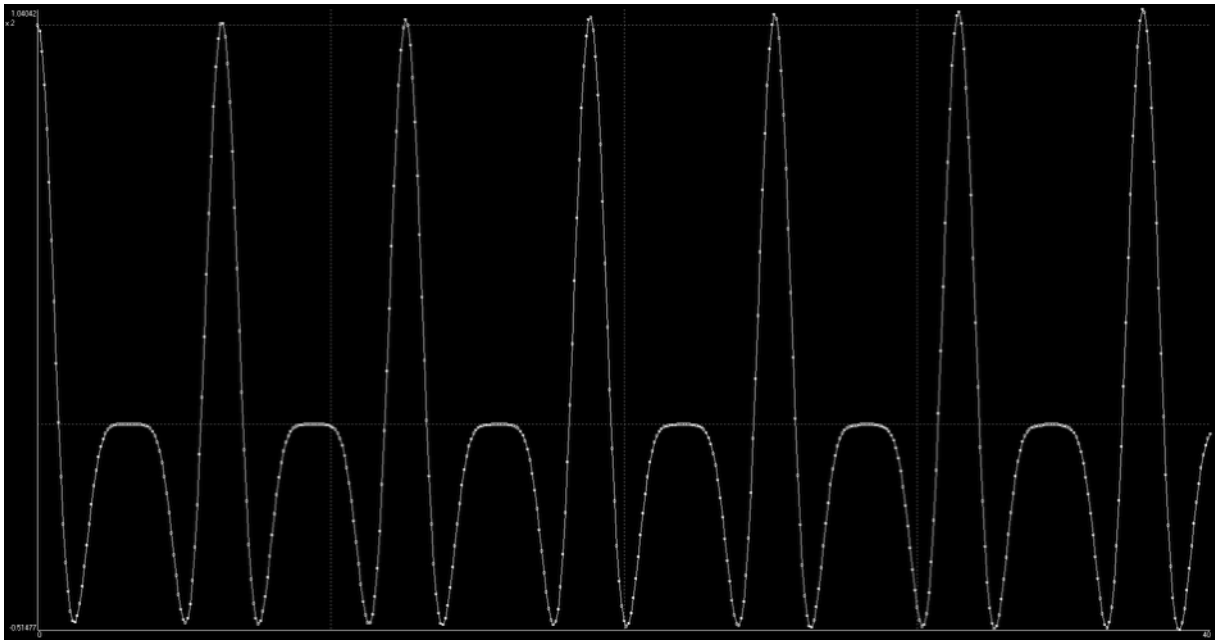


Figura 4.16: admo - tensão no nó 1 com ordem 1

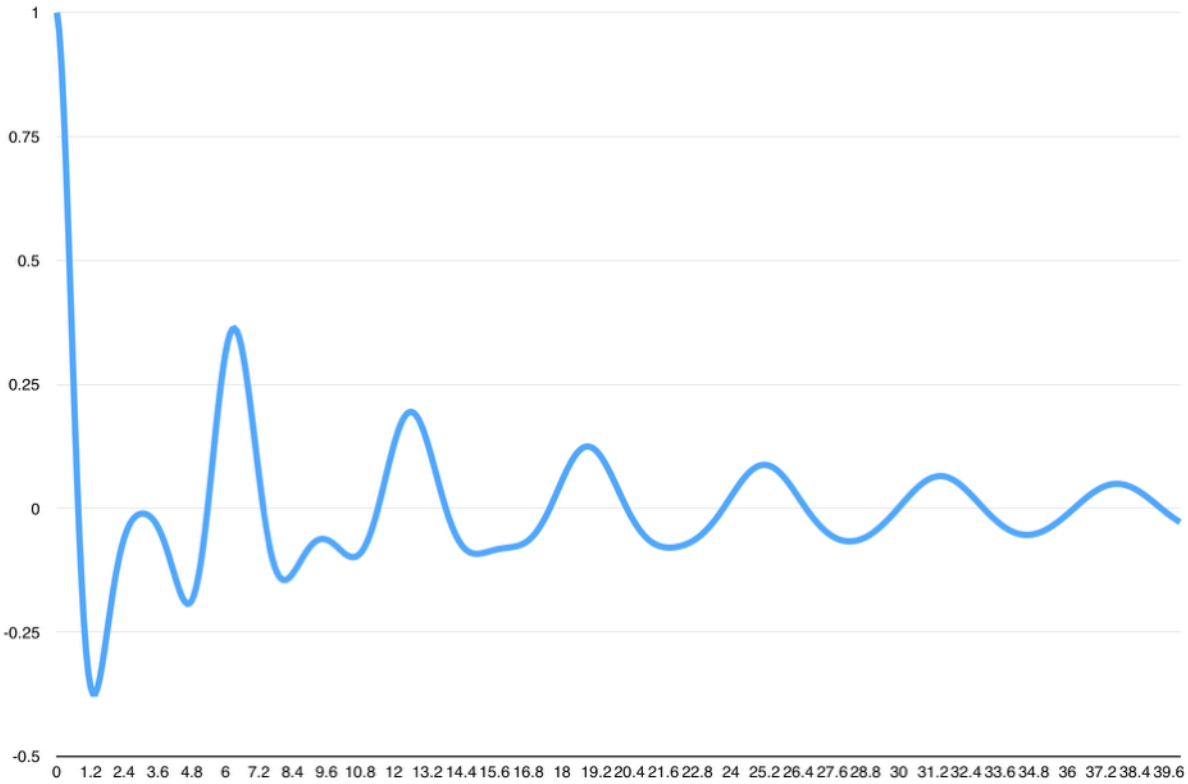


Figura 4.17: admo - tensão no nó 1 com ordem 2

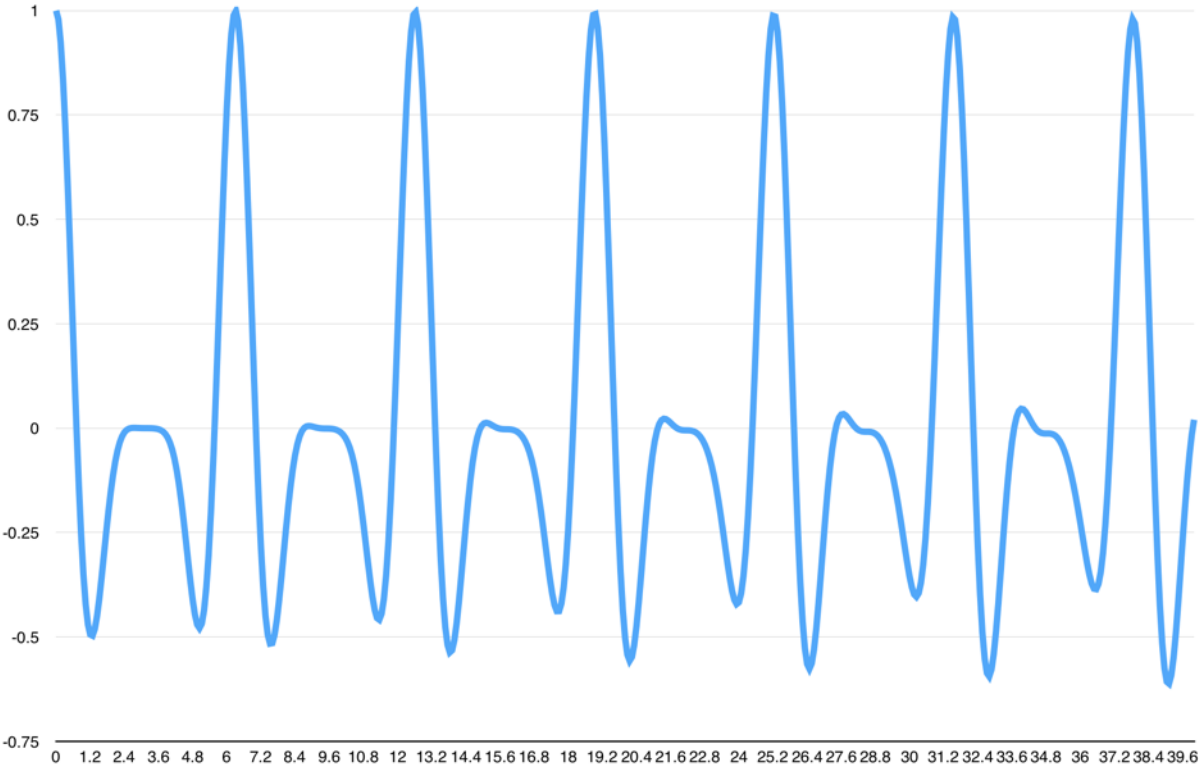


Figura 4.18: admo - tensão no nó 1 com ordem 3

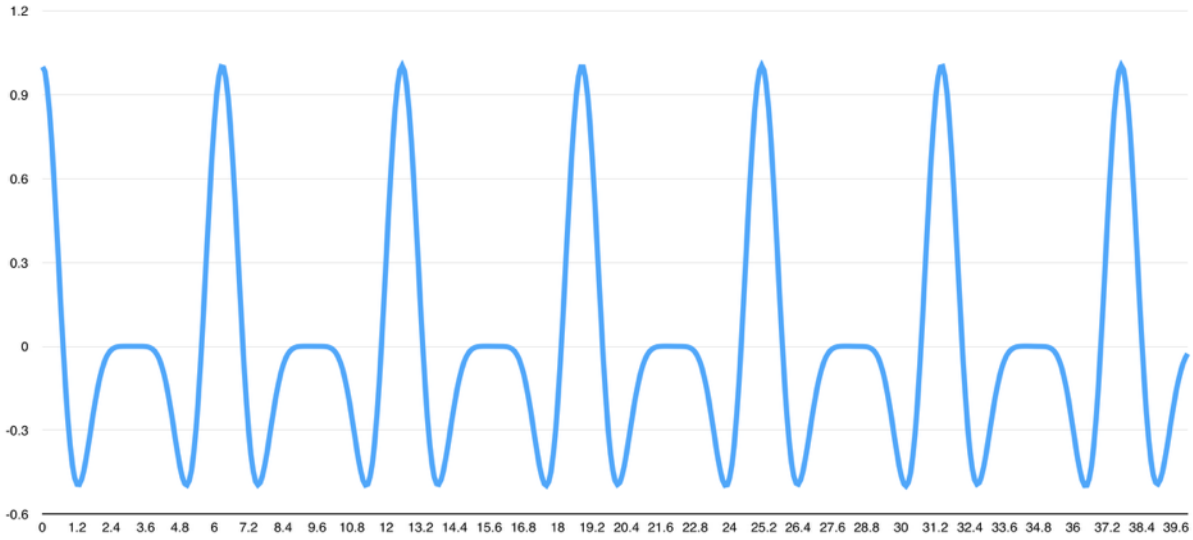
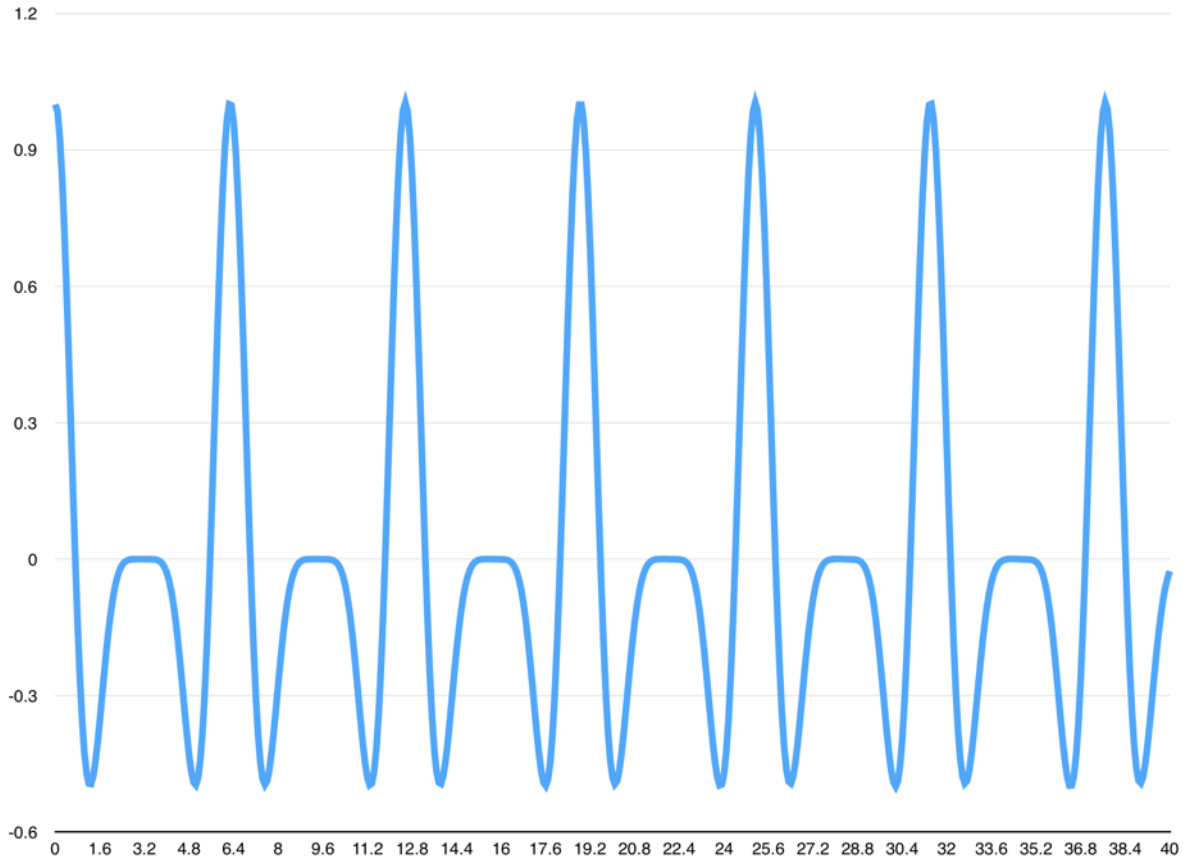


Figura 4.19: admo - tensão no nó 1 com ordem 4



f) neon.net

```
6
R0102 1 2 20000
R0304 3 4 20000
C0100 1 0 1E-6
C0300 3 0 1.1E-6
N0500 5 0 0 0 0.001 50 0.01 10 0.02 20
N0600 6 0 0 0 0.001 50 0.01 10 0.02 20
G0100 1 0 5 0 1
G0500 5 0 0 1 1
G0300 3 0 6 0 1
G0600 6 0 0 3 1
V0200 2 0 PULSE 0 100 0 0.01 0.01 1000 2000 1
V0400 4 0 PULSE 0 100 0 0.01 0.01 1000 2000 1
.TRAN 0.1 0.1E-3 ADMO2 1 UIC
```

