

Universidade de São Paulo
Instituto de Matemática e Estatística
MAC0210

Relatório: Exercício-Programa 2

Alunos:
Gustavo Silva No USP: 9298260
Leonardo Padilha No USP: 9298295

Professor: Ernesto G. Birgin

Índice

1	Objetivo	2
2	Interpolação Bilinear por partes	2
2.1	Resultados no Zoológico	4
2.2	Resultados na Selva	8
3	Interpolação Bicúbica por partes	11
3.1	Resultados no Zoológico	11
3.2	Resultados na Selva	14
4	Conclusão	17

1 Objetivo

Nosso objetivo nesse Exercício-Programa é gerar um programa que comprima e descomprima uma dada imagem.

Para comprimir, basta remover alguns pixels da imagem, mais precisamente, dado um número real k , e considerando que a imagem é uma matriz de pixels, então removemos as linhas e as colunas i tais que $i \equiv 1 \pmod{k+1}$.

Para fazer a descompressão, assumimos que a imagem é, basicamente, uma função que vai do \mathbb{R}^2 para o \mathbb{R}^3 (\mathbb{R}^3 pois estamos considerando a paleta RGB, logo, uma coordenada para cada cor da paleta) e, dessa forma, cada pixel vira um ponto no plano \mathbb{R}^2 (devemos considerar também que o espaçamento entre dois pixels adjacentes h foi definido por nós). Agora, para inserirmos os novos pixels de coordenadas $(x, y) \in \mathbb{R}^2$ basta interpolarmos a função nessa região.

Para fazer a interpolação, usamos dois métodos diferentes que são extensões para \mathbb{R}^2 dos métodos vistos em sala de aula. Vamos explicar e avaliar os resultados obtidos com cada um deles.

2 Interpolação Bilinear por partes

Esse método é uma extensão para \mathbb{R}^2 da interpolação linear de uma função de \mathbb{R} para \mathbb{R} .

Vamos considerar 4 pixels conhecidos adjacentes: Q_0 , Q_1 , Q_2 e Q_3 . Esses pixels podem ser representados como pontos do plano \mathbb{R}^2 de coordenadas (x_i, y_j) , (x_{i+1}, y_j) , (x_i, y_{j+1}) , (x_{i+1}, y_{j+1}) , respectivamente onde $x_i, x_{i+1}, y_j, y_{j+1} \in \mathbb{R}$. Supondo que a imagem é uma amostra de uma função f tal que $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, então, para gerar um pixel que está em uma coordenada (x_0, y_0) tal que $x_0 \in [x_i, x_{i+1}]$ e $y_0 \in [y_j, y_{j+1}]$, podemos criar uma função interpoladora v , que interpola os valores no intervalo, dada por:

$$v(x, y) = c_0 + c_1(x - x_i) + c_2(y - y_j) + c_3(x - x_i)(y - y_j) \quad (1)$$

Que é a mesma coisa que:

$$\begin{bmatrix} f(x_i, y_j) \\ f(x_{i+1}, y_j) \\ f(x_i, y_{j+1}) \\ f(x_{i+1}, y_{j+1}) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & h & 0 & 0 \\ 1 & 0 & h & 0 \\ 1 & h & h & h^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Podemos, então, utilizar as funções do Octave para calcular os coeficientes rapidamente.

Na nossa implementação, selecionamos cada pixel conhecido e, para cada um da linha i coluna j , definimos que ele é a origem do sistema de coordenadas, assim conseguimos simplificar as contas, já que os pixels adjacentes são $(0, h)$, $(h, 0)$ e (h, h) e teremos sempre as mesmas coordenadas dos pixels que queremos interpolar que serão os pontos tal que $(\frac{m}{k+1}h, \frac{n}{k+1}h)$ onde

$m, n \in 1, 2, \dots, k$. Após isso, calculamos os coeficientes do polinômio v e conseguimos interpolar todos os $(k+2)^2$ pontos que estão no quadrado definido entre o pixel conhecido da linha i coluna j e o pixel da linha $i+1$ coluna $j+1$, incluindo estes quatro pontos já conhecidos.

