Passo a Passo
EP2 - MAP3121

Por: Um otário qualquer

Seguinte galera, sei que tá bem complicado esse fim de semestre EAD e parece que ninguém que dá aula pra gente tem noção alguma. Tive um tempinho livre então tentei elaborar uma explicação passo a passo com uns pseudocódigos que achei pra facilitar a vida de vocês, não sei quão útil vai ser, mas vamo lá. (Isso era pra ser umas mensagens no WhatsApp mas acabei me estendendo demais então virou esse Docs).

No EP1 a gente tinha as *condições iniciais* do nosso problema e queríamos iterá-lo para encontrar a solução em diferentes instantes de tempo até o tempo 1s.

Nesse EP, o que temos é a "solução" no instante final, só que a fonte são diversas **fontes pontuais**, iguais ao item C do EP1 mas em posições diferentes, e queremos entender como cada uma contribui para solução final.

$$f(t,x) = r(t)\sum_{k=1}^{nf} a_k g_h^k(x)$$

Assim, a função que representa as fontes será uma somatória de várias g_h em que p vale algo diferente em cada, e multiplicadas por intensidades a_k que fazem com que essa fonte seja mais ou menos relevante pra solução final. O problema nesse EP é ***descobrir quanto vale cada fonte.***

Tarefas

Tarefa A

Só que pra "simular" que vocês já sabem a solução final, ele pede pra primeiro você determinar a solução final conhecendo a resposta exata pros valores de a_k, usando o *método de Crank-Nicolson* do **EP1**, e com a solução final encontrada, fazer o processo inverso (que vou explicar melhor adiante).

E é isso que a **Tarefa A** pede, basicamente é só você pegar seu **EP1**, jogar tudo fora menos o *Crank-Nicolson* (e o método LDLt né, você pode usar o antigo pra essa parte ou o novo que vou falar mais adiante) e resolver ele pra cada uma das fontes pontuais que ele mencionar, depois somar tudo multiplicando pelas intensidades que ele definir.

a) Desenvolver um código que: dados os pontos $p_1, \cdots, p_{nf}$ gere os vetores $u_k(T, x_i), i = 1, ..., N-1$ a partir da integração da equação do calor (1)-(4), com condição inicial $u_0(x) = 0$ e de fronteira $g_1(t) = g_2(t) = 0$, com forçante $f(t,x) = r(t)g_h^k(x)$, $k = 1, \cdot, nf$. Para tanto você deve utilizar o método de Crank-Nicolson desenvolvido no EP1 (com $M = N$).

Tarefa B

A **Tarefa B** é a parte "importante" do EP, porque é a que realmente trata da matéria de Numérico. Nesse EP, o importante é resolver o ***problema inverso***. Para isso, temos o vetor que representa a solução final, que é u_T, e os vetores que representam a solução final de cada uma das fontes, $u_1, u_2, ..., u_k$. Uma vez que as fontes são uma **somatória** das funções g_h ponderadas pelas intensidades a_k, a gente sabe que a solução final u_T será uma **somatória** das soluções $u_1, u_2, ..., u_k$ de cada uma das fontes, também ponderadas pelas intensidades a_k:

$$u_T(x) = \sum_{k=1}^{nf} a_k u_k(T, x)$$

Entretanto, no problema inverso, a gente não conhece as intensidades a_k e são elas que queremos determinar. Como $u_1, u_2, ..., u_k$ são vetores com 129 pontos (para $N = 128$), na verdade a gente tem um **sistemaço** com 129 equações para determinar k incógnitas, que são as intensidades (e te garanto, não vão ter 129 fucking fontes).

Daí, temos um sistema com muito mais equações do que incógnitas, o que a gente chama de **sistema sobredeterminado**. E pra resolver esse tipo de problema, usamos o famoso, porém não tão amado, ***Método dos Mínimos Quadrados*** (também conhecido como ***Método da Melhor Aproximação*** dos resumos da Minerva de *Algelin 2*).

