Gauss

Um dos algoritmos que faz parte da nossa cultura, e que não aprendemos nas cadeiras de programação, é o método de Gauss, para resolver sistemas de equações lineares.

Abreviarei os detalhes, porque toda a gente conhece. O que queremos fazer agora é, bem entendido, programar o método de Gauss.

Os dados serão o sistema de equações e o resultado será a solução do sistema, se houver, ou uma mensagem apropriada, se não houver.

Encontrará na Internet explicações completas e código em várias linguagens, por exemplo aqui. Não perca tempo, a não ser que queira aproveitar para conhecer aquelas linguagens todas. Nós não programamos desta maneira, pois não?

Ler o sistema

O sistema de equações é dado na forma matricial: havendo **N** equações, teremos uma matriz com **N** linhas e (**N**+1) colunas. A última coluna é a coluna dos termos independentes.

Por exemplo, o seguinte sistema de equações

```
2x  + y  − 3z = −1
−x  + 3y + 2z = 12
3x  + y  − 3z = 0
```

será representado pela matriz

```
 2  1 -3 -1
-1  3  2 12
 3  1 -3  0
```

A primeira tarefa será ler o sistema, construir a matriz, e, para confirmar, escrever a matriz novamente na consola.

Temos ao nosso dispor, bem entendido, a biblioteca `our_doubles` para arrays de `double` e as funções sobre matrizes estudadas nas aulas.

Escreva então uma função para ler a matriz com o sistema, a partir da consola. O protótipo deve ser o seguinte:

```
void get_system(int x, double a[x][x+1]);
```

Aqui, o parâmetro de entrada `x` representa o número de equações; o parâmetro de saída `a` é a matriz que receberá os dados.

Note que no exemplo os coeficientes parecem números inteiros, mas devem ser tratados como números `double`.

Programe também uma função para mostrar o sistema, com o seguinte protótipo:

```
void print_system(int x, const double a[x][x+1]);
```

Tal como antes, o parâmetro `x` representa o número de equações e o parâmetro `a` representa o sistema.

O sistema será mostrado em forma matricial. Cada coluna deve ter largura 12. Os números devem ser escritos com `%12f`. Por exemplo, o sistema acima deve aparecer na seguinte forma:

```
    2.000000    1.000000   -3.000000   -1.000000
   -1.000000    3.000000    2.000000   12.000000
    3.000000    1.000000   -3.000000    0.000000
```

Escreva então a função de teste, que primeiro lê o número de equações, depois lê o sistema completo, linha a linha, e finalmente escreve o sistema no formato indicado. No caso do exemplo, o input seria assim:

```
3
2 1 -3 -1
-1 3 2 12
3 1 -3 0
```

Em rigor, não teria de ser exatamente assim. O que importa é que se o primeiro número for **X** seguem-se X * (X+1) números, arrumados de maneira indeterminada.

Requisito técnico: em nenhuma função deste guião haverá um `for` dentro de um `for`, nem um `if` dentro de um `if`, nem um `for` dentro de um `if`.

Submeta no problema A.

Verificação da solução

Resolver um sistema de equações pode ser complicado e trabalhoso. Ao invés, verificar uma solução é relativamente simples. Por "verificar uma solução" queremos dizer determinar se uma dada atribuição de valores às incógnitas constitui uma solução do sistema ou não.

No respeitante à programação, o que temos de fazer é multiplicar a matriz dos coeficientes (sem a coluna dos termos independentes) pelo array das soluções e comparar com o array que representa a coluna dos termos independentes.

Repare que, do ponto de vista da álgebra, multiplicar uma matriz por um array corresponde a multiplicar uma matriz **R***C (com **R** linhas e **C** colunas) por uma matriz **C***1, sendo o resultado uma matriz **R***1. Mas note que a "coluna" com a solução a testar é dada pelo array, não por uma matriz **R***1.

Programe então uma função `is_solution`, que, dados um sistema de equações e um array, devolva 1 se o array constituir uma solução e 0 se não.

Programe uma função de teste que aceite um sistema (primeiro o número de equações, depois o sistema propriamente dito, como no caso anterior) e depois

uma sequência de linhas, cada uma com tantos números como o número de incógnitas, e que, para cada uma dessas linhas escreva 1 ou 0, numa linha do output consoante os números dessa linha constituírem uma solução do sistema ou não.