Para realizar esta interpolação, adaptamos a equação responsável por interpolar um ponto, que é dada por:

$$v(x, y) = [1 \ x \ y \ xy] \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

para que ela pudesse, em uma única multiplicação matricial, nos retornar todos os pontos compreendidos entre $[0, h] \times [0, h]$, em uma matriz quadrada $k+2 \times k+2$.

A equação que nos dá todos os pontos é:

$$\begin{bmatrix} v(x_0, y_0) & \dots & v(x_n, y_0) \\ v(x_0, y_1) & \dots & v(x_n, y_1) \\ \vdots & & \vdots \\ v(x_0, y_n) & \dots & v(x_n, y_n) \end{bmatrix} = X \begin{bmatrix} c_0 & 0 & 0 & 0 \\ 0 & c_1 & 0 & 0 \\ 0 & 0 & c_2 & 0 \\ 0 & 0 & 0 & c_3 \end{bmatrix} Y \quad (2)$$

Onde

$$X = \begin{bmatrix} 1 & x_0 & 1 & x_0 \\ 1 & x_1 & 1 & x_1 \\ \vdots & & \vdots & \\ 1 & x_n & 1 & x_n \end{bmatrix}, Y = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ y_0 & y_1 & \dots & y_n \\ y_0 & y_1 & \dots & y_n \end{bmatrix}$$

e (x_l, y_m) são as coordenadas dos novos pixels que queremos interpolar. Assim, para cada pixel conhecido na imagem (excetuando-se os da última linha e última coluna), calculamos os coeficientes da $v(x, y)$ que abrange o quadrado de lado h formado por este pixel e seus vizinhos, e utilizamos esta multiplicação matricial para interpolar todos os pontos novos dentro do quadrado de uma só vez. Em seguida, esse quadrado é posicionado na matriz que receberá a imagem interpolada, no seu devido local. Ao final, a combinação de todos os quadrados terá gerado a imagem interpolada.

Com esse método, garantimos um custo de tempo quadrático ($\mathcal{O}(p^2)$ onde p é a quantidade de pixels na vertical ou horizontal da imagem original) pois precisamos olhar para cada pixel da imagem.

2.1 Resultados no Zoológico

Nesses experimentos, executamos o método com 6 tipos diferentes de funções $\mathbb{R}^2 \rightarrow \mathbb{R}^3$ para a criação da imagem original, são elas:

- 1) $f(x, y) = (\sin(\frac{1}{x - 50}), \frac{e^x}{xy - 30}, xy)$
- 2) $f(x, y) = (\tan(xy - 10), \cos(x), \sin(\frac{1}{xy}))$
- 3) $f(x, y) = (\frac{1}{\sqrt{xy - 10}}, xy - 10, \cos(x^2 - 2xy + y^2))$
- 4) $f(x, y) = ((x - y)^2, x, y)$
- 5) $f(x, y) = (\sin(x^2 - y^2), e^x + y^2, e^y - x^2)$
- 6) $f(x, y) = (\operatorname{sen}(x), \frac{\operatorname{sen}(y) + \operatorname{sen}(x)}{2}, \operatorname{sen}(x))$

O intervalo que escolhemos para a criação da imagem relativa a essas funções foram de $[0, 200] \times [0, 200]$ com espaçamento de 0.5 entre os pontos.

Perceba que as 3 primeiras funções não são de classe C^1 enquanto as 3 últimas são de classe C^∞ no intervalo definido por nós. Isso é proposital, pois queremos verificar como a nossa interpolação funciona com funções de classe C^2 .

Nos nossos testes, percebemos que o erro não depende de h , por isso, vamos mostrar aqui apenas para $h = 1$.

Vamos verificar o valor do erro da compressão e em seguida a descompressão com $k = 1$.

