

UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

RODRIGO DE CASTRO MICHELASSI

NUSP: 13672703

Exercício-Programa 2: Interpolação em imagens

São Paulo

2023

Sumário

1	Introdução	2
2	Detalhes da Implementação	3
3	Algoritmos	4
3.1	<i> Compress</i>	4
3.2	<i> Decompress - Bilinear</i>	4
3.3	<i> Decompress - Bicubic</i>	4
3.4	<i> Calculate Error</i>	4
3.5	<i> Generate Image</i>	5
4	Testes e eficiência	6
4.1	<i> Imagens externas</i>	6
4.2	<i> Imagens geradas</i>	6
4.3	<i> Compressão</i>	7
4.4	<i> Descompressão Bilinear e Erro</i>	8
4.5	<i> Descompressão Bicúbica e Erro</i>	9
5	Perguntas pedidas	11
6	Considerações Finais	13

1 Introdução

Esse EP tem como objetivo mostrar como funciona o processo de interpolação polinomial na prática e usos possíveis dentro da área de computação, além de mostrar, na prática, o que ocorre quando interpolamos um polinômio, e o quanto eficiente são os resultados obtidos.

Dito isso, o EP consiste de três processos distintos a serem realizados, a compressão de imagens, a descompressão e o cálculo do erro da aproximação do polinômio, que deve ser analisada comparando a imagem original com a imagem obtida pós compressão seguida de descompressão.

Nesse relatório, será possível analisar alguns dos resultados obtidos, exemplos, análise de performance etc.

2 Detalhes da Implementação

Antes de partirmos para falar sobre o programa em si, vamos falar um pouco sobre como ele pode ser testado.

O arquivo enviado contará com 4 arquivos de extensão *.m*, os quais possuem os códigos do programa. Além disso, algumas imagens de exemplos também foram enviadas. Para testar o programa, será necessário possuir o software OCTAVE instalado em sua máquina. Ele pode ser obtido através do comando:

```
$ apt install octave
```

Note que o programa enviado não possui um arquivo main, que consideraria cada função separadamente, logo podemos executar as funções desejadas separadamente. Para isso, vá até o diretório desejado e rode o comando:

```
$ octave --gui
```

Com a interface gráfica do octave aberta, podemos executar nossas funções. Segue um exemplo de entrada para cada uma delas:

Compress: compress("bobEsponja.webp", 3), onde "bobEsponja.webp" deve ser digitado com aspas e se refere à imagem, e 3 é a taxa de compressão. A saída será uma imagem chamada "compressed.png", salva no mesmo diretório.

Decompress: decompress("compressed.png", 1, 3, 5), onde "compressed.png" deve ser digitado com aspas e se refere à imagem anteriormente comprimida, 1 é o método de descompressão, que pode ser 1 ou 2, 3 é a taxa de descompressão e 5 é um valor para h, a distância entre pontos. Pode haver duas saídas diferentes, para cada método testado, são elas "bicubicdec.png" e "bilineardec.png".

Calculate Error: calculateError("bobEsponja.webp", "bilineardec.png"), onde bobEsponja se refere a imagem original, e bilineardec se refere a mesma imagem, pós compressão e descompressão. Essa função deve funcionar apenas para casos em que são utilizadas a mesma imagem para comparação, e a saída será o valor do erro, impresso no próprio GNU Octave.

Para o arquivo de gerar imagens, basta enviar a dimensão desejada.

3 Algoritmos

3.1 Compress

O algoritmo de compressão é simples e funciona com base nos dados do enunciado. Para isso, rodamos com dois loops encaixados todos os pontos da imagem de entrada, e, verificamos se, no nosso loop, o resto da divisão entre o valor do iterador e $k + 1$ é zero. Caso seja, salvamos esse ponto na nossa nova imagem. Os testes de eficiência e resultados podem ser encontrados na próxima seção.

3.2 Decompress - Bilinear

Esse é o primeiro algoritmo de descompressão implementado, referente ao código 1 no nosso programa. Para esse algoritmo, começamos com uma ampliação da imagem comprimida, com diversos pontos escuros, e a nossa ideia é interpolar esses pontos para tentar reconstruir a imagem. Note que o resultado final não terá a melhor resolução possível, pois a imagem comprimida perde muita resolução. A ideia desse algoritmo consiste em resolver o sistema linear proposto no intervalo para definir o polinômio interpolante e preencher a imagem ampliada conforme encontramos pontos escuros nela.