Tá, mas como eu faço isso? Temos a igualdade

$$u_T = a_1 u_1 + a_2 u_2 + a_3 u_3 + ... + a_k u_k$$

Para transformar esse problema em um sistema com *k* equações, para que possamos achar todas as *k* intensidades de forma ***única* e *determinada***, a gente faz o **produto interno** dos dois lados da igualdade por *cada um dos vetores* que estão do lado direito - o que vai manter a igualdade, caso vocês lembrem de *Algelin*. Caso não lembrem, bom, foda-se. Assim, o sistema fica com uma cara:

$$< u_T, u_1 >= a_1 < u_1, u_1 > + a_2 < u_2, u_1 > + a_3 < u_3, u_1 > + ... + a_k < u_k, u_1 >$$
$$< u_T, u_2 >= a_1 < u_1, u_2 > + a_2 < u_2, u_2 > + a_3 < u_3, u_2 > + ... + a_k < u_k, u_2 >$$

...

$$< u_T, u_k >= a_1 < u_1, u_k > + a_2 < u_2, u_k > + a_3 < u_3, u_k > + ... + a_k < u_k, u_k >$$

E por algum motivo bobo, isso ai chama Sistema Normal. Daí, se a gente montar uma matrizona com tudo isso pra poder resolver o sistema computacionalmente, a matriz que a gente chega é o que o enunciado fala:

$$\begin{bmatrix} < u_1, u_1 > & < u_2, u_1 > & \cdots & < u_{nf}, u_1 > \\ < u_1, u_2 > & < u_2, u_2 > & \cdots & < u_{nf}, u_2 > \\ \vdots & \vdots & \vdots & \vdots \\ < u_1, u_{nf} > & < u_2, u_{nf} > & \cdots & < u_{nf}, u_{nf} > \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{nf} \end{bmatrix} = \begin{bmatrix} < u_T, u_1 > \\ < u_T, u_2 > \\ \vdots \\ < u_T, u_{nf} > \end{bmatrix}$$

Então basicamente, montar essa matriz e o vetor do lado direito são a **Tarefa 2**:

> b) Dados os valores de $u_T(x_i), i = 1, ..., N-1$, monte a matriz e o sistema normal do problema de mínimos quadrados para o cálculo das intensidades.

Tem diversos jeitos de implementar o **produto interno** entre dois vetores - que é basicamente multiplicar eles entre si *coordenada por coordenada* e somar tudo pra chegar em um número pra por na matriz, então deixo pra criatividade de vocês, mas basicamente o código pra isso tem que ser tipo:

- Cria uma matriz $k \, x \, k$ e um vetor de tamanho k;
- Faz um *loop* pra rodar as linhas e um *loop* pra rodar as colunas da matriz e para cada posição *i,j* da matriz, calcula e armazena o produto interno entre u_{i+1} e u_{j+1} (porque né, a indexação começa em 0 e a contagem do EP começa em 1, mas ignorem esses índices se isso mais confundir do que ajudar).
- Ao final (ou no começo, segue seu coração total) do *loop* mais externo, faz o produto interno entre o valor de u_{i+1} e a solução u_T e armazena na posição i do vetor das soluções.