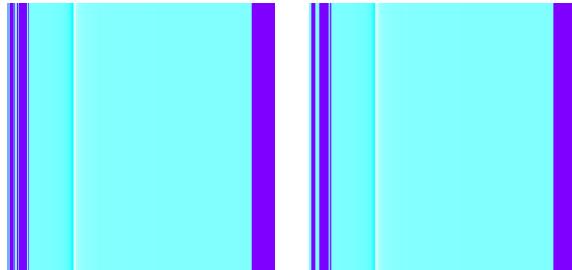


Figure 1: À direita a imagem da função 1 original e à esquerda a imagem após passar pelo processo de compressão e descompressão com $k = 1$.

O erro observado nessa primeira função foi de 5.5861%. Podemos perceber que esse erro ocorre principalmente na parte esquerda da imagem, onde, na imagem original, existem algumas colunas onde a cor vai se suavizando para o azul e, na imagem após o processo, as colunas não possuem essa suavidade. Isso se dá pelo método adotado, que não interpola a função de forma "suave".

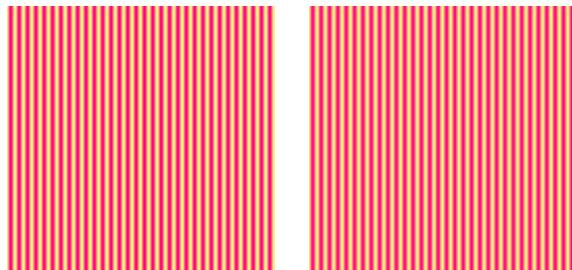


Figure 2: Resultados com a função 2 (original à direita e à esquerda após o processo).

Para a função 2, verificamos um erro de 1.8% entre a imagem original e a imagem após o processo, o pequeno erro pode ser decorrente do fato de que a imagem gerada não possui muitas curvas, e assim a interpolação bilinear se torna suficiente.

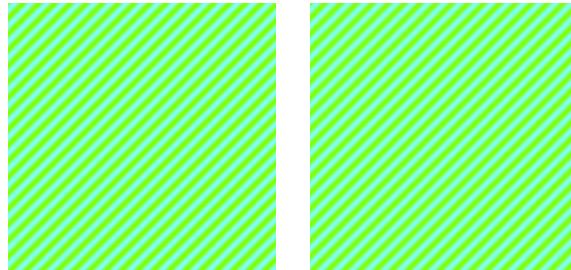


Figure 3: À direita a imagem da função 3 original e à esquerda a imagem após passar pelo processo de compressão e descompressão com $k = 1$.

Para a função 3, temos um erro de 0.74%, provavelmente pelo mesmo motivo da função 2, ou seja, a imagem gerada pela função não possui curvas.

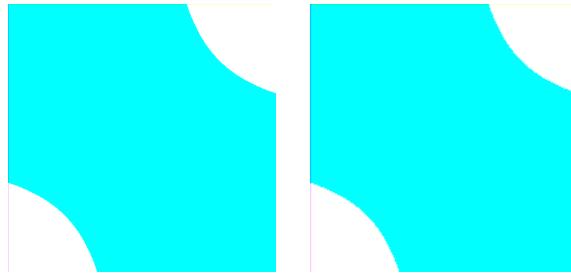


Figure 4: À direita a imagem da função 4 original e à esquerda a imagem após passar pelo processo de compressão e descompressão com $k = 1$.

A função 4 apresentou um erro de 0.8%, um caso interessante, dado que a imagem apresenta algumas curvas. Porém, esse baixo erro deve ser causado pelo fato de que a imagem apresenta uma cor constante no centro, mudando apenas nas bordas de tal forma que a interpolação bilinear seja suficiente, o erro, porém, está nas bordas dos círculos dos cantos.