Figura 4.20: mnae.exe - tensão nos nós

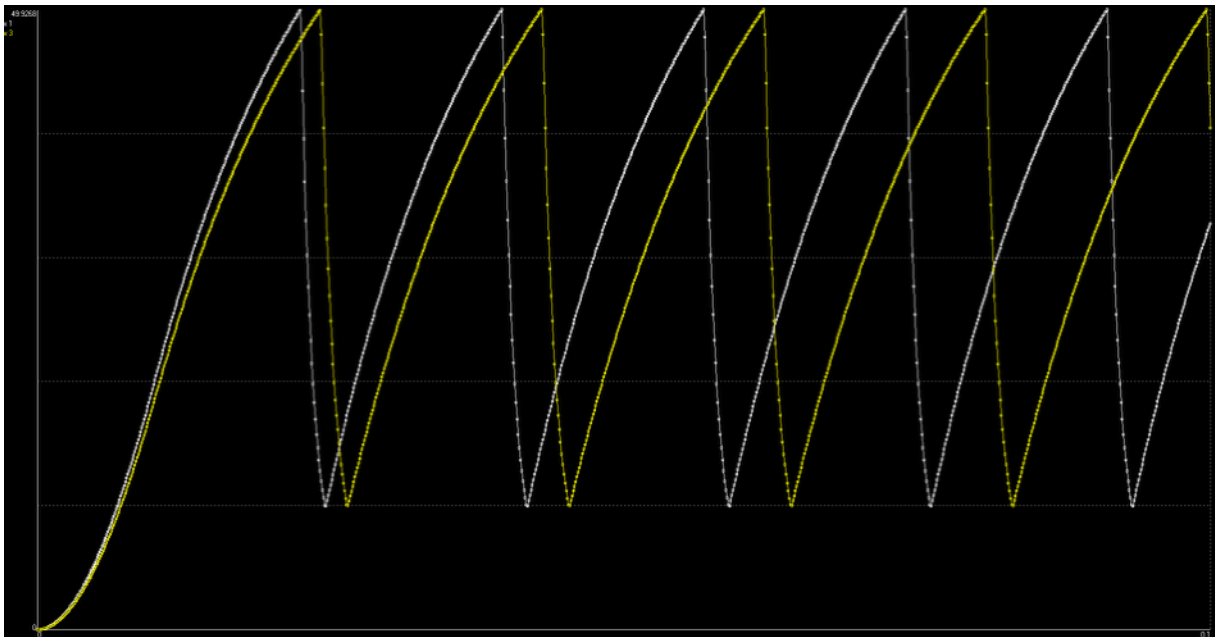
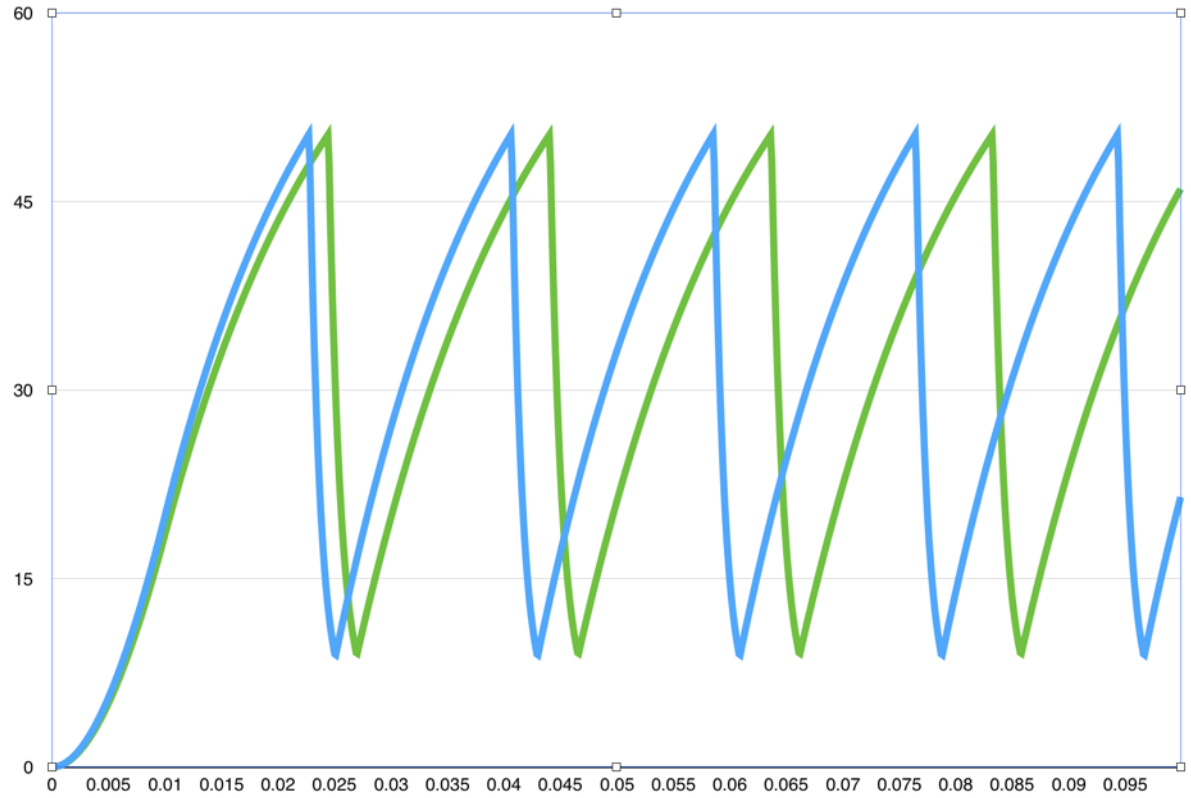


Figura 4.21: admo - tensão nos nós



g) P1-1.net (questão 1 da P1)

```
questao 1
C1 1 0 2
F1 2 1 2 3 2
C2 2 0 1 IC=10
R1 3 0 1
.TRAN 10 0.01 ADMO2 1 UIC
```