Tarefa C

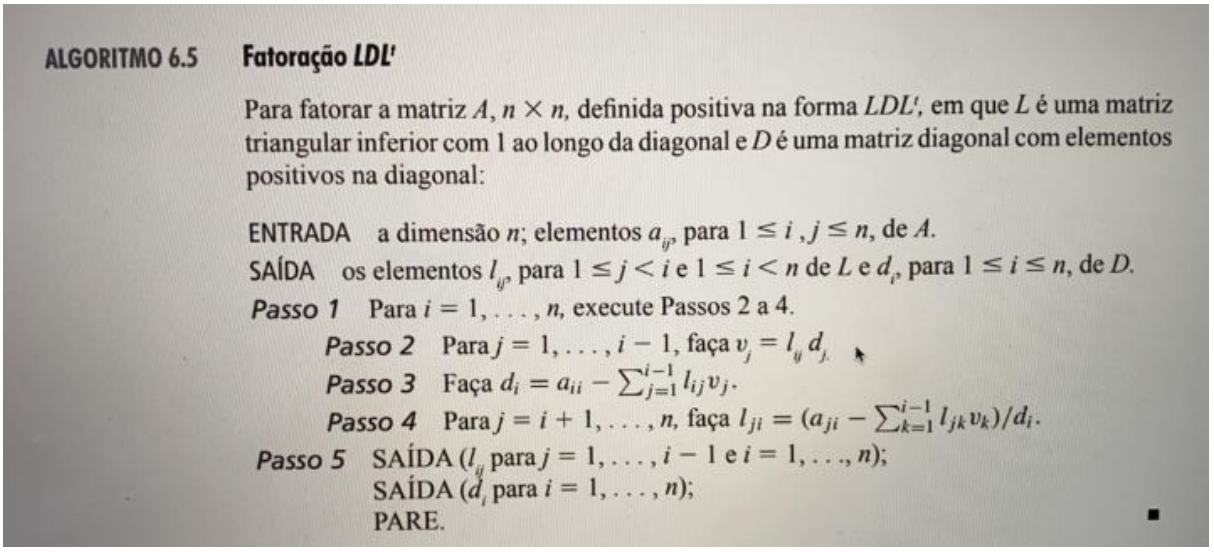
Bom, a **Tarefa C** é, agora que você tem o sistema montadinho, resolver ele. Pra resolver ele, a gente tem que usar o mesmo método da decomposição LDL^t que usamos no **EP1**, com uma diferença: a matriz não é esparsa. E o que significa não ser esparso? Significa que nada vai ser zero e isso é triste, porque seu PC vai chorar resolvendo esses sistemas. Mas tudo bem.

Então, vocês vão ter que fazer novos códigos pra fatoração LDL (mas não vai mudar muito) e pra resolução do sistema resultante (o que também não vai mudar muito). Vocês podem usar tanto esses novos códigos quanto os códigos do **EP1** já

construídos para resolver o *Crank-Nicolson* na Tarefa 1, mas tratar as coisas como vetores, igual o **EP1** pedia, é mais amigável com seu PC. Mas de novo, segue seu coração.

No livro da matéria tem pseudocódigos pra calcular as matrizes *L* e *D* e pra resolver o sistema resultante pra matrizes cheias, que estão logo abaixo:

Fatoração LDL



ALGORITMO 6.5 **Fatoração LDL^t**

Para fatorar a matriz A, $n \times n$, definida positiva na forma LDL^t, em que L é uma matriz triangular inferior com 1 ao longo da diagonal e D é uma matriz diagonal com elementos positivos na diagonal:

ENTRADA a dimensão n; elementos a_{ij}, para $1 \le i, j \le n$, de A.

SAÍDA os elementos l_{ij}, para $1 \le j < i$ e $1 \le i < n$ de L e d_i, para $1 \le i \le n$, de D.

Passo 1 Para $i = 1, \ldots, n$, execute Passos 2 a 4.

Passo 2 Para $j = 1, \ldots, i-1$, faça $v_j = l_{ij} d_j$.

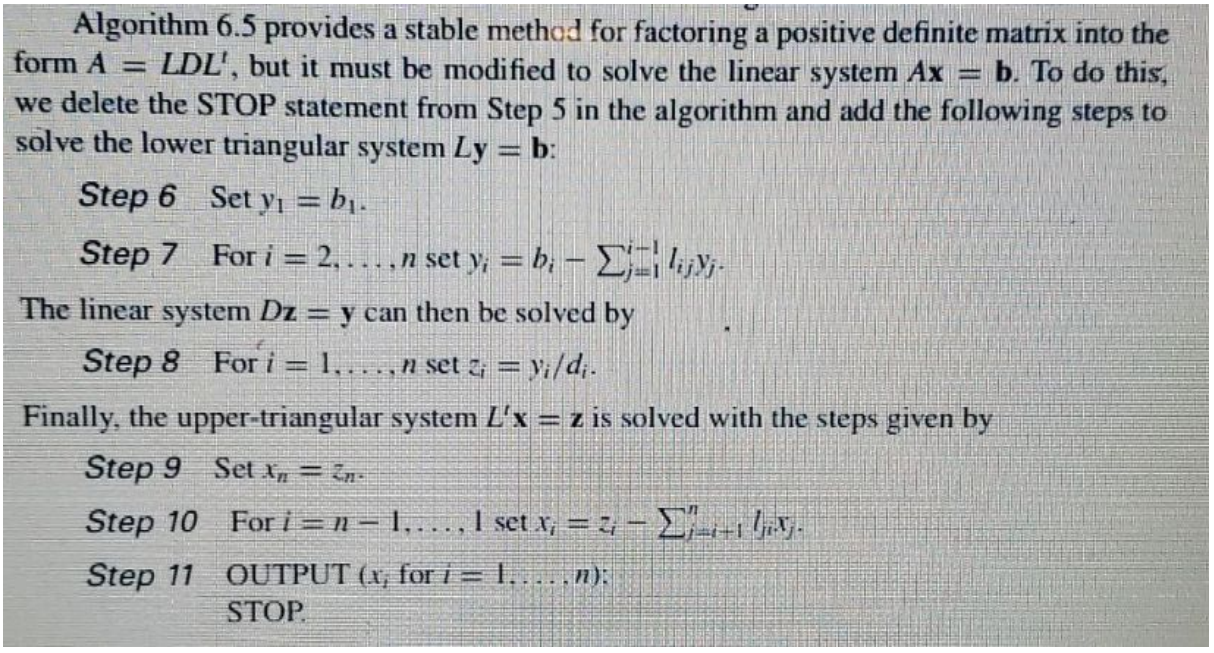
Passo 3 Faça $d_i = a_{ii} - \sum_{j=1}^{i-1} l_{ij} v_j$.

