Relatório final de

Laboratórios de Comunicações 1

- Momento 4 -

Ricardo Maciel, nº 50037
ricardomcl@yahoo.com

Ângelo Alves, nº 48320
angeloalves@netcabo.pt

Joana Silva, nº 50027
a50027@alunos.uminho.pt

Grupo 204

Índice

Introdução 2

Código da versão final do programa 3

Análise Crítica 8

Conclusão 16

Introdução

O projecto deste semestre tem como objectivo a implementação de uma solução para o cálculo das tensões nos nós de um circuito eléctrico, o qual será fornecido como um dado de entrada, sob a forma de um ficheiro (netlist) que contém as informações sobre cada componente, como por exemplo o tipo de componente, o seu valor e os nós onde se encontra ligado.

Para que isto seja possível, iremos fazer uso de matérias leccionadas em outras unidades curriculares, nomeadamente a resolução de sistemas de equações usando matrizes (Álgebra Linear), as equações do sistema são obtidas pelo Método das Tensões Nodais (Componente e Circuitos Electrónicos) e por fim implementá-las em Linguagem C (Métodos de Programação I).

Neste relatório final vamos descrever as alterações e melhoramentos realizados nesta última versão do programa. Para isso vamos mostrar excertos de código das duas versões (protótipo e programa final).