Figura 4.22: mnae.exe - tensão no nó 1

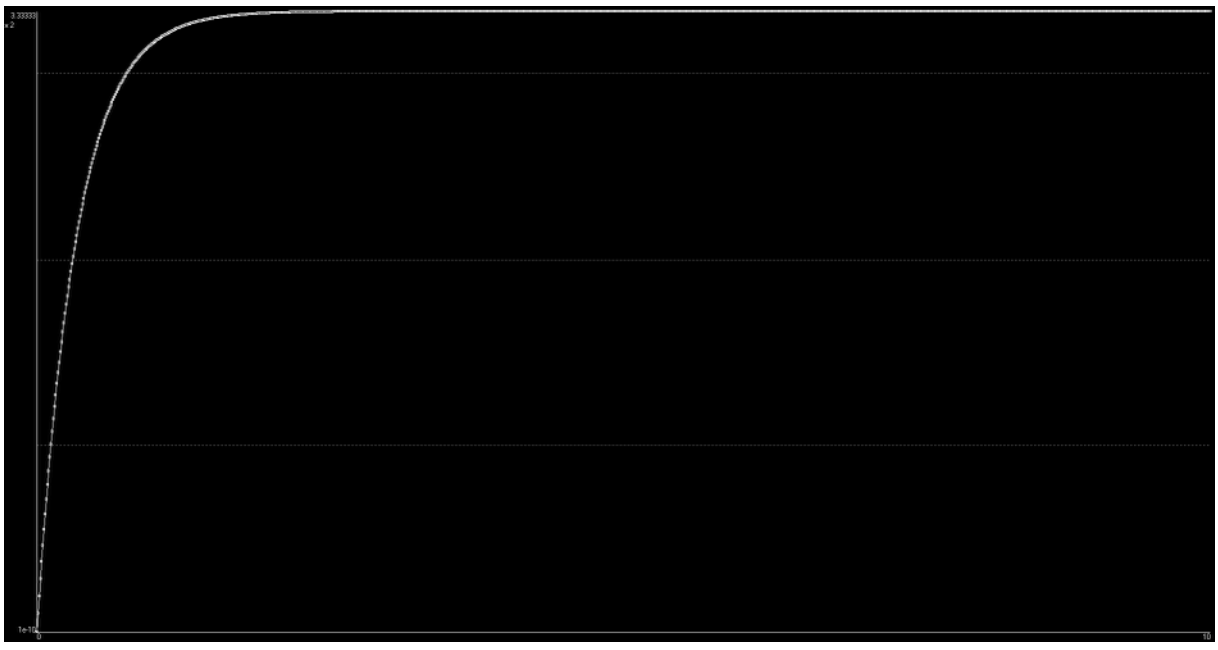
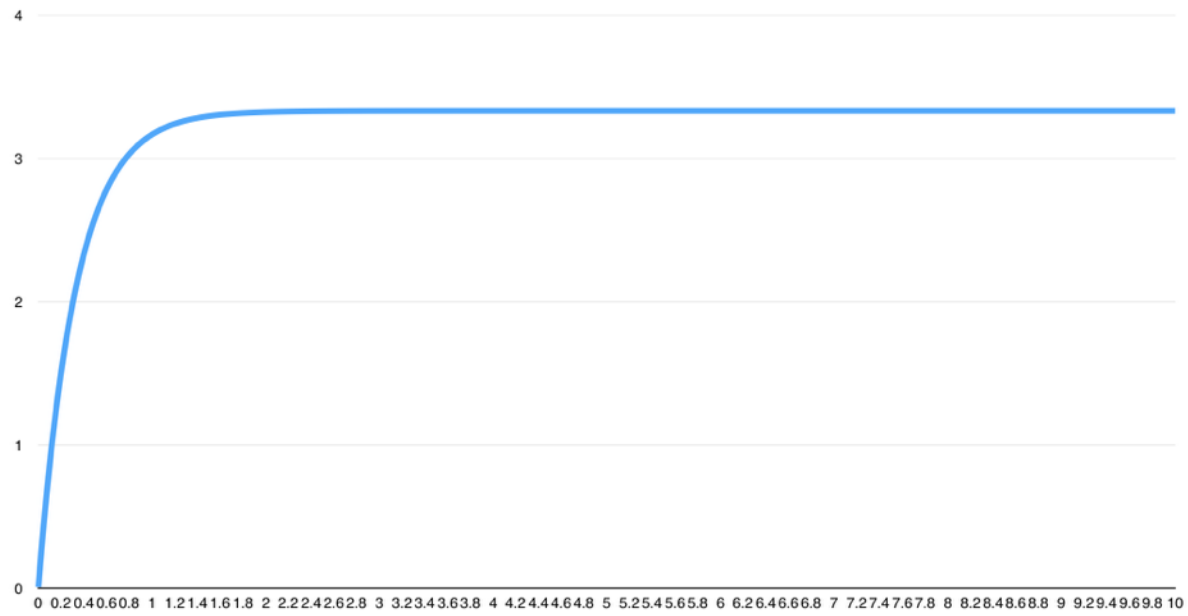


Figura 4.23: admo - tensão no nó 1



h) pag51.net (circuito da página 51, um teste de fontes controladas)

```
5
I1 0 1 DC 1
R1 2 0 5
R2 3 0 4
R3 4 3 2
R4 5 2 1
V1 3 1 DC 2
G1 2 4 3 0 4
E1 4 0 2 4 2
F1 3 0 1 5 6
.TRAN 10 1 ADMO1 1 UIC
```

Figura 4.24: mnae.exe - tensões

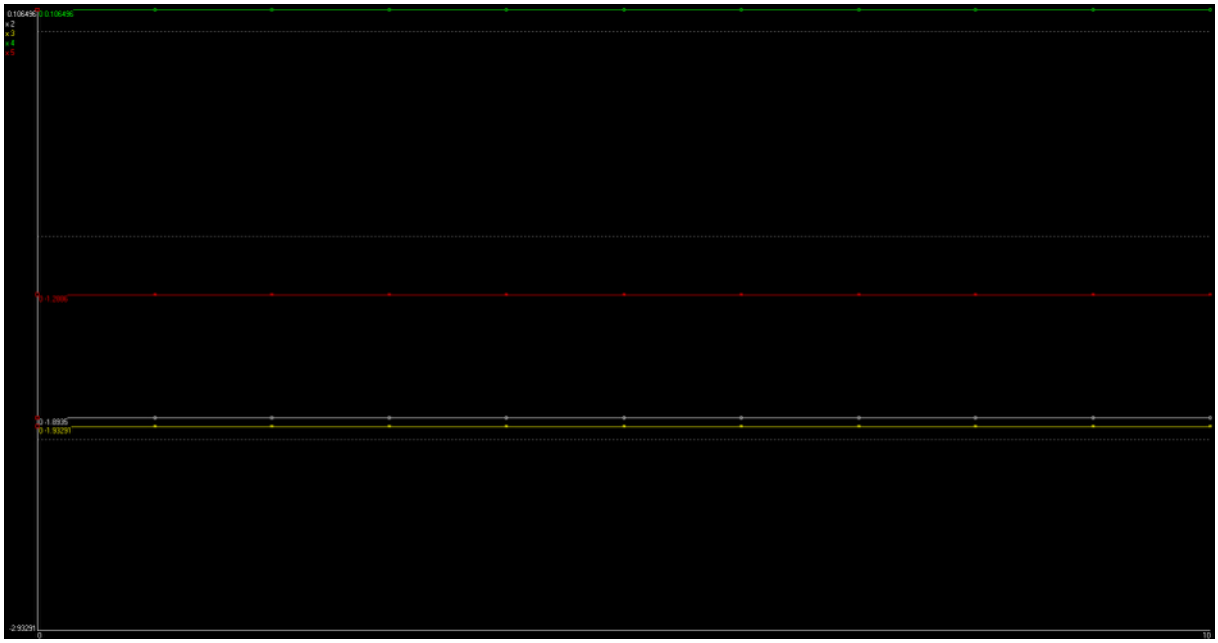
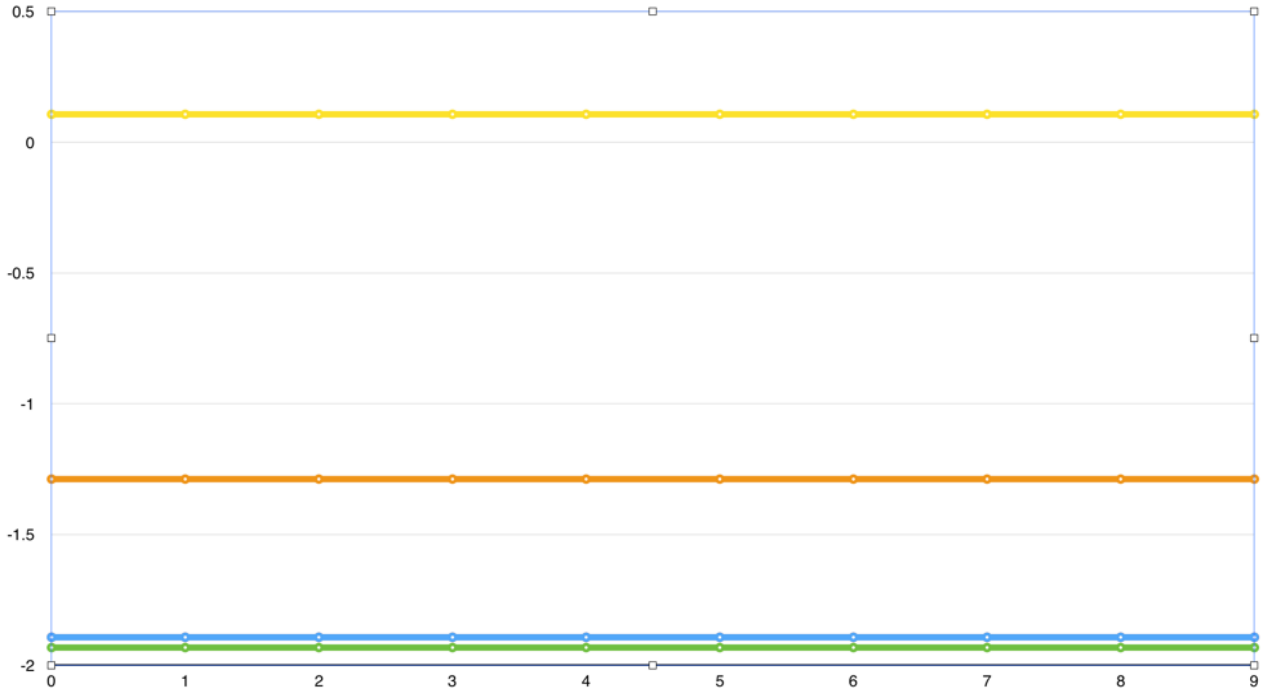