Passo 4 Para $j = i+1, \ldots, n$, faça $l_{ji} = (a_{ji} - \sum_{k=1}^{i-1} l_{jk} v_k)/d_i$.

Passo 5 SAÍDA (l_{ij} para $j = 1, \ldots, i-1$ e $i = 1, \ldots, n$);
SAÍDA (d_i para $i = 1, \ldots, n$);
PARE. ■

Resolução do sistema

(o print que tenho é em Inglês, se alguém tiver em português eu mudo)



Algorithm 6.5 provides a stable method for factoring a positive definite matrix into the form $A = LDL^t$, but it must be modified to solve the linear system $A\mathbf{x} = \mathbf{b}$. To do this, we delete the STOP statement from Step 5 in the algorithm and add the following steps to solve the lower triangular system $L\mathbf{y} = \mathbf{b}$:

Step 6 Set $y_1 = b_1$.

Step 7 For $i = 2, \ldots, n$ set $y_i = b_i - \sum_{j=1}^{i-1} l_{ij} y_j$.

The linear system $D\mathbf{z} = \mathbf{y}$ can then be solved by

Step 8 For $i = 1, \ldots, n$ set $z_i = y_i / d_i$.

Finally, the upper-triangular system $L^t\mathbf{x} = \mathbf{z}$ is solved with the steps given by

Step 9 Set $x_n = z_n$.

Step 10 For $i = n-1, \ldots, 1$ set $x_i = z_i - \sum_{j=i+1}^{n} l_{ji} x_j$.

Step 11 OUTPUT (x_i for $i = 1, \ldots, n$);
STOP.

Só precisa prestar atenção, de novo, com os índices usados no livro e os índices das suas matrizes no código. Caso vocês não acreditem no pseudocódigo acima e queiram deduzir, a vontade, acho válido. No fim do arquivo, vou deixar

minha dedução de *L* e *D* também, pra caso ajude algum de vocês (**reparem que não sigo os mesmos índices do livro, então não vai estar exatamente a mesma fórmula e nem vai estar igual ao que provavelmente será seu código**).

Enfim, daí com isso implementado, é só mandar ele resolver o sistema normal que você montou na **Tarefa B** e brasil.

Erro quadrático

Uma coisa que ele pede pra fazer mas não especifica exatamente se se encaixa em alguma tarefa ou não, então pus aqui, é o **erro quadrático**. É basicamente o erro da aproximação final que você chega com relação ao valor exato:

$$E_2 = \sqrt{\Delta x \sum_{i=1}^{N-1} \left(u_T(x_i) - \sum_{k=1}^{nf} a_k u_k(T, x_i) \right)^2} \ ;$$

Bom, o $\Delta x = \frac{1}{N}$, então isso vai ser suave de achar. No fim de calcular as aproximações - dos testes *c* e *d*, no caso - para as intensidades a_k, os passos a serem executados são:

- Multiplicar cada vetor u_k por sua respectiva intensidade a_k;
- Somar todas essas multiplicações;
- Fazer a subtração, elemento a elemento, de u_T pela soma dos $a_k u_k$;
- Elevar cada um dos resultados acima ao quadrado, somar tudo, e multiplicar por Δx;
- Fazer raiz disso e partir pro abraço.