Note que neste exercício pode haver no input sistemas indeterminados, com mais que uma solução.

Eis um exemplo de input para este problema:

```
2
2 3 13
5 -1 7
0 0
1 2
2 3
3 2
5 1
```

O correspondente output será assim:

```
0
0
1
0
0
```

Submeta no problema B.

Resolvendo à força bruta

Visto que já somos capazes de verificar soluções automaticamente, podemos resolver sistemas de equações *à bruta*: experimentamos todas as "soluções possíveis" e guardamos as que forem soluções verdadeiras. Neste contexto, uma "solução possível" é qualquer array de valores com tantos elementos quantos o número de incógnitas.

Isso só pode ser piada, dirá você, porque o número de soluções possíveis é infinito.

No entanto, para aqueles sistemas que mais nos moeram a cabeça nas aulas de álgebra até daria, digo eu, porque os professores arranjam quase sempre exemplos em que os valores das incógnitas são números inteiros pequenos.

Programe então uma função `brute_force`, que dado um sistema de equações e um número inteiro positivo, calcule, se houver, a solução desse sistema em que todos os valores das incógnitas são números inteiros menores ou iguais em valor absoluto a esse número dado.

Programe também uma função de teste que aceita um sistema (como antes) e ainda, na linha seguinte, um valor para o tal número inteiro positivo. A função escreverá uma linha com os valores das incógnitas, na forma de números inteiros (separados cada um do seguinte por um espaço), ou `"***"` (sem as aspas), se nenhuma solução for encontrada.

Note que estamos apenas a considerar sistemas de N equações a N incógnitas. Sistemas destes podem ser determinados, isto é, ter uma única solução, indeterminados, isto é, ter mais do que uma solução, ou impossíveis, isto é, não ter solução. Para efeitos deste exercício, todos os casos de teste representam sistemas determinados e o tal número positivo será escolhido de maneira que não haja mais que 1000000 de casos a experimentar.

O input é análogo ao do problema A. O array com a solução deve ser escrito com a função `doubles_println`. No caso do sistema de equações do problema B, o resultado será:

```
2 3
```

Submeta no problema C.

Método de Gauss

No método de Gauss que eu aprendi, a primeira fase dos cálculos consistia em triangularizar a matriz, somando linhas umas com as outras de maneira a anular os coeficientes abaixo da diagonal. Faremos isso aqui também, mas à maneira da programação.

Na verdade, o que queremos é anular *todos* os coeficientes numa certa coluna, exceto o da diagonal e fazer isso para todas as colunas (se conseguirmos…). Nessa altura, teremos um sistema diagonal (com coeficientes zero em todo o lado exceto na diagonal, onde nenhum será zero) e o sistema resolve-se nas calmas.

Uma vez escolhida a coluna `x`, a técnica será dividir cada linha do sistema pelo valor da coluna `x` nessa linha. Resultará uma matriz em que a coluna `x` vale sempre 1.0 ou 0.0. Note que a divisão só se faz se se puder fazer, isto é, se o valor na coluna `x` não for zero. Se for zero, melhor, deixa-se a linha como está.

Programe então a função `columnize`, para realizar esta operação, em relação à matriz, para um certa coluna: o protótipo deve ser o seguinte:

```
void columnize(int n, double a[n][n+1], int x);
```

Sugestão: programe primeiro uma função que, dado um array de `double` e um número `double` diferente de 0.0, construa outro array em que cada elemento é o quociente da divisão do correspondente elemento do array dado pelo número dado. Note bem, esta função é muito simples e é muito parecida com outras funções que programámos sobre arrays. Trata-se, mais uma vez, de *mapear* uma array noutro, aplicando a cada elemento do array uma dada função.