Figura 4.25 : admo - tensões



Edit Data References

Table 1

t	e0	e1	e2	e3	e4	e5	jV1	jE1	jF1ctrl
0	0	-1.8935	-1.93291	0.106496	-1.2886	-1.8935	-0.960596	1.12354	0.0394036

i) pulse.net (simples implementação de uma fonte PULSE)

```
comentario
IP 0 1 PULSE 1 5 0.4 0.1 0.1 0.4 1 4
R1 1 0 1
.TRAN 5 0.01 ADMO1 1 UIC
```

Figura 4.26: mnae.exe

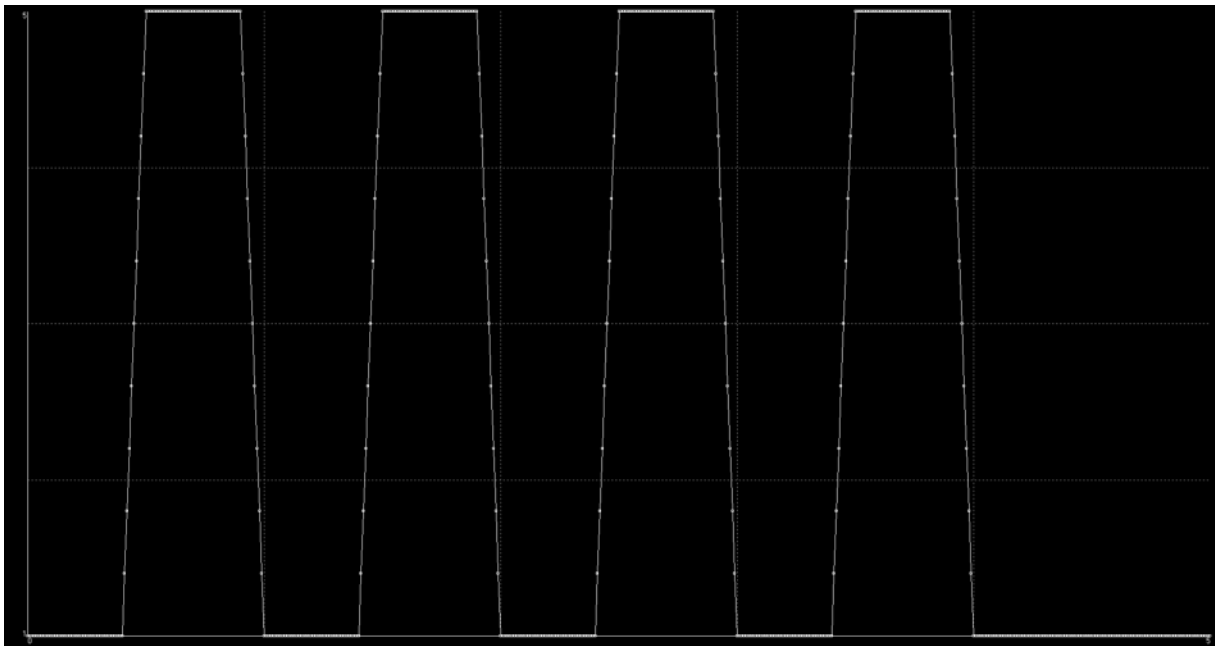
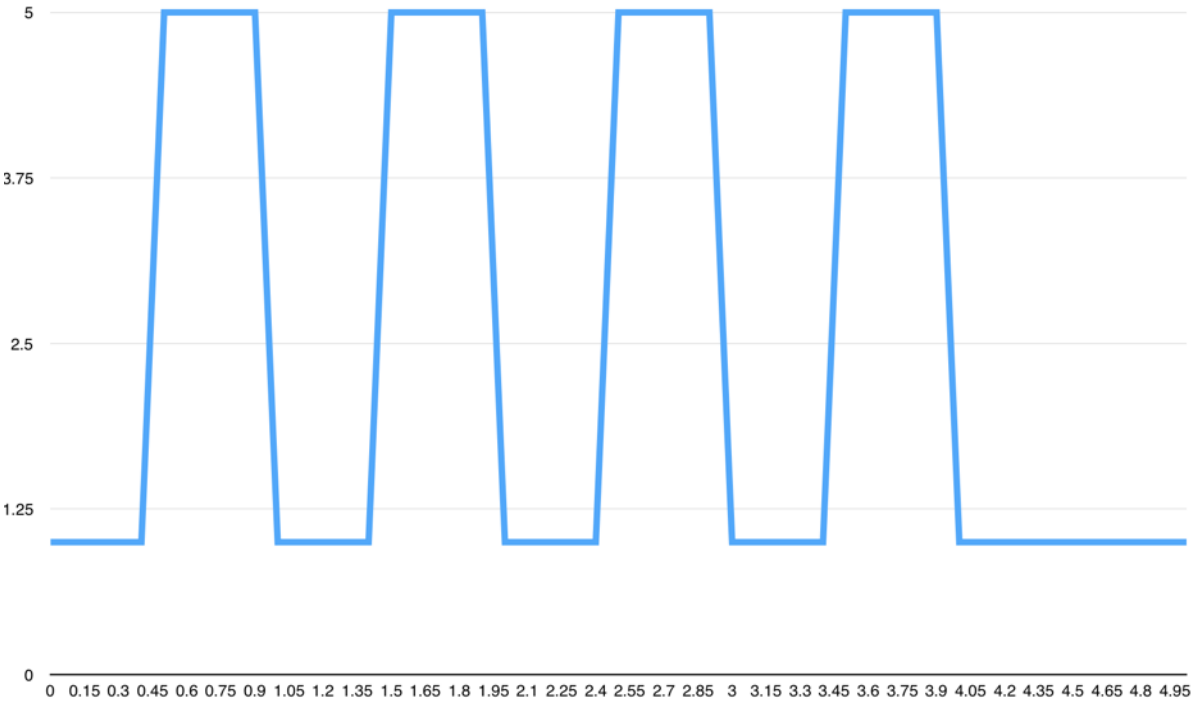