Código da versão final do programa

```
/************************************************************/
/* Nome: Projecto MIECOM 1a1s – Grupo 204                   */
/* Descricao: Calculo das tensoes nos nos de um dado circuito. */
/* Data de criacao: 24/11/2006                              */
/* Ultima modificacao: 29/12/2006
/************************************************************/

#define _CRT_SECURE_NO_DEPRECATE 1
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<string.h>
#include<malloc.h>
#define DIM 50
#define GREATER(A,B) ((A) > (B) ? (A) : (B)) /* definition of macro to return the greatest between two values */

void gotoeol(FILE *in_file);  /* Declaration of the function gotoeol.      */
                              /* This function places the file pointer at  */
                              /* the end of the line.                      */
int handle_error(int code, int ln); /* function to treat error situations */

int main ()
{
   char test_str[3],              /* String that saves the first characters from a line */
        temp_v1[10], temp_v2[10], /* Temporary variables to save value1 and value2 */
        *endptr,                  /* Pointer needed by the function strtod */
        ch;                       /* Character that controls the loop of the program */
   int i, j, count, p, c, l, q,   /* Counter variables, used in 'for' cycles */
       line,                      /* Variable to count line number of the netlist file */
       comp_n,                    /* Counts the number of components in the netlist file */
       num_nodes,                 /* Counts the number of nodes of the given circuit */
                                  /* which will be used by the next variable...      */
       itc,                       /* This will keep the value of the Independent Terms Column */
       error_found;               /* This will store the return value of the function handle_error */
   struct NLTYPE{                 /* Structure to save the components description: */
      char c[3], r;               /*  - for component and reference*/
      int n1, n2;                 /*  - for node 1 and node 2 */
      float v1, v2;               /*  - for value 1 and value 2 */
      struct NLTYPE *next;        /*  - pointer to the next record of the structure */
   } *prior, *first, *current;    /* pointers used when filling the records */
   FILE *nl_file;                 /* file pointer */
   char filename[255];            /* string to store the filename */
   float matrix[DIM][DIM],        /* this is the matrix of the equations */
        aux[DIM],                 /* this is an auxiliary array used to swap lines in the matrix */
        temp;                     /* variable used in the gauss-jordan transformation */

   /* program start */
   do     /* this do..while cycle is used to control the loop of the program */
   {
     error_found = 0;   /* this is to reset the error verification variable in case of loop */
     first = NULL;    /* and this is to reset the first record created, just in case */
                      /* no records will be created in this loop.                    */
     system("CLS");  /* clears the screen */
     printf("\nIntroduza o nome (e caminho completo, se necessario) do ficheiro: ");
     scanf("%s", filename);  /* ask for filename (including path if file is not in the same folder) */
     if ((nl_file = fopen(filename,"r")) == NULL)   /* tests if file exists */
       error_found = handle_error(1,0);   /* displays error message, in case of file missing */
     else  /* in case of file found, the program continues to next phase */
     {
       /* Loading of values to the structure starts here */
       line = comp_n = 1;
       num_nodes = 0;
       while(fscanf(nl_file, "%2s", test_str) != EOF)  /* this reads the first two characters of each line     */
       {                                               /* and ends the cycle when end of file has been reached */
         strcpy(test_str, _strupr(test_str)); /* this turns those characters into uppercase if they are lowercase */
         if (!strcmp(test_str, "//")) /* if those characters are '//' (comment line)... */
           gotoeol(nl_file);  /* ...this function ignores this line, and places the file pointer */
```

```
                        /* ready to read the next line.                                    */
    else if ((!strcmp(test_str, "UR")) || (!strcmp(test_str, "R")) || (!strcmp(test_str, "I")))
         {            /* or else, if the characters correspond to any of the 3 valid components... */
           if (comp_n == 1)    /* this is needed because the first record is always a special case */
             first = current = (struct NLTYPE *) malloc(sizeof(struct NLTYPE));
           else                                                  /* creates space for the new record */
             current = (struct NLTYPE *) malloc(sizeof(struct NLTYPE));
           strcpy(current->c,test_str); /* copy the component characters to the structure variable c */
           fscanf(nl_file," %c %d %d %s", &current->r, &current->n1, &current->n2, temp_v1);
                                /* reads the values to the structure, except for value 1    */
                                /* which is kept in a string....                            */
           current->v1 = (float) strtod(temp_v1, &endptr);  /* ... and here is converted to float  */
                                                            /* and placed in the structure.        */
           if (!strcmp(test_str, "UR")) /* this condition is necessary because just the power source (UR) */
           {                            /* has a second value (v2) to be read...                         */
             fscanf(nl_file, " %s", temp_v2);    /* and here is the instruction to read it... */
             current->v2 = (float) strtod(temp_v2, &endptr);  /* ...convert it to float, and place it */
           }                                                  /* in the structure.                    */
           if (comp_n > 1)      /* once again, the first record is special, and this can't be done with that */
             prior->next = current;  /* the field 'next' of the previous record stores the location of */
                                /* the current record.                                              */