Aplicando a função `columnize` à primeira coluna do sistema usado acima como exemplo, resultaria o sistema que mostrado (com `print_system`) apareceria assim:

```
1.000000    0.500000   -1.500000   -0.500000
1.000000   -3.000000   -2.000000  -12.000000
1.000000    0.333333   -1.000000    0.000000
```

A seguir, para anular os valores da coluna 1, a partir da segunda linha, subtraímos de cada linha a primeira linha. Obteríamos o seguinte:

```
1.000000    0.500000   -1.500000   -0.500000
0.000000   -3.500000   -0.500000  -11.500000
0.000000   -0.166667    0.500000    0.500000
```

Programe esta operação, por meio de uma função `subtract_row`, que dada uma matriz e o índice de uma linha dessa matriz subtraia de cada uma das linhas a li-

nha dada, exceto da própria linha. Isto é, a linha dada fica na mesma. O protótipo é o seguinte:

```
void subtract_row(int n, double a[n][n+1], int x);
```

A seguir, faz-se o mesmo para a segunda coluna. Obtém-se primeiro, após `columnize`:

```
 2.000000    1.000000   -3.000000   -1.000000
 0.000000    1.000000    0.142857    3.285714
 0.000000    1.000000   -3.000000   -3.000000
```

E depois de `subtract_row` para a segunda linha, obtém-se o seguinte.

```
 2.000000    0.000000   -3.142857   -4.285714
 0.000000    1.000000    0.142857    3.285714
 0.000000    0.000000   -3.142857   -6.285714
```

Repetindo para a terceira coluna, teríamos as duas seguintes configurações, primeiro após `columnize`:

```
-0.636364    0.000000    1.000000    1.363636
 0.000000    7.000000    1.000000   23.000000
 0.000000    0.000000    1.000000    2.000000
```

e a seguir após `subtract_row`:

```
-0.636364    0.000000    0.000000   -0.636364
 0.000000    7.000000    0.000000   21.000000
 0.000000    0.000000    1.000000    2.000000
```

Este último é sistema diagonal, que se resolve facilmente:

```
 1.000000    0.000000    0.000000    1.000000
 0.000000    1.000000    0.000000    3.000000
 0.000000    0.000000    1.000000    2.000000
```

A solução está na última coluna.

Programe uma função para realizar estas operações e calcular a solução, num array de saída. Programe também a função de teste, que lê o sistema, como o problema A e escreve a solução, como no problema C.

Para o caso do exemplo, o output será o seguinte:

```
1 3 2
```

Submeta no problema D.

The devil is in the details 👹

O método explicado na secção anterior foi capaz de resolver o sistema de equações usado no exemplo e os outros sistemas de equações usados no Mooshak, porque todos esses sistemas eram "bonzinhos". Com outros sistemas menos favoráveis, o método, tal como descrito acima, não funcionaria. De facto, omitimos alguns aspetos do problema, que, não obstante, são essenciais.

Caso dos menos zero

O esquema geral de resolução de sistemas de equações lineares é o que expusemos. Ainda assim, se programou como explicado no guião, mostrando a matriz após cada `columnize` e após cada `subtract_row`, os valores escritos talvez não tenham sido exatamente aqueles. Em particular, alguns dos zeros nas tabelas terão aparecido com sinal menos: `-0.000000`.

De facto, com os números `double`, em C, há o zero e o menos zero!!! É uma questão muito engraçada, que, na prática, não tem consequência na aritmética, mas apenas na maneira como os valores são mostrados. Para perceber o menos zero, corra a seguinte função de teste:

```
void test_minus_zero(void)
{
  double x = 0.0;
  double y = x * -2.0;
  double z = x * 2.0;
  printf("%g %g %g\n", x, y, z);
}
```

No meu computador obtive:

```
0 -0 0
```

Quer isto dizer que `y` e `z`, valendo ambos zero, são diferentes?

Experimentemos de novo:

```
void test_minus_zero_bis(void)
{
  double x = 0.0;
  double y = -0.0;
  printf("%g %g\n", x, y);
  int b = x == y;
  printf("%d\n", b);
  int *p = (int *)&x;
  int *q = (int *)&y;
  ints_println(p, 2, " ");
  ints_println(q, 2, " ");
}
```

Desta vez obtive:

```
0 -0
1
0 0
0 -2147483648
```

A primeira linha mostra que os dois zeros são diferentes; a segunda mostra que têm mesmo valor. A terceira linha mostra a memória do zero "positivo", por meio dos valores inteiros registados em cada uma das duas palavras que contêm o número `double` com o zero positivo. E analogamente para a quarta linha, em relação ao zero negativo. Constatamos que as duas representações são diferentes. No entanto, tal como já observámos (na segunda linha) os dois valores (tendo representações diferentes) são aritmeticamente iguais.