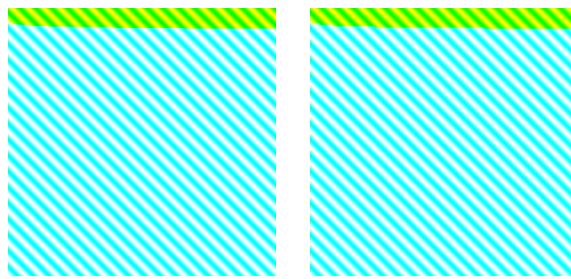


Figure 5: À direita a imagem da função 5 original e à esquerda a imagem após passar pelo processo de compressão e descompressão com $k = 1$.

O erro apresentado pela função 5 é de 1.24%, provavelmente causado pela mudança de cor da parte superior da imagem.



Figure 6: À direita a imagem da função 6 original e à esquerda a imagem após passar pelo processo de compressão e descompressão com $k = 1$.

A função 6, entretanto, apresentou um erro elevado (4.25%), causado pelo fato da função ser muito suave (gerando várias mudanças de cores que ao interpolarmos usando o método bilinear, perderemos as informações).

Percebemos, portanto, que a interpolação funciona bem com todo tipo de imagem (preto e branco ou colorido), com erro máximo de 5% nos experimentos, mas apenas para $k = 1$, vamos ver abaixo um caso que torna o método ineficiente para um k maior. É importante ressaltar que as funções de classe C^2 não apresentaram diferenças em relação àquelas que não são, pois a média dos erros das funções de classe C^2 foi semelhante à média dos erros das que não são de classe C^2 .

Por fim, fizemos mais alguns testes especificamente com a função 6: o primeiro foi comprimir e descomprimir com $k = 7$. Para isso, criamos uma imagem menor (para que exista a condição de existência de um n natural tal que $p = n + (n - 1)k$ onde p é a quantidade de pixels) com 233x233 pixels e o intervalo da função passou a ser de $[0, 115] \times [0, 115]$. O resultado segue abaixo:

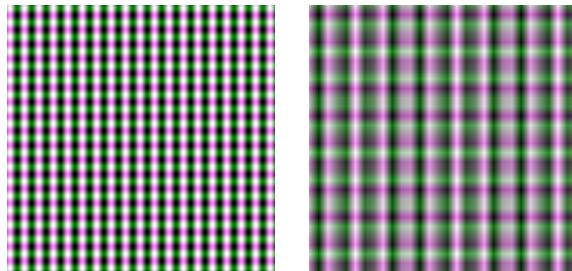


Figure 7: À direita a imagem da função 6 original e à esquerda a imagem após passar pelo processo de compressão e descompressão com $k = 7$.

O erro com esse valor de k subiu para 50%! A causa está na compressão, que faz a imagem de 233x233 pixels para 30x30 pixels, perdendo muita

informação e, ao recriarmos a imagem com o método bilinear, não obtemos as informações novamente, fazendo com que a diferença entre a imagem original e a imagem após o procedimento seja grande.

Nosso último experimento foi rodar o compress três vezes com $k = 1$ (vale ressaltar que a imagem gerada aqui foi exatamente a mesma que a do último experimento, com $k = 7$) e em seguida decompress três vezes com $k = 1$. Obtemos os seguintes resultados:

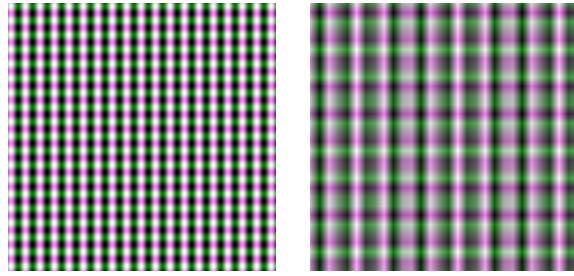


Figure 8: À direita a imagem da função 6 original e à esquerda a imagem após passar pelo processo de compressão e descompressão com $k = 1$ três vezes.

O erro obtido dessa forma foi também de 50% pelo mesmo motivo do último experimento: a imagem, ao ser comprimida, acaba perdendo muita informação (afinal, ao final dos três compress, obtivemos uma imagem de tamanho 30x30 pixels).