3.3 Decompress - Bicubic

Esse algoritmo funciona de maneira mais complexa com o anterior. Note que na imagem gerada aqui haverá um erro maior, que pode ser perceptível em algumas imagens, onde no canto direito haverá uma falha, uma linha em preto que não será preenchida. Da mesma forma do anterior, a imagem será ampliada e preenchida conforme a implementação, e resolveremos também o sistema linear proposto no enunciado, onde a matemática por trás do problema é explicada com mais clareza.

3.4 Calculate Error

Esse algoritmo consiste no cálculo da forma exposta no enunciado. Utilizamos dois laços encaixados para calcular o erro em cada camada rgb da imagem, e ao fim dividimos

essa soma pelo quadrado de suas dimensões, e extraímos sua raiz quadrada. O erro será a média do erro nas três cores. Note que aqui ocorre um problema quanto à dimensões das imagens, visto que após comprimir e descomprimir as imagens, ela perde uma quantidade de pixels relativa à taxa de compressão, logo a imagem original será comparada apenas até o tamanho da imagem descomprimida. Os resultados obtidos podem ser vistos a seguir.

3.5 Generate Image

Esse algoritmo recebe as dimensões da imagem quadrada a ser gerada e calcula, em cada uma das cores, o valor correspondente, para cada ponto, da função escolhida. Para os testes, serão escolhidas algumas funções diferentes expostas a seguir, e será entregue o programa conforme a função definida no enunciado.

4 Testes e eficiência

Nessa seção, estaremos apresentando testes de compressão, descompressão e cálculo do erro, utilizando todos os métodos implementados no algoritmo. Além disso, serão realizados testes com imagens em diferentes escalas, e taxas de compressão variadas. Note que, utilizaremos imagens em diferentes formatos, sendo que os formatos .jpeg, .webp e .png foram testados e ambos são lidos corretamente pelo programa. As imagens de saída são geradas sempre com a extensão .png.

4.1 Imagens externas

Para o programa, foram escolhidas as seguintes imagens externas para teste:



As dimensões delas são, respectivamente: 150x150, 984x984, 1000x1000. Note que as imagens precisam ser quadradas, com proporção 1x1.

4.2 Imagens geradas

Nessa seção, iremos gerar 3 imagens com dimensões 300x300, utilizando funções matemáticas pré-definidas. As 3 imagens geradas também serão enviadas junto ao arquivo de entrega do EP. Note que a nossa função, será da forma

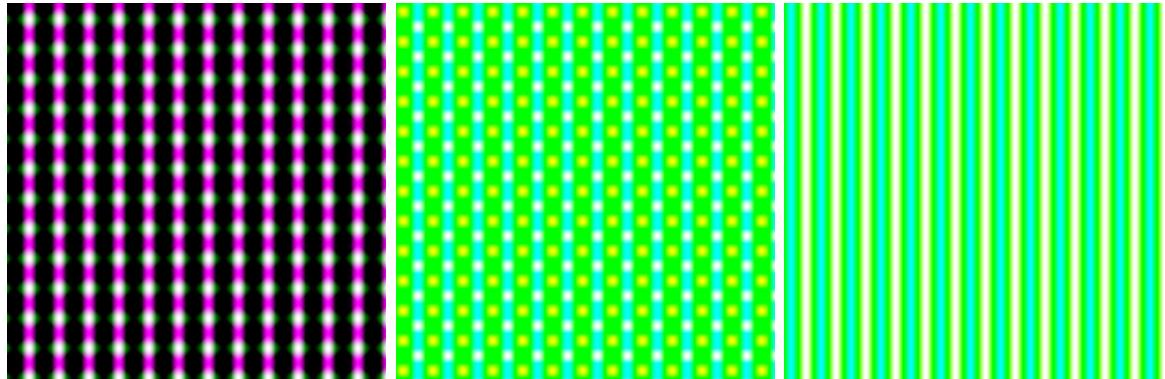
$$f : \mathbb{R}^2 \rightarrow \mathbb{R}^3,$$

ou seja, receberá 2 parâmetros x e y, e gerará 3 imagens, correspondentes a cada uma das cores RGB possíveis. As funções escolhidas foram:

$$\bullet f(x, y) = (\operatorname{sen}x, \frac{\operatorname{sen}y + \operatorname{sen}x}{2}, \operatorname{sen}x);$$