Observamos também que na memória o *mais zero* é representado por 64 bits (duas palavras de 32 bits), todos valendo zero. Por sua vez, no *menos zero*, o primeiro bit é 1 e os outros 63 bits são zero. Concluímos que esse primeiro bit é o bit do sinal: zero representa o sinal mais e 1 representa o sinal menos. O valor absoluto, por assim dizer, é zero em ambos os casos.

Não se confunda por o bit do sinal surgir na segunda das duas palavras (de 32 bits) que representam um `double` na memória. É assim no meu computador, mas não é obrigatório que seja assim em todos os computadores.

Charada: será que o zero negativo é um número negativo?

O menos zero não atrapalha os cálculos, mas desfeia os resultados impressos? Como podemos evitá-lo?

Com uma instrução deveras estranha:

```
if (x == 0) x = 0;
```

No nosso caso, antes de escrever a matriz, faríamos isto a todos os elementos da matriz e depois escreveríamos normalmente.

No entanto, faremos outra coisa, mais útil e mais importante, que por tabela resolve também a questão do menos zero.

Note que os casos de teste usados para o problema D foram escolhidos de maneira a nenhuma incógnita valer 0.0, para evitar a possibilidade de os cálculos darem -0.0, o que complicaria a análise dos resultados pelo Mooshak, ainda que o valor -0.0 estivesse certo aritmeticamente.

Questão dos valores aproximados

Uma primeira observação fundamental: nem todos os números reais que têm uma representação decimal finita têm uma representação exata através de um número `double`. Isto pode parecer surpreendente, mas até nem é, se pensarmos que os números `double` são representados por uma soma potências de 2 de expoente negativo, soma essa multiplicada por uma potência de 2. Por exemplo, o número 6.0 é representado pela soma 0.5+0.25 multiplicada por 8. Internamente, há um bit para cada potência de dois (de expoente negativo) e mais alguns bits para o tal expoente da potência de 2. (No exemplo, o expoente seria 3.)

Outro exemplo: 10.0. Neste caso temos 10.0 = 16 * (0.5 + 0.125). Aqui, o expoente é 4 e as potências são 2^-1 e 2^-3. Quer dizer, os tais bits serão `101` e depois tudo zero.

Ainda um terceiro exemplo: 2.5. Este até é fácil, se repararmos que 2.5 é 10.0 dividido por 4. Sendo 4 uma potência de 2, é claro que, então, 2.5 = 4 * (0.5 + 0.125). Portanto, os bits correspondentes aos expoentes negativos são iguais aos do número 10.0, mas o expoente agora é 2 (em vez de 4).

Agora a parte espantosa: o número 0.1 não tem uma representação exata enquanto número `double`. Porquê? Porque não existe nenhuma combinação finita de potências negativas de 2 cuja soma seja 0.1.

O fenómeno, que é um facto matemático fácil de demonstrar, é o mesmo a que estamos habituadíssimos com a notação decimal. Por exemplo: número representado pela fração 1/3 não é representável por uma dízima finita. E não o é porque não existe nenhuma soma finita de potências de 10 que dê exatamente 1/3.

Voltando agora aos cálculos com números `double`.

Ao fazermos no computador uma operação com números que não têm representação exata, forçosamente o resultado obtido não será a representação exata do resultado matemático "perfeito". Em muitos casos, o erro relativo será pequeno e não afetará a utilidade do resultado. Mas, em casos patológicos o resultado não será aproveitável de todo. Vejamos um exemplo, com o seguinte sistema de equações:

```
0.1x + 0.7y + z = 18
0.3x + 2.1y = 24
y + z = 20
```

Claramente, de cabeça ou à mão, podemos concluir que a solução é x=10, y=10 e z=10. No entanto, correndo um programa que implemente à letra as regras que descrevemos, obtive x=0, y=10 e z=10. Algo terá corrido terrivelmente mal. 😭

Aplicando as regras, primeiro dividimos a primeira equação por 0.1, a segunda por 0.3 e deixamos a terceira na mesma obtendo o seguinte sistema, equivalente ao primeiro:

```
x + 7y + 10z = 180
x + 7y = 80
y + z = 20
```