Testes

Daí, o EP pede pra fazer alguns testes né, pra provar que tá tudo certinho. Lembrem de ler as **observações finais** porque, por ALGUM MOTIVO o professor gosta de pôr as coisas importantes no fim. O programa tem que permitir que usuário **escolha** qual dos testes quer rodar e, nos testes *c* e *d*, o usuário tem que escolher também qual o *N* escolhido. E tem que plotar gráficos nesses dois testes também (tanto com os valores do arquivo - que vou falar mais pra frente - quanto com o que você achou pelo código) - mas se quiserem plotar gráfico pros 4 itens, deixa mais visual o que está acontecendo, mas bolhufas. **IMPORTANTE: Mudou a $r(t)$ agora, então mudem isso nos seus EP1 que vão reaproveitar.**

Em todos os testes utilizaremos $T = 1$ e $r(t) = 10(1 + \cos(5t))$.

Teste A

a) Para os parâmetros $N = 128$, $nf = 1$ e $p_1 = 0.35$ você deve fazer uma primeira verificação do seu programa. Construa $u_1(T, x_i), i = 1, ..., N-1$. Defina $u_T(x_i) = 7u_1(T, x_i)$ e resolva o problema inverso. Neste caso, o sistema linear é 1×1, com solução trivial e você deve obter $a_1 = 7$.

No teste A, é só uma validação de tudo que você implementou. Em primeiro lugar, você vai simplesmente usar a parte que você só deu Ctrl C-Ctrl V do seu **EP1** e resolver o item C dele de novo, mas com $p = 0.35$. Depois, com esse valor achado, você faz isso vezes 7, define que é a solução final e "finge" que não sabe que a intensidade é 7. Aí você usa a **Tarefa B** pra montar o sistema (que vai ser 1x1, só vai ter um número), e usa a **Tarefa C** pra resolver ele - ou não, porque é só 1x1, então se quiser pode só dividir um número pelo outro e acabou.

Teste B

b) Como segundo teste de verificação, ainda com $N = 128$, tomemos $nf = 4$ e $p_1 = 0.15$, $p_2 = 0.3$, $p_3 = 0.7$ e $p_4 = 0.8$. Construa as funções $u_k(T, x)$ e defina $u_T(x_i) = 2.3u_1(T, x_i) + 3.7u_2(T, x_i) + 0.3u_3(T, x_i) + 4.2u_4(T, x_i)$ e resolva o problema inverso, testando também seu método que resolve o sistema linear. Os coeficientes representando as intensidades das fontes devem ser recuperados.

No teste B, ainda é uma validação só do que você implementou, mas aumenta a dimensão do sistema. De novo, agora tem 4 fontes, então você vai ter que resolver o **EP1** 4 vezes, pra cada um dos valores de p dados, e armazenar as soluções em $T = 1$. Depois, definir a solução final u_T como ele define aí, e resolver o problema inverso, usando as **Tarefas B e C** pra isso, e terá de achar as mesmas intensidades usadas pra definir u_T (no caso, 2.3, 3.7, 0.3 e 4.2).

Teste C

c) No arquivo teste.txt fornecido são dadas as localizações de 10 fontes e a seguir os valores de $u_T(x)$ em uma malha com $\Delta x = 1/2048$. Estão fornecidos os valores de $u_T(x_i), i = 0, \cdots, 2048$, incluindo os extremos do intervalo. Com estes dados você deve rodar 5 testes, para os valores de $N = 128, 256, 512, 1024$ e 2048. Para os testes com N menor que 2048, você deve utilizar os valores do arquivo nos pontos de malha adequados. Por exemplo, se $N = 512$, os pontos seriam $x_0, x_4, x_8, \cdots, x_{2048}$, ou seja, se tomam os valores do arquivo de 4 em 4. Os valores nos extremos, que correspondem às fronteiras ainda serão descartados, para se ficar com $N - 1 = 511$ valores de $u_T(x)$ nesta malha. (O procedimento para os outros valores de N é análogo). Para cada N, você deve imprimir além dos valores das intensidades a_k, o valor do erro quadrático E_2, conforme (39).