- $f(x, y) = (\operatorname{sen}x, \frac{\operatorname{tgy} + \operatorname{sen}hx}{2}, \operatorname{sen}^2x);$
- $f(x, y) = (\operatorname{sen}x * \operatorname{tanh}(x) * \cos(y), \frac{1/\cos(x)*\operatorname{sen}(xy)+\operatorname{senh}(x)}{2}, \operatorname{sen}x).$

Com essas funções, geramos, respectivamente, as imagens:



4.3 Compressão

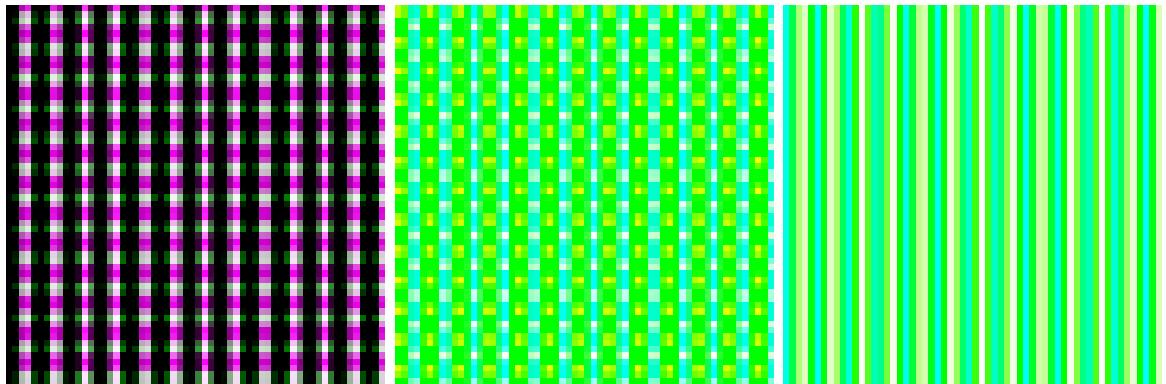
Para a compressão, utilizamos uma taxa $k = 4$ para cada uma das imagens mencionadas acima, e obtivemos as imagens:



O tempo de geração dessas imagens foi, respectivamente em segundos: 0.031884, 0.751479, 0.630504. Note que, apesar de a segunda imagem ter uma menor densidade de pixels que a terceira, o nível de detalhamento nela é muito maior, então sua compressão levou mais tempo.

Além disso, na primeira imagem, que era menor e possuía uma baixa densidade de pixels, uma boa parte da qualidade da imagem foi perdida durante a compressão.

Agora, para as imagens geradas pelo programa, obtivemos os seguintes resultados:



Os tempos de execução para comprimir cada uma das imagens foram: 0.0642238, 0.0680861, 0.0557029.

4.4 Descompressão Bilinear e Erro

Para essa descompressão, estaremos utilizando uma taxa $k = 4$, um valor de $h = 2$, e o método 1, equivalente à descompressão bilinear. Segue os resultados obtidos:

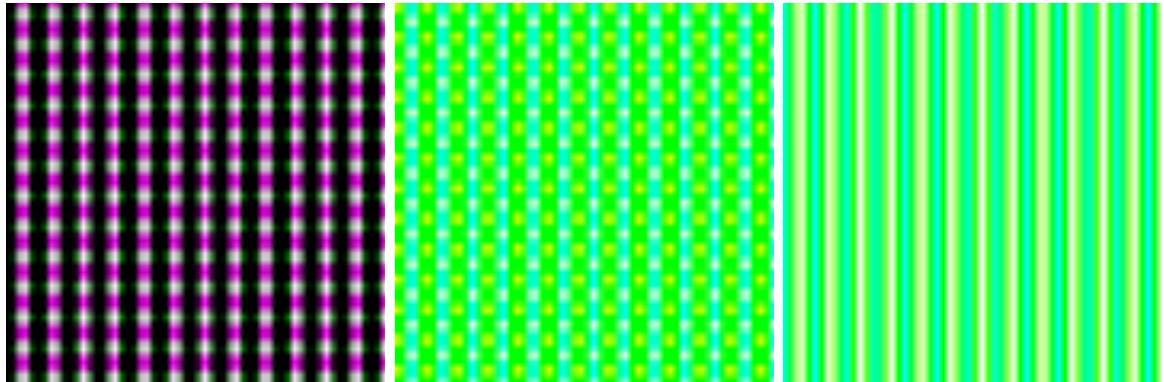


O tempo para a geração dessas imagens, em segundos, foi respectivamente: 0.800506, 36.0459, 36.5976. Note que agora tivemos um tempo de interpolação próximo entre a segunda e terceira imagem. Embora também tenhamos recuperado um pouco da qualidade na primeira imagem, ela continua fortemente borrada e mal definida, devido à qualidade perdida na compressão. As outras imagens não perderam tanta qualidade no processo.

O erro obtido entre as imagens originais e as interpoladas foram, respectivamente: 0.2752, 0.1714, 0.085542. Nota-se que as taxas de erro foram maiores na primeira imagem,

a qual tinha densidade de pixels menor e visualmente perdeu maior qualidade após a compressão.

Para as imagens geradas pelo programa descomprimidas, obtivemos as seguintes imagens:



Além disso, obtivemos os seguintes tempos de execução, em segundos, para cada uma delas, respectivamente: 3.4105, 3.3867, 3.26183.

Por fim, os erros obtidos para cada imagem foram, respectivamente: 0.3632, 0.2471, 0.2995

Podemos observar aqui que os tempos de execução se mantiveram constantes para essas três imagens, com variação muito baixa, o que nos leva a acreditar que o programa funciona de maneira estável para imagens com a mesma dimensão, e certos pixels em cada imagem podem modificar seu desempenho. O erro foi relativamente alto.

4.5 Descompressão Bicúbica e Erro

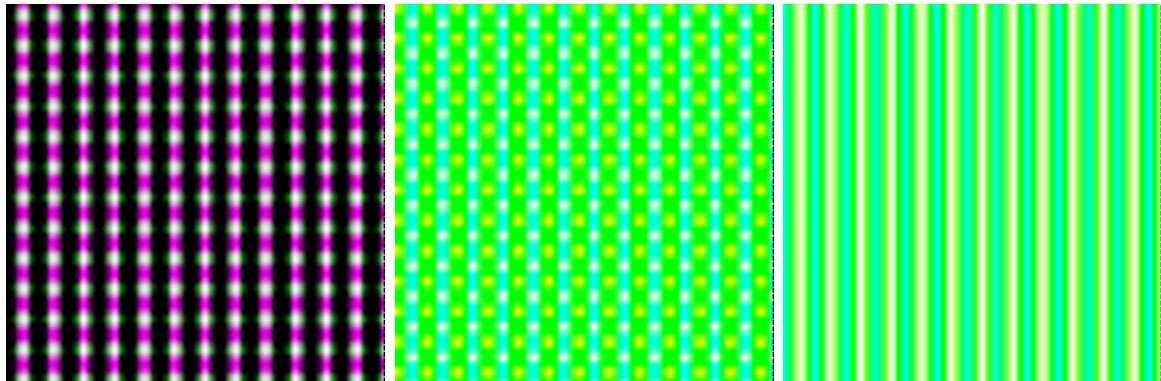
Continuaremos utilizando os mesmos valores de referência da descompressão anterior, porém agora utilizaremos um outro método. Note, pelos motivos citados na introdução, pode ser que os erros sejam maiores agora. Obtivemos as seguintes imagens:



O tempo para a geração dessas imagens foi, respectivamente, em segundos: 1.68988, 75.5371, 77.4273

Os erros obtidos para cada imagem, respectivamente, foi: 0.3044, 0.1803, 0.098207. Assim como era esperado, o valor dos erros foi ainda maior com esse método, e o tempo de execução também foi maior.

Para as imagens geradas pelo programa descomprimidas, obtivemos as seguintes imagens:



Além disso, obtivemos os seguintes tempos de execução, em segundos, para cada uma delas, respectivamente: 7.01679, 7.43408, 6.83535.