Mas, isto foi fazendo as contas nós próprios, com base na notação decimal. Ora no programa, quem faz as contas é o computador, que usa (por assim dizer) a notação binária, em que os números decimais 0.1. 0.7. 0.3 e 2.1 não são repre-

sentáveis. Quer dizer, o computador faz as contas com *outros* números, não com aqueles que são dados! Os números do computador são aproximações dos números dados, mas nem sempre são iguais aos números dados. Neste caso, não são, e o que se passou foi que os coeficientes de *y* na primeira e segunda equações não ficaram iguais. Isso aconteceu porque dividindo a aproximação de 0.7 pela aproximação de 0.1 e dividindo a aproximação de 2.1 pela aproximação de 0.3, o computador chegou a quocientes diferentes. Vendo bem, não é de admirar.

A seguir, o computador, seguindo as ordens do programa, subtraiu da segunda equação a primeira (ambas já na forma em que o coeficiente de *x* vale 1.0). Como os coeficientes de *y*, sendo muito próximos, não são iguais, a diferença é um número muito pequeno, da ordem de 10^-15, mas não é zero.

No passo seguinte, a segunda equação será dividida pelo coeficiente de *y*, que é o tal número da ordem de 10^-15. Esta divisão por um número muito pequeno amplifica o erro dos cálculos descomunalmente, e os valores deixam de ter significado.

Note que sempre que trabalhamos com valores aproximados há um "erro" em relação ao valor "exato". No entanto, normalmente esse erro é pequeno e desprezável, até porque os valores "exatos" não são exatos à partida, tendo sido obtidos por medição de alguma grandeza física (comprimento, velocidade, temperatura, etc.), o que envolve sempre alguma imprecisão. Em casos patológicos como o que descrevemos, o erro pode "disparar" e fica tudo estragado.

Que podemos fazer para evitar esta triste situação?

Podemos fazer batota, e supor que sempre que nos cálculos com valores aproximados surja um número muito pequeno, isso deve ser porque o resultado certo seria zero mesmo. Quer dizer, depois de `subtract_row`, chamaremos uma função que transformará em zero todos os elementos da matriz cujo valor seja muito pequeno.

Na gíria da programação "muito pequeno" quer dizer "menor que *épsilon* em valor absoluto". Épsilon será o valor que estabelece o limite da pequenez, e que deverá ser afinado em cada problema.

Programe então uma função `epsilonify` que transforma em zero os valores da matriz menores que épsilon em valor absoluto:

```
void epsilonify(int n, double a[n][n+1], double epsilon);
```

Esta função transforma zeros em zeros. Por isso, resolve também o problema do menos zero. 😀

No nosso problema o valor 10^-13 é razoável para épsilon, sabendo nós que a precisão dos números `double` é de 16 casas decimais.

Os casos de teste usados para o problema D foram escolhidos de maneira a evitar estas complicações…

Seleção da linha subtrativa

A descrição do algoritmo de diagonalização, acima, tem uma falha imperdoável: sugere que para anular o coeficiente da coluna *i* (após ter anulado os coeficientes das colunas 0, 1, … *i*-1), devemos subtrair a linha *i* de cada uma das outras linhas em que o coeficiente da coluna *i* não é já zero. Ora, isto só funcionará se o coeficiente na linha *i*, coluna *i*, não valer zero! Note, que após termos "colunizado" a coluna *i*, cada um dos valores nessa coluna será 1.0 ou 0.0. Se for zero, ao subtrairmos a linha *i* das outras, as outras não verão o seu coeficiente na coluna *i* anulado, que era o que pretendíamos (a não ser que ele já valesse zero antes).

Portanto, as coisas fiam mais fino: a linha subtrativa não pode valer zero na posição da diagonal. Que fazer então? Se a linha subtrativa valer zero na diagonal, então trocamos a linha por outra mais abaixo que tenha o valor 1.0 nessa coluna. Esta operação equivale a trocar duas equações e é certamente benigna.

Podemos exprimir o mesmo raciocínio sem o famigerado "se": antes de "colunizar" a coluna *i*, trocamos a linha *i* com a primeira linha a partir da linha *i* para a qual o coeficiente na coluna *i* seja 1.0. Só será preciso trocar mesmo se a tal "primeira linha a partir, etc." não for a própria linha *i*.