Esse teste e o próximo já são um pouco mais complicados. Vocês vão ter que ler um arquivo *.txt*, o que talvez não tenham feito nunca (não sei se tinha no **EP3** de *MAC2166* de 2018), mas se tiver algum amigo de 2017, ele com certeza já teve que implementar isso e teria um código pronto, mas dá pra achar meio fácil na internet também. Dele você vai extrair a localização das 10 fontes - e para cada uma delas, resolver o *Crank-Nicolson* do **EP1** para achar os valores de $u_1, u_2, ..., u_{10}$ com o valor de N fornecido pelo usuário.

Do arquivo, você também vai ter que extrair a solução final u_T para a qual resolveremos o problema inverso. Depois de extrair, você deve deixar o usuário

escolher o valor de *N* e pega os valores de u_T com os índices $m\frac{2048}{N}$, onde $m = 0, 1, 2..., N$.

Depois, você **joga fora** o primeiro e o último ponto de u_T - porque são as bordas, e você já conhece elas - e dos vetores $u_1, u_2, ..., u_{10}$ (caso você tenha salvado eles com as condições de contorno - ou seja, se esses vetores tiverem tamanho *N+1*, joga as duas bordas fora), e monta o sistema normal com a **Tarefa B** e esses vetores. Daí, usa a **Tarefa C** pra resolver o sistema.

Nessa parte, é interessante rodar os testes pra **todos** os *N* e analisar como o erro quadrático E_2 mencionado muda, e comparar os gráficos também. Pelo menos são menos gráficos que no **EP1**, galera.

Teste D

d) Caso com ruído: Você deve agora repetir os testes do item c), mas com a introdução de ruído nos dados, representando erros de medição na temperatura final. Para tal, multiplique cada valor de $u_T(x_i)$ por $1. + r\epsilon$ com $\epsilon = 0.01$ e r um número randômico entre -1 e 1.

Observação: Em python a função random() (do módulo random) produz a cada chamada um valor (pseudo) aleatório entre 0 e 1. Subtraindo 0.5 e multiplicando por 2, você obterá números entre -1 e 1.

Esse teste é basicamente igual ao *c*, porém agora você vai pegar o seu vetor u_T e multiplicar **cada ponto** por $1 + r\varepsilon$, sendo o valor de r é um número randômico entre -1 e 1. O resto é exatamente igual ao item *c*, a única diferença é que o erro quadrático deve ser maior (porque você está considerando a presença de ruído no sistema, o que vai arapalhar a precisão dos valores).

Anexos

Respostas Encontradas - Teste C

*Não é 100% de certeza dessas respostas, mas bateram com algumas pessoas que já fizeram o EP e parecem fazer sentido.