Figura 4.27: admo



j) sc.net

```
6
C0100 1 0 1E-9
C0203 2 3 1E-9
$0104 1 4 5 0 5E-4 0 2.5
$0301 3 1 6 0 5E-4 0 2.5
O0200 2 0 3 0
V0500 5 0 PULSE 0 5 0 0.01E-3 0.01E-3 0.03E-3 0.1E-3 10000
V0600 6 0 PULSE 0 5 0.05E-3 0.01E-3 0.01E-3 0.03E-3 0.1E-3 10000
V0400 4 0 SIN 0 1 1000 0 0 0 1000
.TRAN 0.0006 5E-6 ADMO3 1 UIC
```

Figura 4.28: mnae.exe

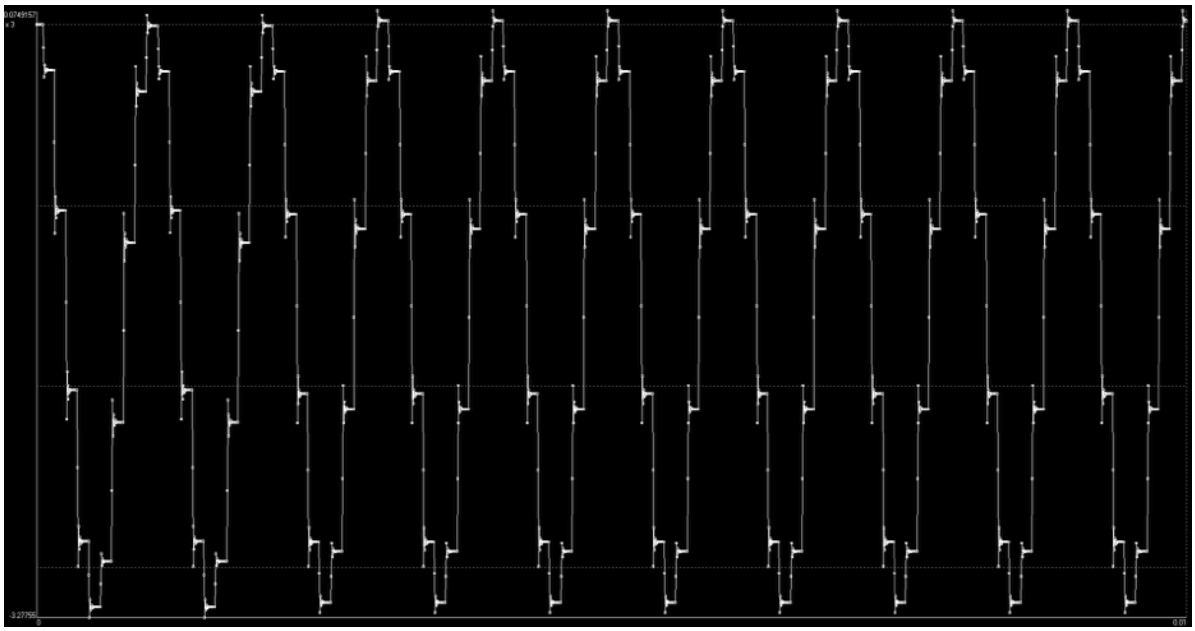
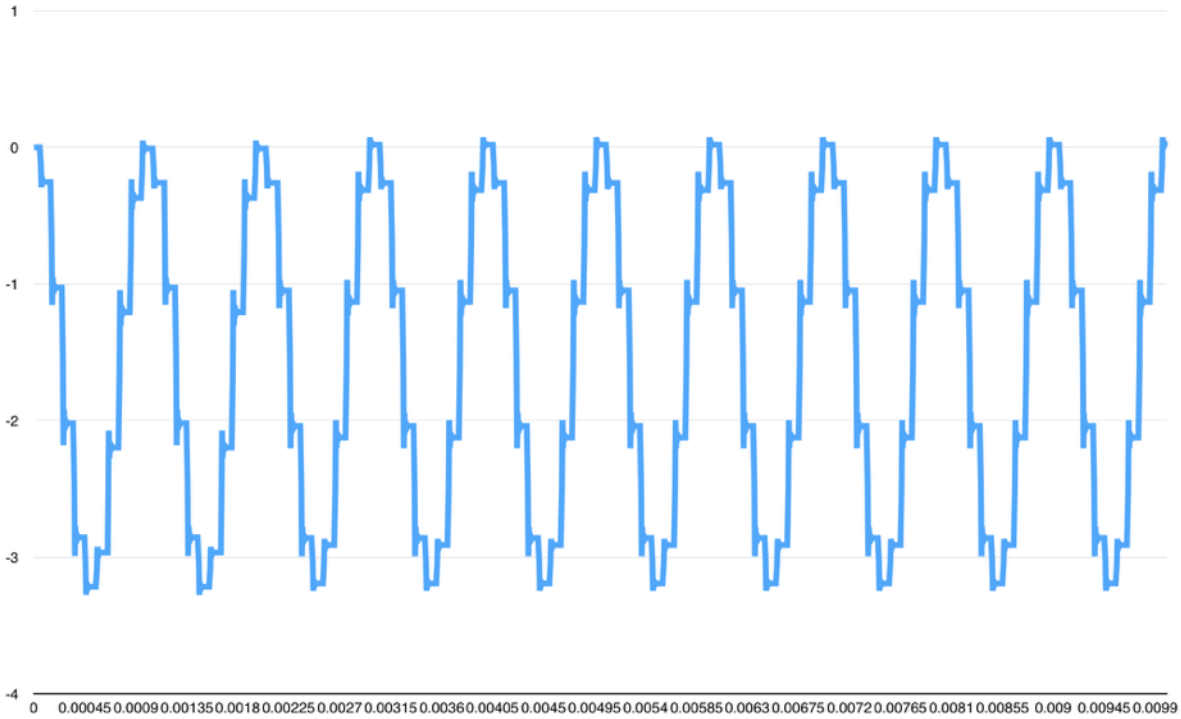


Figura 4.29: admo



k) simples.net

```
6
R0102 1 2 1
R0301 3 1 1
R0403 4 3 1
R0004 0 4 1
R0502 5 2 1
R0605 6 5 1
O0300 3 0 1 0
O0600 6 0 5 4
V0200 2 0 DC 1
.TRAN 40 8e-2 ADMO3 1 UIC
```