Por fim, os erros obtidos para cada imagem foram, respectivamente: 0.3779, 0.2838, 0.3396.

Novamente, podemos analisar aqui um erro maior quanto à interpolação bicúbica, além do tempo de execução ser quase o dobro.

5 Perguntas pedidas

Para responder essas perguntas, utilizamos agora uma imagem em preto e branco, com dimensões 450x450. A seguir, podemos ver essa imagem original, sua versão comprimida em uma taxa $k=7$ e sua versão descomprimida, na mesma taxa, pelos dois métodos testados:



Para essas imagens, obtivemos erros 0.1551 e 0.1643 para as interpolações bilinear e bicúbica, respectivamente.

Observando esses resultados obtidos, podemos responder algumas das perguntas pedidas, com respeito aos testes das imagens geradas e externas:

1. Funciona bem para imagens preto e branco? Sim, podemos comparar com os resultados obtidos acima que, para essa imagem em preto e branco, mesmo com uma taxa de compressão mais alta, a definição da imagem é maior. Por questões de dificuldade em gerar imagens em preto e branco, estaremos utilizando para esse teste apenas essa imagem externa.

2. Funciona bem para imagens coloridas? Sim, considerando os resultados obtidos na seção anterior, podemos ver um erro relativamente baixo para imagens coloridas, em especial aquelas mais simples ou com maior resolução.

3. Funciona bem para todas funções de classe C^2 ? Pelos testes feitos acima, o programa aparenta funcionar bem para funções de classe C^2 , pois o erro se manteve baixo.

4. E para funções que não são de classe C^2 ? É fácil de ver que o programa funcionará para funções de classe acima de C^2 , por continuidade, porém para funções de classe C^1 ocorre uma maior limitação na geração de imagens, que depende de valores de X e Y para as funções.

5. Como o valor de h muda a interpolação? O valor de h define, na nossa interpolação, a distância considerada entre os pontos. Sendo assim, esse valor nos ajuda

na definição dos valores iniciais do nosso polinômio interpolante e, quanto maior o valor de h , um pouco mais preciso será nossa interpolação também. Todavia, não foi possível verificar isso nos testes.

6. Como se comporta o erro? O erro se comporta com valores entre 0.08-0.50, em média, sendo mais baixo para figuras com taxa de compressão menor e para figuras com menos detalhes, isso é, uma quantidade maior de pixels repetidos, concentrada na mesma porção da imagem.

7. Teste com $k=7$ e a descompressão com $k=1$ 3 vezes. Ao aplicarmos uma compressão com $K = 7$ e a descompressão com $K = 7$ também, obtemos o resultado mostrado nas figuras acima. Todavia, ao aplicarmos a descompressão com $k = 1$, durante 3 vezes diferentes, obtivemos uma figura com as mesmas dimensões (449x449, para a imagem testada) e o mesmo erro, 0.1551. Todavia, o que foi percebido é que, mesmo que esse resultado seja curioso e impressionante, o tempo de execução para comprimir a mesma imagem 3 vezes, com $K = 1$ acaba sendo maior do que comprimir apenas uma vez com $K = 7$.

6 Considerações Finais

Pelos resultados e análises feitas acima, podemos ver que a interpolação bicúbica gasta um tempo de execução e possui uma complexidade computacional muito maior que a interpolação bilinear. Nesse sentido, ela também apresentou na nossa implementação um erro um pouco maior, porém isso provavelmente ocorre devido ao fato de o algoritmo não interpolar bem os pontos à direita da imagem e no canto inferior.

Com base nisso, chegamos a conclusão que, para as imagens apresentadas e o algoritmo entregue, a interpolação bilinear parece valer mais a pena, e valeria realizar testes com um algoritmo totalmente correto da interpolação bicúbica, para tomar uma conclusão mais precisa sobre os experimentos. Todavia, a interpolação bilinear faz um bom trabalho, com pequenas taxas de erro e relativamente rápido.

Além disso, foi possível analisar um pouco mais a fundo os algoritmos e perceber a forma como ele age para interpolar pontos, além de notar que a forma que os pixels são levados em consideração pelo programa influenciam o desempenho do processo, visto que, para imagens como *bobEsponja.webp* o tempo de compressão e descompressão foi menor que para imagens como *taylor.webp*, devido ao fundo branco da imagem.