           current->next = NULL;   /* the 'next' field of this record points to nothing because the next */
                                   /* record doesn't exist yet.                                          */
           prior = current;        /* now prior is pointing to this record, so the next record */
                                     /* can be created.                                          */
           comp_n++;  /* this variable is incremented because one component has been read to the structure */
           if (current->n1 == current->n2)
           {
             error_found = handle_error(3, line); /* displays error message, sets error_found and      */
             break;                               /* exits this cycle, if the two nodes of a component */
           }                                      /* are equal.                                        */
           num_nodes = GREATER(GREATER(num_nodes, current->n1),current->n2);
         }
         else
         {
           error_found = handle_error(2, line); /* displays error message, sets error_found and exits this */
           break;                              /* cycle if other character is in the beginning of a line. */
         }
         line++;  /* this variable is just used in the messages above, to tell where the error was found */
  }
  fclose(nl_file); /* closes the file for reading, because the loading of values is done */
}

if (error_found == 0)  /* if no error was found, the program continues normally */
{
  itc = num_nodes + 1;  /* this variable will keep the value of the independent terms column. */
  for (i=0; i<DIM; i++)  /* this is to fill the matrix with zeros */
    for (j=0; j<DIM; j++)
      matrix[i][j] = 0.0;
  /* Loading of values to the matrix starts here */
  current = first;
  do /* this cycle runs through structure array of components with the count variable */
  {
    prior = current->next;
    if (!strcmp(current->c,"UR"))  /* if c is 'UR' ... */
    if ((current->n1 != 0) && (current->n2 != 0))
           /* this tests to the nodes are necessary because they are used as indexes of the matrix */
           /* where the values will be placed (summed or subtracted).                              */
    {   /* the following two instructions will sum the inverse of the resistance values (v2) */
        /* to the values stored in the diagonal of the matrix                                 */
      matrix[current->n1][current->n1] += (1/current->v2);
      matrix[current->n2][current->n2] += (1/current->v2);
        /* the following two will subtract them to the values stored outside the diagonal */
      matrix[current->n1][current->n2] -= (1/current->v2);
      matrix[current->n2][current->n1] -= (1/current->v2);
        /* this will sum the value v1/v2 (tension source divided by resistance) to the value stored */
        /* in the column of the independent terms (itc), in the line given by node 1 (n1)   */
      matrix[current->n1][itc] += (current->v1/current->v2);
        /* this will subtract the same value to the value placed in the same column, but in the given */
        /* by node 2 (n2)                                                                             */
      matrix[current->n2][itc] -= (current->v1/current->v2);
    }
    else if ((current->n1 != 0) && (current->n2 == 0))
```

```
        /* if one of the nodes is zero, only the other node will be used as index of the matrix.   */
        /* therefore placing the same values described above (1/v2 and v1/v2) only in the diagonal */
        /* and the independant terms column (itc), respectively.                                    */
      {
        matrix[current->n1][current->n1] += (1/current->v2);
        matrix[current->n1][itc] += ((current->v1)/(current->v2));
      }
      else
      {
        matrix[current->n2][current->n2] += (1/current->v2);
        matrix[current->n2][itc] -= ((current->v1)/(current->v2));
      }
      else if (current->c[0] == 'R') /* if c is 'R', the same criteria applies here, except that only */
                                     /* v1 is used and there is no operations to itc column.  */
            if ((current->n1 != 0) && (current->n2 != 0))
            {
              matrix[current->n1][current->n1] += (1/current->v1);
              matrix[current->n2][current->n2] += (1/current->v1);
              matrix[current->n1][current->n2] -= (1/current->v1);
              matrix[current->n2][current->n1] -= (1/current->v1);
            }
            else if ((current->n1 != 0) && (current->n2 == 0))
                  matrix[current->n1][current->n1] += (1/current->v1);
                else
                  matrix[current->n2][current->n2] += (1/current->v1);
          else if (current->c[0] == 'I') /* if c is 'I', the same method is applied, but it only places */
                                         /* the values in the itc column.                                */
                if ((current->n1 != 0) && (current->n2 !=0))
                {
                  matrix[current->n1][itc] += current->v1;
                  matrix[current->n2][itc] -= current->v1;
                }
                else if ((current->n1 != 0) && (current->n2 == 0))
                       matrix[current->n1][itc] += current->v1;
                     else
                       matrix[current->n2][itc] -= current->v1;
  current = current->next;
}while(prior != NULL);

/* gauss-jordan transformation */
i = 1; q = 1;    /* initialization of the variables used to run through the matrix */
while (i <= num_nodes) /* the instructions inside is executed until it reaches the last line of the matrix */
{
  for (j = q; j <= itc; j++)  /* cycles to run the matrix and search for the first non-zero element */
  {                           /* j is for the columns... */
    for (l = i; l <= num_nodes; l++) /* ...and l is for lines.  */
      if (matrix[l][j] != 0)     /* if it's found, it exits the cycle, with l and j variables */
        break;                   /* keeping its values, wich will be used again, later.       */
    if (matrix[l][j] != 0)  /* this is needed to fully exit the cycles */
      break;
  }
  /* find pivot element */
  if (matrix[i][j] == 0)  /* if the element in line i of column j (with j being the column where  */