Os casos de teste usados para o problema D foram escolhidos de maneira que nunca surgisse um zero na diagonal, o que constitui uma limitação muito grande.

Caso dos sistemas impossíveis ou indeterminados

Esquecemo-nos ainda de um caso, na secção anterior: que acontece se em todas as linhas a partir da linha i, o valor da coluna i for 0.0? Nesse caso, estamos tramados: não temos como prosseguir os cálculos.😡

Por outro lado, podemos concluir que, nesse caso, o sistema ou é impossível, ou é indeterminado. Portanto, é legítimo desistir de tentar resolvê-lo.

Resta a questão técnica de como abandonar os cálculos de maneira ordeira, após termos descoberto que não dá para continuar. Sugiro a técnica da função `error`, explicada no livro de Kernighan & Ritchie:

```
// error: print an error message and die. From K&R, page 174.
void error(const char *fmt, ...)
{
  va_list args;
  va_start(args, fmt);
  vfprintf(stderr, fmt, args);
  fprintf(stderr, "\n");
  va_end(args);
  exit(EXIT_FAILURE);
}
```

Note que esta função vem no livro, mas não faz parte da biblioteca do C. Além disso, chamar-se `error` não significa que só deva ser aplicada em situações de erro. No nosso caso, o sistema ser impossível ou indeterminado não é um *erro*. É um caso normal que o nosso programa deve detetar e processar de acordo com que tiver sido especificado.

Além disso, esta função, tal como está, escreve a mensagem no `stderr`, e não no `stdout`. Por defeito, ambos estão associados à consola, e parece serem a mesma coisa, mas não são: por exemplo, se redirecionarmos o `stdout` para um ficheiro, o `stderr` fica na mesma.

Querendo usar isto para emitir uma mensagem e depois terminar, convém mudar de `stderr` para `stdout`, visto que não se trata propriamente de um erro.

Finalmente, aquele `EXIT_FAILURE` é uma constante simbólica cujo valor é transmitido ao sistema operativo por meio da função. Quando um programa termina normalmente, com êxito, o valor transmitido ao sistema operativo é zero. Conhecemos bem isso: é o `return 0;` que conclui a função `main`. Neste caso, o programa não termina na função `main`, mas sim através da função `exit`. Ainda assim, o valor transmitido deve ser o mesmo zero que do `return 0` da função `main`. Por outras palavras, em vez de `exit(FAILURE);` devemos ter `exit(0);`.

Nenhum dos casos de teste usados no problema D tinha sistemas impossíveis ou indeterminados.

Programação do método de Gauss, completo

Escreva uma função de teste que leia da consola um sistema de equações (como nas tarefas anteriores), resolva o sistema usando o método de Gauss na forma completa, e escreva a solução numa linha, como nos outros casos. No caso de sistema ser impossível ou indeterminado, o programa escreverá apenas uma linha com a mensagem `"System is impossible or indeterminate. Computation halted."` (sem as aspas).

Submeta da tarefa E.

Epílogo

Não se iluda. Observando os resultados escritos, e verificando à mão, até parece que o programa acertou em cheio na solução, em cada um dos casos. No entanto, se aplicarmos a função `is_solution` à solução *calculada* (e não apenas à solução *escrita*) teremos a surpresa de que, afinal, a solução calculada *não* é a solução do sistema de equações.

O que se passa é que, em geral, ainda que os coeficientes do sistema sejam todos inteiros (portanto, representáveis exatamente), os cálculos introduzem imprecisão e, por conseguinte, os resultados finais também são imprecisos. O que o nosso programa escreveu não foi a solução que calculou, mas sim essa solução arre-

dondada a um pequeno número de casas decimais. Esse arredondamento coincidiu com a solução exata, mas foi batota, por assim dizer.

Se em vez de escrever as soluções não com `doubles_println`, escrevermos com `doubles_printfln` usando a cadeia de formato `" %.17g"`, veremos a diferença. No meu caso, a solução do sistema de equações usado como exemplo no problema 2 veio assim:

```
1.9999999999999998 2.9999999999999996
```

e o do sistema no problema C veio assim:

```
1.0000000000000002 3 2
```