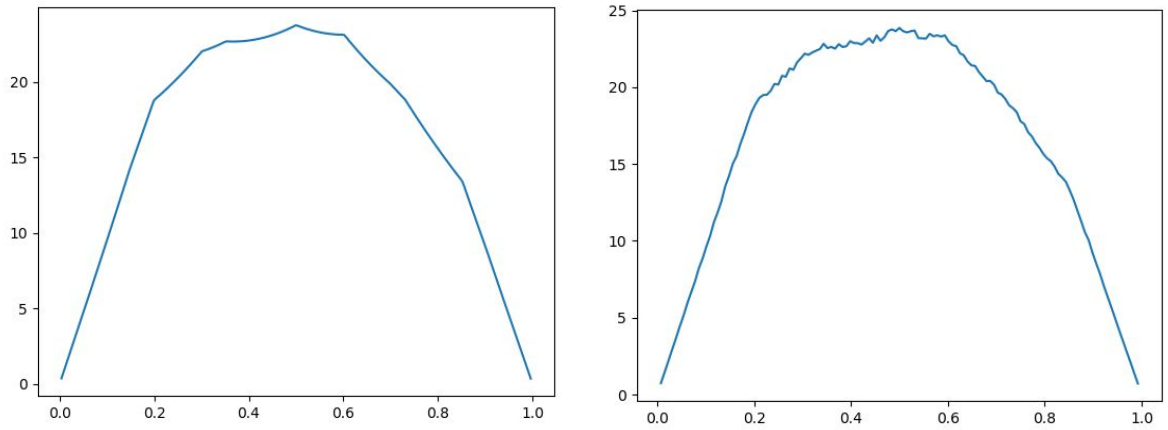
```
Para um N= 128 as fontes encontradas foram:
a0= 1.2091231792052213
a1= 4.8392587157453875
a2= 1.8872408557581473
a3= 1.5833999318631413
a4= 2.214504046288475
a5= 3.1212947787761767
a6= 0.37734028637353134
a7= 1.4923482881211108
a8= 3.9751388015994467
a9= 0.4041451536482194
E o erro quadrático para esse N foi  0.024453403799692287
```

```
Para um N= 256 as fontes encontradas foram:
a0= 0.9045010343180415
a1= 5.077572635561019
a2= 2.100853595478437
a3= 1.414155685089053
a4= 2.229245013053225
a5= 3.1046138569913344
a6= 0.5094525973917108
a7= 1.3865087904572508
a8= 3.9498786461527127
a9= 0.4148931283304515
E o erro quadrático para esse N foi  0.012363464048877426

Para um N= 512 as fontes encontradas foram:
a0= 0.9286883784929074
a1= 5.053707844481231
a2= 2.043701048902804
a3= 1.4676706728645108
a4= 2.1967633319997475
a5= 3.0911311689006373
a6= 0.6375875163805107
a7= 1.2716872153224923
a8= 3.8780948673279467
a9= 0.5305567786436954
E o erro quadrático para esse N foi  0.008476628330811815
```

```
Para um N= 1024 as fontes encontradas foram:
a0= 1.0072813220760786
a1= 4.992443012453549
a2= 1.9858767276154232
a3= 1.5132584652244283
a4= 2.1926928376835892
a5= 3.0951528759327376
a6= 0.6523266477839922
a7= 1.2537898890586359
a8= 3.8796670569413596
a9= 0.5297366253015315
E o erro quadrático para esse N foi  0.0037793104633012057

Para um N= 2048 as fontes encontradas foram:
a0= 0.9999999999943014
a1= 5.000000000014566
a2= 2.0000000000134825
a3= 1.500000000049809
a4= 2.2000000000525954
a5= 3.100000000035437
a6= 0.600000000018869
a7= 1.3000000000243883
a8= 3.9000000000233217
a9= 0.5000000000067633
E o erro quadrático para esse N foi  2.5499517755874645e-12
```