  {                       /* the non-zero element was found in the previous cycle)...             */
    for (count = 1; count <= itc; count++)
    {
      aux[count] = matrix[i][count];       /* ... line i is swap with line l (wich is the line */
      matrix[i][count] = matrix[l][count]; /* where the non-zero element was found before).    */
      matrix[l][count] = aux[count];
    }                                      /* after this the pivot element is in the position (i,j) */
  }
  /* eliminate elements bellow pivot */
  for   (p = i+1; p <= num_nodes; p++)   /* here the elements below the pivot will be turned zero */
  {
    temp = matrix[p][j];       /* temp variable will hold the value of the element that will be      */
    matrix[p][j] = 0.0;        /* turned zero, because this will be needed in the instruction below  */
    for (count = j+1; count <= itc; count++)
      matrix[p][count] = matrix[p][count] - ((temp / (matrix[i][j])) * matrix[i][count]);
                    /* this instruction changes all elements of each line, based on the element */
                    /* that was to be turned zero and the pivot                                 */
  }
  i++; q++;    /* this will set the the index of the lines (i) and columns (q) to the next line/column */
}
i--;  /* this is necessary because in the last cycle of the previous while block i was incremented */
```

```
        /* after the last line was reached. so it must be decremented in order to set i to the       */
        /* last line of the matrix, again.                                                          */
    /* now its time to turn the pivot elements to 1 and null the elements above it */
    while (i >= 1) /* cycle terminates once i reaches the top line */
    {
      /* find the pivot element */
      for (l = i; l >= 1; l--) /* l runs the lines (from the position set by i) until it reaches the top line */
      {
        for (c = 1; c <= num_nodes; c++) /* c runs through the columns */
          if (matrix[l][c] != 0)   /* when it finds the pivot (the first non-zero element of that line) */
            break;                 /* it stops the cycle.                                               */
        if (matrix[l][c] != 0)  /* this is needed to fully exit the cycles. */
          break;
      }
      if (matrix[l][c] != 1) /* if the pivot is not 1 ... */
      {
        temp = matrix[l][c];   /* ... temp takes its value... */
        for (count = 1; count <= itc; count++)   /* ... this is will run through the columns. */
          if (count == c)              /* when it reaches the column of the pivot ... */
            matrix[l][count] = 1.0;    /* ...it explicitely turns it 1.           */
          else                                          /* all the other elements of that line    */
            matrix[l][count] = (1/temp) * matrix[l][count]; /* will be tranformed by this instruction */
      }
      /* the next cycle will null the elements above the pivot */
      for (p = l - 1; p >= 1; p--) /* this will run through the lines from above the pivot to the first */
      {
        temp = matrix[p][c]; /* temp will take the value of the element above the pivot */
        for (count = 1; count <= itc; count++) /* this will run through the columns */
          if (count == c)            /* when it reaches the columns of the pivot ... */
            matrix[p][count] = 0.0;  /* ... it is explicitely turned to zero. */
          else                                    /* all the other elements will be tranformed */
            matrix[p][count] -= (temp * matrix[l][count]);  /* by this instruction.                      */
      }
      i--; /* line index is decreased */
    }
    /* presentation of the final results wich are*/
    /* the elements of the column itc    */
    printf ("\n\nValores das tensoes:\n\n");
    for (i = 1; i <= num_nodes; i++)      /* i runs through the lines... */
      printf ("\t\tU%d = %2.2f V\n", i, matrix[i][itc]);  /* ...and each values is presented. */
  }
  current = prior = first;
  while(prior != NULL)
  {
    prior = current->next;
    free(current);
    current = prior;
  }
  printf("\nDeseja abrir outra netlist? (S/N) ");
  do
  {                                 /* this cycle will loop until the key pressed is S or N... */
    ch = (char) _getch();
  }while((ch != 'S') && (ch != 's') && (ch != 'N') && (ch != 'n')); /* ...independently of the Caps Lock */
  printf("%c\n", ch);
}while((ch == 'S') || (ch == 's'));  /* end of the first do..while cycle. Here will be decided if */
                                     /* the program starts again or not.                          */
} /* end of program */

void gotoeol(FILE *in_file) /* this function is used to ignore the comment lines in the netlist file */
{
  char test_ch; /* this variable will store each character read from the line */
  do
    test_ch = fgetc(in_file);         /* this cycle runs through the line ... */
  while((test_ch != 10) && (test_ch != EOF)); /* until it reaches the character with ASCII code 10 (wich is */
}                                             /* line feed character), meaning that it reached the end of   */
                                              /* the line, or the end of file (EOF, returned by fgetc)      */
                                              /* this will make the next fscanf function start reading the  */
                                              /* elements of the next line.                                 */

int handle_error(int code, int ln) /* this function will display error messages and return the same code */
{                                  /* that entered as a parameter.                                       */
  switch (code) /* this code will decide what message should appear */
  {
```

```
    case 1: printf ("\nFicheiro nao encontrado.\n"); break;    /* ...for file not found */
    case 2: printf ("\nEntrada invalida na linha %d.\n", ln); break;  /* ...for invalid data on the netlist file */
    case 3: printf ("\nNo 1 nao pode ser igual ao no 2. (linha %d)\n", ln); break; /* ...for equal nodes in */
  }                                                                                /* one component.        */
  return(code);  /* this return value will determine what should be done in the main program */
}
```