Figura 4.30: mnae.exe - tensões

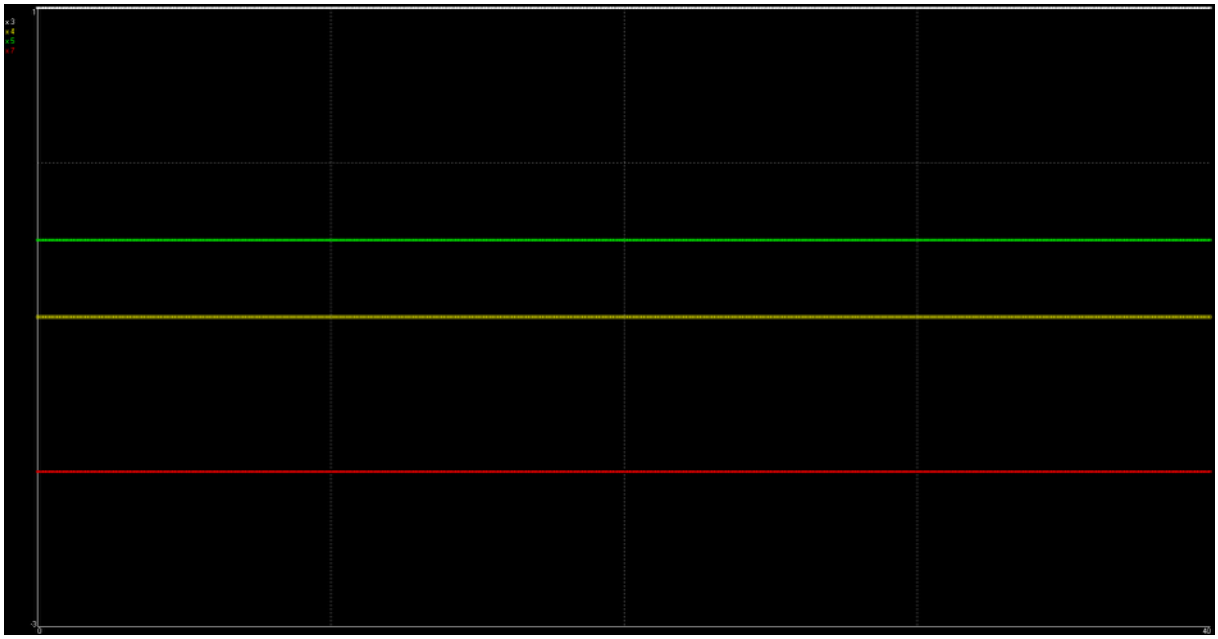


Figura 4.31: admo - tensões

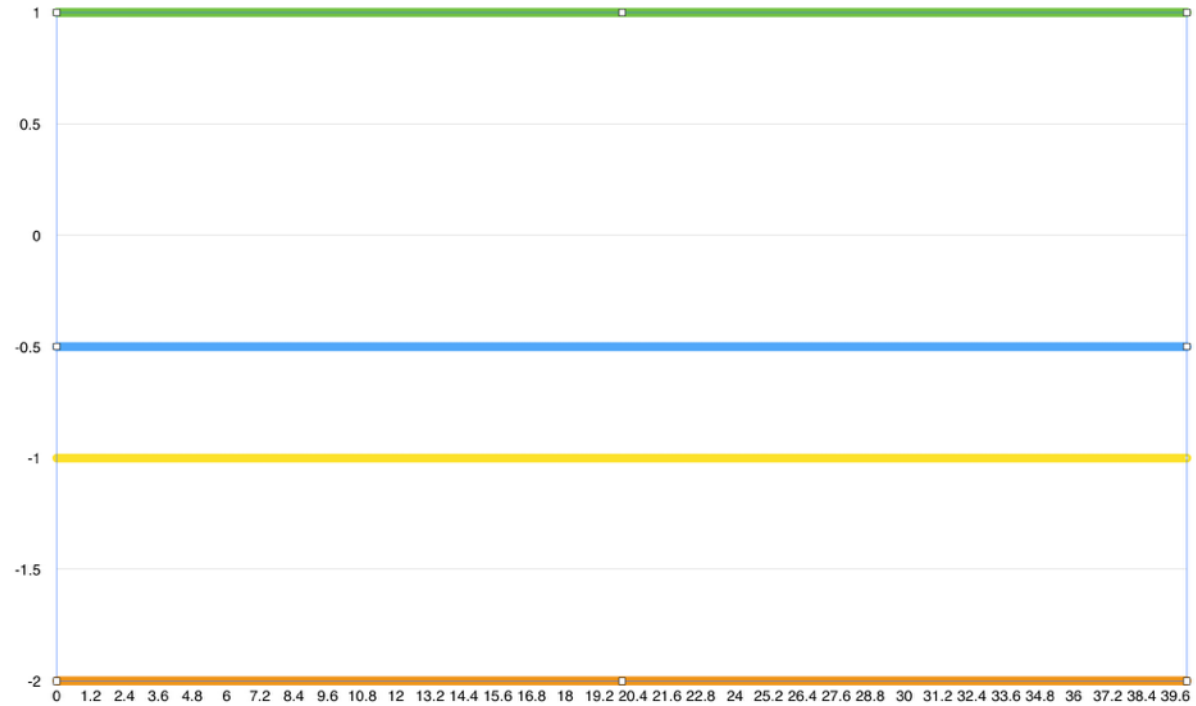


Table 1

t	e0	e1	e2	e3	e4	e5	e6	jV0200
0	0	0	1	-1	-0.5	-0.5	-2	-2.5

Edit Data References

l) nic.net (conversor negativo de impedância)

```
comentario
VSIN 1 0 SIN 1 5 1 0.2 0.5 45 4
R1 3 0 1
R2 2 3 1
R3 1 2 2
OPAMP 2 0 1 3
.TRAN 5 0.01 ADMO1 1 UIC
```