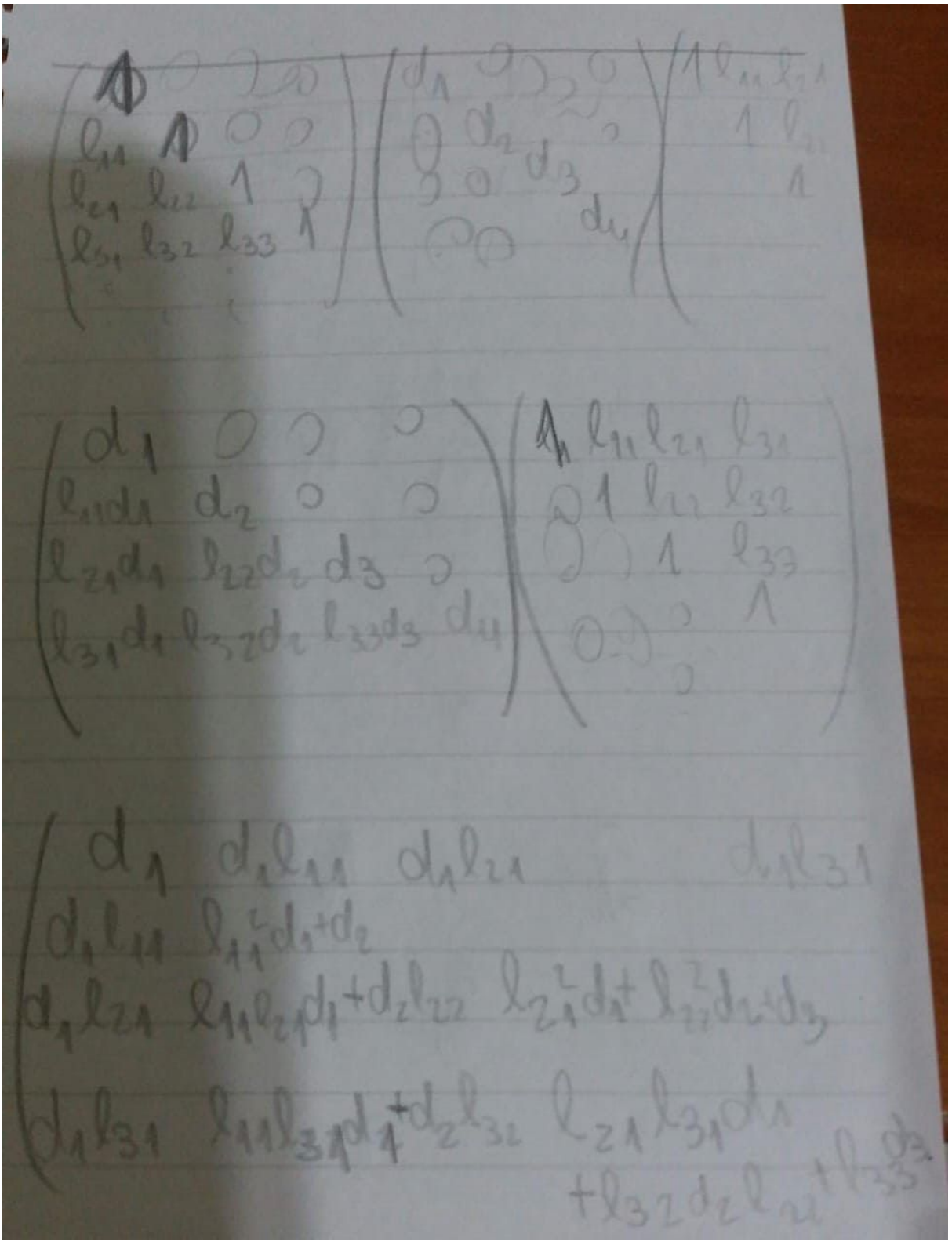
*Para N=2048, é natural não dar exatamente a mesma coisa, porque terão erros de aproximação dependendo de como se multiplica as matrizes - provavelmente vão achar menores ainda que esse, porque *numpy* aumenta os erros - pros outros também podem ter pequenas diferenças.

Soluções do teste C e D para N=128



*O gráfico do C é pra sair bem parecido com o gabarito mesmo, porque não tem ruído, e do D pode sair um pouco diferente desse também, porque usamos números aleatórios. Podem plotar a exata em cima da aproximação pra comparar e plotar cada um dos $u_1, u_2, ..., u_k$ obtidos por *Crank-Nicolson* se quiserem também.

Dedução de expressões para *L* e *D*



$$d_1 = a_{11}$$

$$l_{11} = \frac{a_{21}}{d_1}$$

$$d_2 = a_{22} - l_{11}^2 d_1$$

$$l_{21} = \frac{a_{31}}{d_1} \qquad l_{22} = \frac{a_{32} - l_{11} l_{21} d_1}{d_2}$$

$$d_3 = a_{33} - l_{21}^2 d_1 - l_{22}^2 d_2$$

$$l_{31} = \frac{a_{41}}{d_1} \qquad l_{32} = \frac{a_{42} - l_{11} l_{31} d_1}{d_2}$$

$$l_{33} = \frac{a_{43} - l_{21} l_{31} d_1 - l_{32} l_{22} d_2}{d_3}$$

$$d_4 = a_{44} - l_{31}^2 d_1 - l_{32}^2 d_2 - l_{33}^2 d_3$$

$$d_i = a_{i,i} - \sum_{j=1}^{i-1} l_{i-1,j}^2 \, d_j$$

$$l_{i,j} = \frac{a_{i+1,j} - \sum_{k=1}^{i-1} l_{j,k} \, d_k \, l_{i,k}}{d_j}$$