Análise Crítica

Para a elaboração deste código final para o projecto em questão, foram feitas algumas modificações ao código apresentado no relatório anterior (protótipo). A primeira foi a criação de mais uma função, a qual serve para tratar de erros encontrados na abertura e posterior leitura dos dados do ficheiro *netlist*. A função em questão é a *handle_error()* e a sua implementação é a seguinte:

```
int handle_error(int code, int ln)
{
  switch (code)
  {
    case 1: printf ("\nFicheiro nao encontrado.\n"); break;
    case 2: printf ("\nEntrada invalida na linha %d.\n", ln); break;
    case 3: printf ("\nNo 1 nao pode ser igual ao no 2. (linha %d)\n", ln); break;
  }
  return(code);
}
```

Esta função aceita, como parâmetros, dois valores. O primeiro (*code*), corresponde ao código do erro que foi encontrado, o qual vai depois decidir que mensagem de erro vai ser apresentada, recorrendo para isso ao *switch..case*. O segundo parâmetro (*ln*), corresponde à linha onde foi encontrado o erro, a qual vai também ser usada nas mensagens de erro. O valor devolvido pela função, através da função *return()*, é o mesmo que é passado como parâmetro, pois o mesmo será atribuído, na função principal à variável *error_found*, como se pode ver em baixo.

Ficheiro não encontrado:

```
if ((nl_file = fopen(filename,"r")) == NULL)
  error_found = handle_error(1,0);
```

Os valores dos nós de um componente são iguais:

```
if (current->n1 == current->n2)
{
  error_found = handle_error(3, line);
  break;
}
.
.
.
```

Foi encontrada uma entrada inválida no ficheiro:

```
if ((!strcmp(test_str, "UR")) || (!strcmp(test_str, "R")) || (!strcmp(test_str, "I")))
{
  .
  .
}
else
{
  error_found = handle_error(2, line);
  break;
}
```

A função break, usada nas duas ultimas condições, vai determinar a saída imediata do ciclo *while* que rege a leitura dos dados da *netlist*. A variável *error_found* vai então servir de controlo para determinar a interrupção da execução do programa, sem contudo o dar completamente por terminado, como vai ser descrito mais à frente. As condições de controlo são as seguintes:

```
if ((error_found == 2) || (error_found == 3))
      {
        .
        .
  .
}
else if (error_found == 0)
      {
        .
        .
        .
        .
        .
}
```

A primeira condição testa se os erros foram encontrados aquando da leitura dos dados (depois de já aberto o ficheiro) e a seguinte, como é obvio, é para o caso de não ter sido encontrado nenhum erro, o que vai permitir a normal execução do programa. No caso de terem sido encontrados erros durante a leitura, é executado um pedaço de código (o qual vai ser explicado mais à frente e que tem que ver com a alocação dinâmica). Depois de executado esse código o programa passa para as instruções transcritas abaixo. O mesmo acontecerá no caso de o erro encontrado ter ocorrido durante a abertura do ficheiro (*code=1*).

```
printf("\nDeseja abrir outra netlist? (S/N) ");
do
{
  ch = (char) _getch();
}while((ch != 'S') && (ch != 's') && (ch != 'N') && (ch != 'n'));
printf("%c\n", ch);
```

Como é fácil de perceber, estas instruções vão controlar se o programa deve ou não ser executado, perguntando ao utilizador se pretende abrir outro ficheiro, tal como já foi referido acima. De salientar que isto também acontecerá, mesmo que não tenham sido encontrados erros, ou seja, logo após terem sido apresentados os resultados pretendidos.

Para conseguir que o programa seja de novo executado, todas as instruções estão contidas num ciclo *do..while()*, o qual vai apenas permitir a repetição da execução, se o utilizador carregar na tecla 'S' durante o pedido anterior:

```
do
{
  .
  .
  .
  .
  .
  .
  .
}while((ch == 'S') || (ch == 's'));
```

Outra alteração reside, em concreto, na leitura dos dados do ficheiro para a memória, que agora ficou mais simples, pois chegámos à conclusão de que bastava usar uma condição que englobasse os três tipos de elementos (UR, R e I). Para melhor se perceber do que se está a falar, nada melhor do que os excertos de código de cada situação (protótipo e programa final).