Figura 4.32: mnae.exe - fonte SIN utilizada

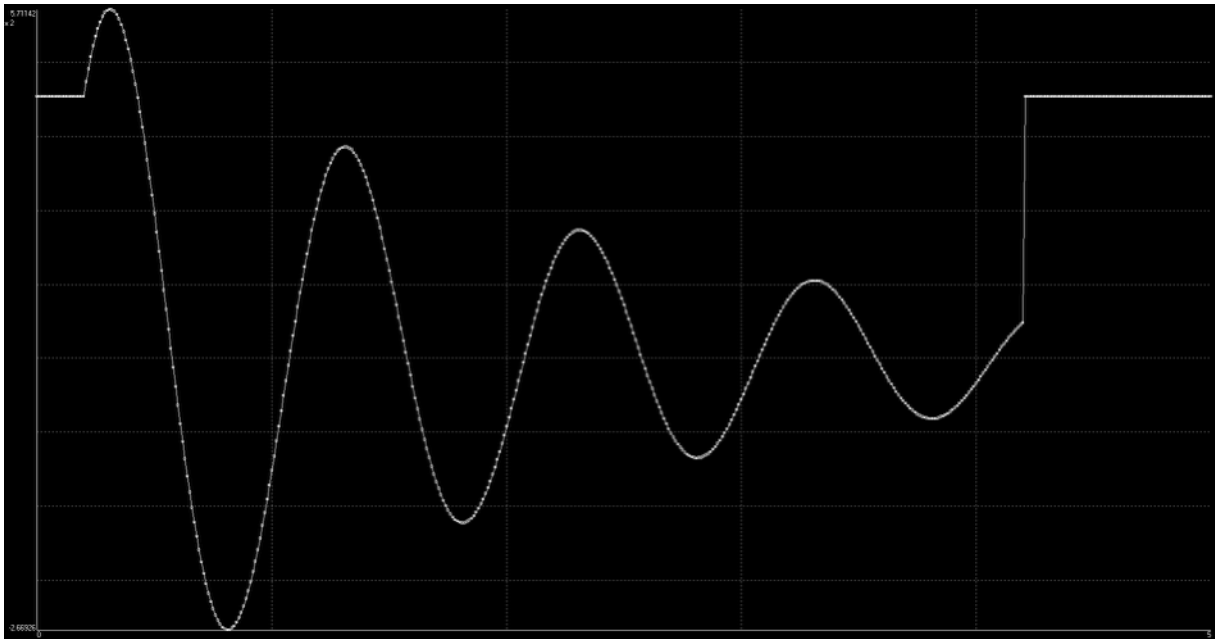


Figura 4.33: admo - fonte SIN utilizada

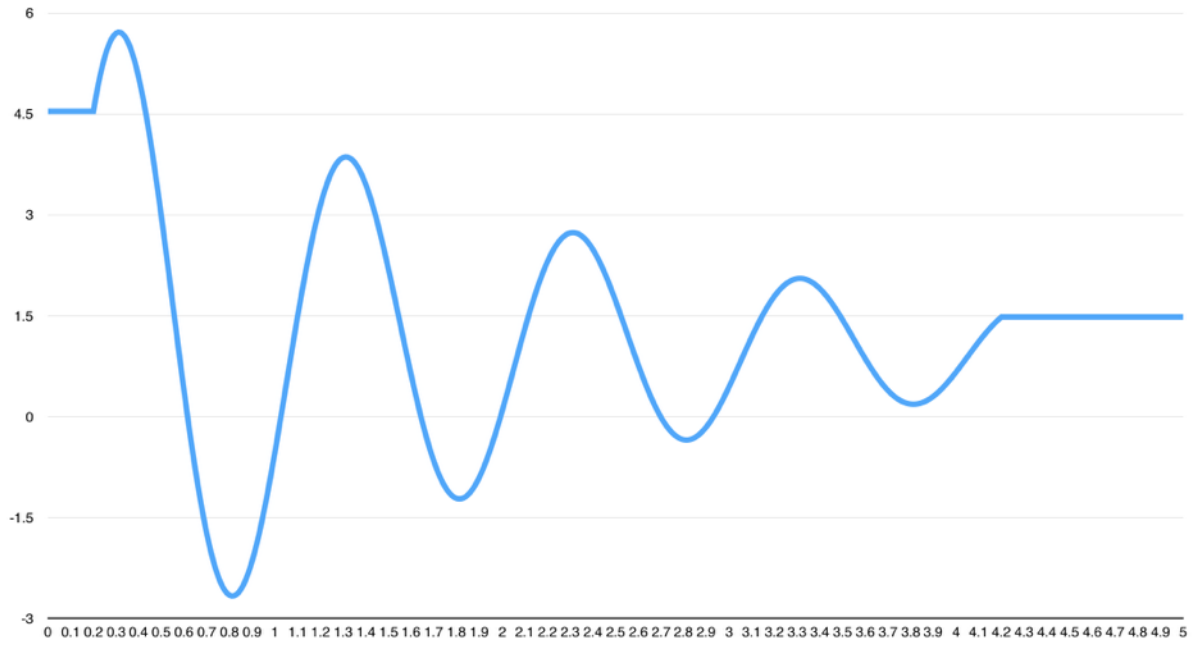
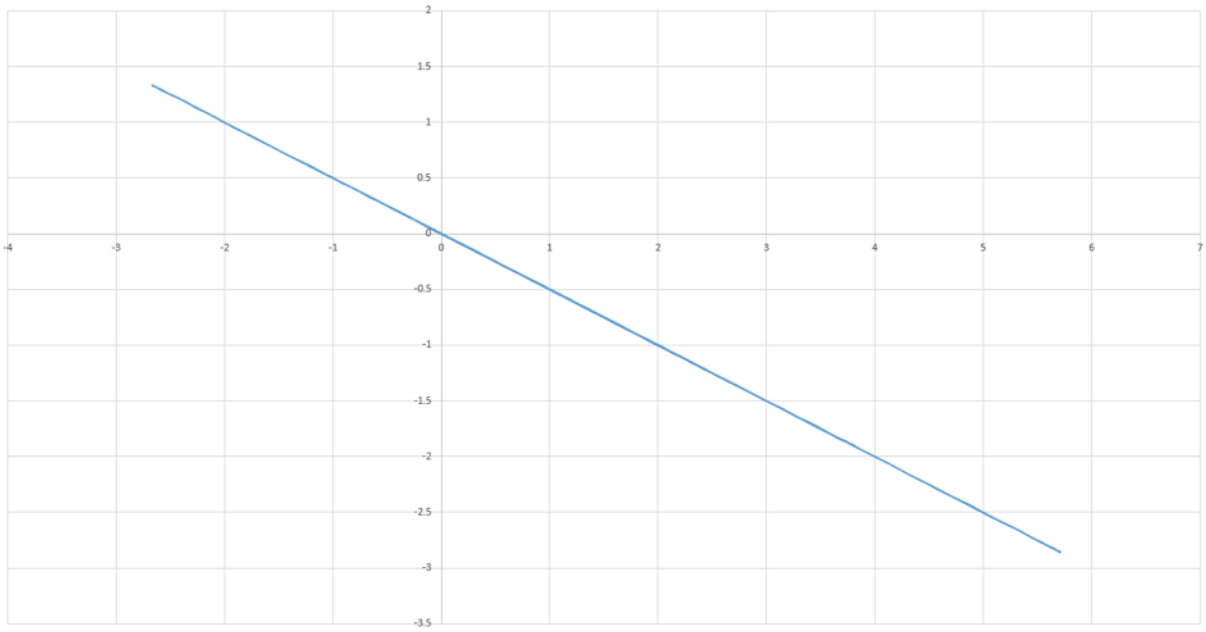


Figura 4.34: admo - curva do resistor



Observações: Escolheu-se adotar o valor final para a fonte após um número limitado de ciclos para não criar descontinuidades, como especificado. O programa exemplo, porém, não se comportava desta maneira.

m) tesla.net

```
3
L0100 1 0 1.0497E-4
L0102 1 2 -4.96999999999999E-6
L0301 3 1 9.89503E-3
C0200 2 0 10E-9 IC=10000
C0300 3 0 100E-12
.TRAN 1E-4 1E-7 ADMO2 1 UIC
```

Figura 4.35: mnae.exe - tensão no nó 1

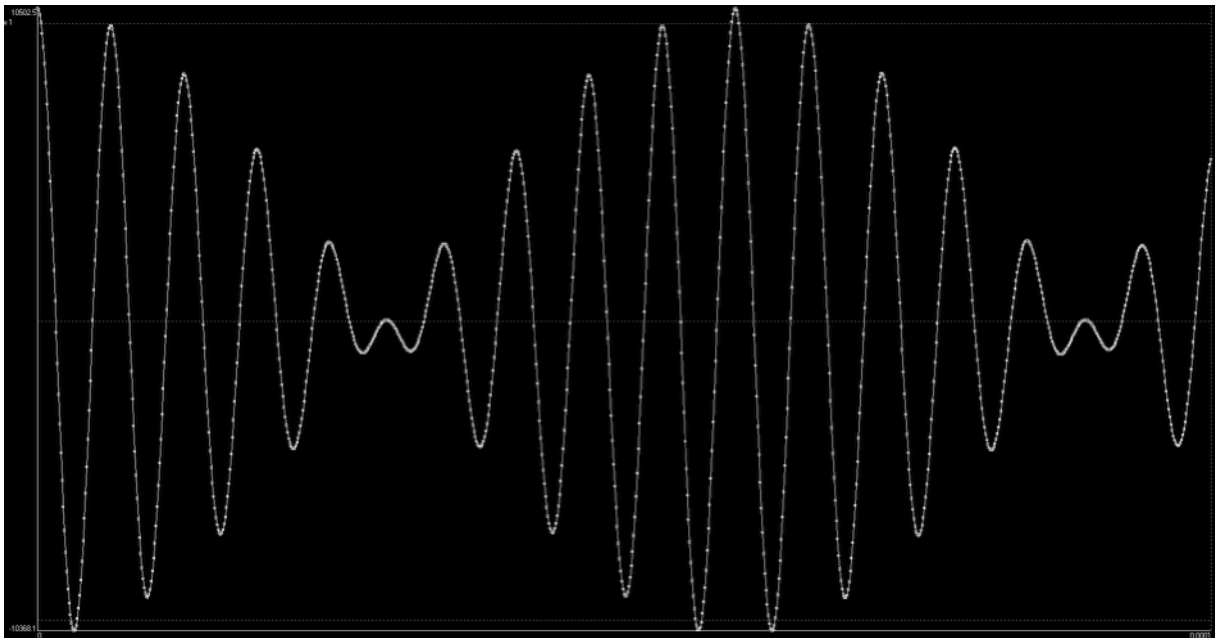


Figura 4.36: admo - tensão no nó 1