Aqui está a leitura dos dados no protótipo:

```
if (!strcmp(test_str, "UR"))
{
  strcpy(netlist[comp_n].c,test_str);
  fscanf(nl_file," %c %d %d %s %s", &netlist[comp_n].r, &netlist[comp_n].n1,
&netlist[comp_n].n2, temp_v1, temp_v2);
        netlist[comp_n].v1 = (float) strtod(temp_v1, &endptr);
        netlist[comp_n].v2 = (float) strtod(temp_v2, &endptr);
  comp_n++;
}
else if (!strcmp(test_str, "R"))
     {
       strcpy(netlist[comp_n].c,test_str);
       fscanf(nl_file, " %c %d %d %s", &netlist[comp_n].r, &netlist[comp_n].n1,
&netlist[comp_n].n2, temp_v1);
       netlist[comp_n].v1 = (float) strtod(temp_v1, &endptr);
       comp_n++;
     }
     else if (!strcmp(test_str, "I"))
          {
                 strcpy(netlist[comp_n].c,test_str);
            fscanf(nl_file, " %c %d %d %s", &netlist[comp_n].r, &netlist[comp_n].n1,
&netlist[comp_n].n2, temp_v1);
            netlist[comp_n].v1 = (float) strtod(temp_v1, &endptr);
            comp_n++;
                }
          else
          {
            printf("Invalid data on line %d.\n",line);
            _getch();
            return (1);
          }
if ((netlist[comp_n-1].c[0] != 'I') && (netlist[comp_n-1].n1 == 0))
{
  printf("\n\nNo 1 nao pode ser zero. (linha %d)\n", line);
  _getch();
  return(1);
}
     if ((comp_n > 1) && (netlist[comp_n-1].n1 == netlist[comp_n-1].n2))
{
```

```
  printf("\n\nNo 1 nao pode ser igual a No 2. (linha %d)\n", line);
  _getch();
  return(1);
}
      line++;
      .
      .
      .
```

E esta é a leitura dos dados no programa final:

```
if ((!strcmp(test_str, "UR")) || (!strcmp(test_str, "R")) || (!strcmp(test_str, "I")))
{
  if (comp_n == 1)
    first = current = (struct NLTYPE *) malloc(sizeof(struct NLTYPE));
  else
    current = (struct NLTYPE *) malloc(sizeof(struct NLTYPE));
  strcpy(current->c,test_str);
  fscanf(nl_file," %c %d %d %s", &current->r, &current->n1, &current->n2, temp_v1);
  current->v1 = (float) strtod(temp_v1, &endptr);
  if (!strcmp(test_str, "UR"))
  {
    fscanf(nl_file, " %s", temp_v2);
    current->v2 = (float) strtod(temp_v2, &endptr);
  }
  if (comp_n > 1)
    prior->next = current;
  current->next = NULL;
  prior = current;
  comp_n++;
  if (current->n1 == current->n2)
  {
    error_found = handle_error(3, line);
    break;
  }
  num_nodes = GREATER(GREATER(num_nodes, current->n1),current->n2);
}
else
{
  error_found = handle_error(2, line);
  break;
}
line++;
   .
   .
   .
```

Como se pode ver, numa só condição lêem-se todos os dados relativos a uma linha da *netlist*, seja ela referente a uma fonte de tensão (UR), fonte de corrente (I) ou resistência (R). No entanto, como UR tem mais um valor que os restantes componentes (*v2*), este precisa de ser lida à parte, depois dos restantes dados serem lidos. Dentro deste bloco consta também, agora, a análise do número de nós, apenas com a seguinte instrução:

```
num_nodes = GREATER(GREATER(num_nodes, current->n1),current->n2);
```

Aqui foi usada uma macro (definida no topo do programa):

```
#define GREATER(A,B) ((A) > (B) ? (A) : (B))
```

a qual devolve o maior de dois números dados. Como a variável *num_nodes* precisa de ser comparada com os dois nós, usámos a mesma macro de um modo encadeado de modo a que numa só instrução, o resultado pretendido

seja o mesmo que o conseguido com as instruções que constavam no protótipo e o qual era executado depois do ciclo de leitura da netlist:

```
for (count=1; count < comp_n; count++)
{
  if (netlist[count].n1 > num_nodes)
    num_nodes = netlist[count].n1;
  if (netlist[count].n2 > num_nodes)
    num_nodes = netlist[count].n2;
}
```

Outra das grandes alterações que já aqui foi referida mais acima, foi a introdução da alocação dinâmica de memória. Contudo, apenas conseguimos fazê-lo para a estrutura de dados que guarda em memória a *netlist* que é lida do ficheiro. Para isso baseamo-nos num dos exemplos apresentados durante as aulas de Métodos de Programação I. O método começa na declaração das variáveis, onde é declarada a estrutura e respectivos apontadores que vão servir de apoio para o armazenamento em memória dos dados lidos do ficheiro:

```
struct NLTYPE{
  char c[3], r;
  int n1, n2;
  float v1, v2;
  struct NLTYPE *next;
} *prior, *first, *current;
```

As primeiras 6 variáveis desta estrutura, já não são novidade, pois já são usadas desde o algoritmo e no relatório referente a esse, já foram explicadas. A diferença aqui reside no apontador *next*, o qual vai guardar o endereço da memória, onde está guardado o registo seguinte, ou seja, vai apontar para o componente seguinte da *netlist*. Os restantes apontadores, declarados, (**prior*, **first* e **current*) vão ser usados como apoio durante a escrita e posterior leitura dos dados. Em resumo, **first* vai apontar sempre para o primeiro registo (componente), **current* vair apontar para o registo que está a ser preenchido/lido em cada ciclo e **prior* tanto apontará para o anterior, como para o posterior, conforme esteja a ser efectuado o preenchimento ou a leitura da estrutura, respectivamente. O preenchimento da estrutura, ou seja, a leitura dos dados do ficheiro para a memória, é efectuado da seguinte forma:

Antes de mais é necessário criar espaço em memória para os dados a serem lidos, o que é conseguido através destas instruções:

```
if (comp_n == 1)
  first = current = (struct NLTYPE *) malloc(sizeof(struct NLTYPE));
else
  current = (struct NLTYPE *) malloc(sizeof(struct NLTYPE));
```

A condição (*comp_n == 1*) é necessária, porque o preenchimento do primeiro registo em memória é sempre um caso especial, isto é, aqui o apontador **first* vai ser usado para guardar o endereço de memória deste primeiro registo, o qual vai ser atribuído pela função *malloc().* Ao mesmo tempo, **current* guardará o mesmo endereço pois será com este apontador que as operações de preenchimento irão trabalhar. No caso dos restantes componentes da *netlist* (depois do primeiro ser lido), apenas **current* será usado, como se pode verificar depois do *else*.

A passagem dos dados do ficheiro para a memória reservada acima, é feita normalmente, só que com a pequena diferença de que para aceder aos diferentes campos da estrutura é usada outra notação, pois aqui estamos a trabalhar com apontadores e não com variáveis reais, como acontecia no protótipo. Ou seja, para aceder aos vários campos da estrutura é usada a seguinte notação:

current->c ou *current->r* ou *current->n1* , etc…

Finda a leitura dos dados de um componente, é necessário preparar os apontadores para a leitura do seguinte componente. Isso é feito com as seguintes instruções:

```
if (comp_n > 1)
  prior->next = current;
current->next = NULL;
prior = current;
```

Mais uma vez é usada uma condição baseada na variável *comp_n*, pois aqui também é preciso diferenciar o primeiro registo dos restantes. Como já foi dito antes, o apontador **prior*, depois de preenchido o primeiro registo, irá apontar para o registo anterior, em cada iteração do ciclo. Por isso é usada a condição (*comp_n > 1*), pois a instrução que se encontra dentro dela, só fará sentido depois de preenchido o primeiro registo. A seguinte instrução, a qual será sempre executada, independente de o componente actual a ser lido, ser o primeiro ou não, vai atribuir um valor nulo, como é óbvio, ao campo *next* do registo actual (*current*), pois o componente seguinte ainda vai ser lido (ou não, caso este seja o último). Se este não for o ultimo componente da *netlist*, este campo será preenchido na seguinte iteração do ciclo, com a instrução que está dentro da condição, pois **prior*, por efeito da última instrução (*prior = current*), estará a apontar para o registo actual (e anterior, do ponto de vista da seguinte iteração).

Em seguida, os dados agora presentes em memória, serão lidos e colocados na matriz final, conforme é apresentado a seguir:

```
current = first;
do
{
  prior = current->next;
      .
      .
      .
      .
      .
  current = current->next;
}while(prior != NULL);
```

Como neste momento estamos a falar da alocação dinâmica, não achamos pertinente, para já, a inclusão do código relativo à distribuição dos dados pela matriz, pois esse assunto será coberto mais à frente, devido a um erro que foi já mencionado no relatório anterior. Conforme se pode ver, a primeira instrução está a recorrer ao endereço do primeiro registo (guardado por *first*) e a guardá-lo em *current*, que vai ser o apontador usado nas instruções de preenchimento da matriz. Já dentro do ciclo, a primeira instrução consiste na atribuição do endereço do registo seguinte a *prior* (*prior = current->next*), pois este será usado na condição de paragem do ciclo, ou seja, quando o último registo for lido, pois o campo *next* do ultimo registo vai ter um valor nulo, conforme se pode verificar analisando as instruções de preenchimento dos registos. No fim do ciclo (quando os valores do registo actual tiverem sido distribuídas pela matriz), *current* vai receber o endereço do seguinte registo, com a instrução *current = current->next*, de modo a que na próxima iteração do ciclo, as operações recorram sempre ao registo seguinte, até chegar ao último.

Finda a apresentação dos resultados (ou não, em caso de erro), é efectuada a libertação da memória ocupada pelos registos dos componentes (se foram criados alguns), como mandam as regras da boa programação:

```
current = prior = first;
while (prior != NULL)
{
  prior = current->next;
  free(current);
  current = prior;
}
```

Aqui o procedimento é exactamente o mesmo utilizado durante a leitura dos dados para a matriz, com a diferença de que *prior* também vai guardar o endereço do primeiro registo (caso exista algum), juntamente com *current* e a condição de paragem é executada logo no inicio, sem que haja a hipótese de se efectuar as instruções interiores, no caso de ter acontecido um erro e,

consequentemente não ter sido criado nenhum registo. A instrução ali inserida (*free(current);*) é a que liberta a memória que o registo actual, em cada iteração do ciclo, está a usar.

A outra correcção que fizemos, a qual também já foi mencionada acima, tem que ver com a distribuição dos dados da *netlist* pela matriz (*matrix*). Desde o algoritmo temos usado como critério a não aceitação de componentes que tenham o valor 0 no nó 1. Mas, como já foi referido no último relatório, chegámos à conclusão de que tal afinal é um elemento aceitável para o nosso projecto. Assim, procedemos à modificação do código de modo a retirar essa restrição, durante a leitura do ficheiro. No que toca à distribuição dos dados pela matriz, também foram efectuadas modificações nesse sentido.

Conclusão

A realização deste projecto foi bastante positiva para nós, pois para que o pudessemos fazer, tivemos que recorrer a matérias aprendidas em alguma das unidades curriculares leccionadas, o que fez com que tivessemos a matéria sempre em dia.

Do ponto de vista da programação, este projecto exigiu bastante de todos os elementos do grupo, visto ter um grau de complexidade bastante elevado.

A implementação em C foi talvez a parte mais difíl de todo o projecto, visto termo-nos deparado com algumas dificuldades nesta fase. Nomeadamente a passagem do sistema da netlist para uma matriz e depois resolve-la para a forma de escada e escada reduzida (método Gauss-Jordan). Outra dificuldade apareceu quando chegámos à parte do método de Gauss-Jordan, pois no algoritmo utilizamos alguns 'saltos' e como isso não é permitido no nosso código, tivemos de efectuar algumas modificações.

Chegar a esta versão final do programa não foi uma tarefa muito difícil, pois como já havia sido dito, a elaboração do algoritmo e do código do protótipo, foi o que exigiu de nós um esforço maior, dado que tivemos de começar do nada.