Percebemos que o erro se comporta de acordo com k , ou seja, quanto mais comprimimos a imagem, maior o erro obtido.

2.2 Resultados na Selva

Nesses experimentos, usamos 4 imagens: duas preto e branco e duas coloridas. O valor k da compressão foi de 1. Abaixo temos os resultados obtidos:



Figure 9: À direita a imagem original e à esquerda a imagem após o processo. O erro obtido foi de 1.3%.



Figure 10: À direita a imagem original e à esquerda a imagem após o processo.
O erro obtido foi de 1%.

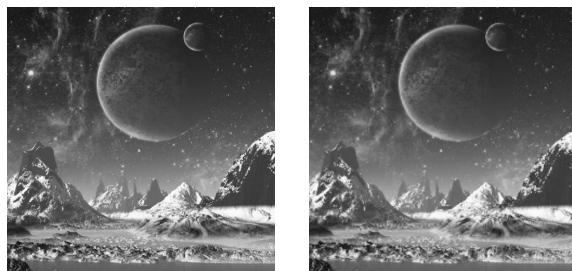


Figure 11: À direita a imagem original e à esquerda a imagem após o processo.
O erro obtido foi de 1.33%.



Figure 12: À direita a imagem original e à esquerda a imagem após o processo.
O erro obtido foi de 1.12%.

O erro obtido aqui foi pequeno para todas as imagens, porém, percebemos que o erro aumenta para as imagens preto e branco (funciona relativamente melhor para imagens coloridas) e além disso, possui um aumento quando a imagem tem muitas curvas (veja a figura 9).

Comprimindo a imagem com h maiores, obtemos imagens cada vez menores e, consequentemente, aumentamos o erro, bem como acontecia no Zoológico. Para constatar esse fato, comprimimos a imagem 1 com $k = 7$ e

verificamos que o erro obtido foi de 5 %. Esse valor de erro também foi observado quando comprimimos a imagem 1 três vezes com $k = 1$ e descomprimimos três vezes com $k = 1$. Abaixo temos os resultados.



Figure 13: À direita a imagem original, ao centro a imagem após uma compressão e descompressão com $k = 7$ e à esquerda a imagem após três compressões e três descompressões com $k = 1$.

3 Interpolação Bicúbica por partes

A ideia desse método é um pouco mais sofisticado que o anterior, pois aqui exigimos que a função seja suave e que a função interpoladora também o seja em todo o domínio da interpolação.

Para entender o funcionamento, vamos considerar novamente os quatro pixels adjacentes Q_0, Q_1, Q_2, Q_3 com as mesmas coordenadas vistas anteriormente. Para interpolar um pixel que possui coordenadas (x_0, y_0) tal que $x_0 \in [x_i, x_{i+1}]$ e $y_0 \in [y_j, y_{j+1}]$, vamos interpolar a função f geradora da imagem por uma função v que interpolará também as primeiras derivadas em relação a x e y e a segunda derivada mista. Isso garante uma aproximação das derivadas (a suavidade da função deverá ser interpolada também).

Em nossa implementação, porém, não tínhamos acesso as derivadas da função geradora da imagem f , por isso, para obter-la, usamos fórmulas que permitem aproximar esses valores. Nas bordas da imagem, utilizamos uma aproximação que leva em conta apenas um pixel a frente, a taxa de erro sendo $\mathcal{O}(h)$. No meio da imagem, usamos a fórmula centrada, que permite um erro de $\mathcal{O}(h^2)$, ou seja, uma aproximação melhor que nas bordas, entretanto, ficamos limitados com o erro das bordas. Vale lembrar que quanto menor for h , melhor será nossa aproximação e, consequentemente, nossa interpolação. Devemos apenas ficar atentos para escolha de um h que seja suficientemente pequeno e que continue mantendo o erro de aproximação do método dominando o erro de arredondamento. Para manusear esses casos diferentes ao calcular as derivadas, implementamos uma moldura matriz, repetindo a primeira e última linhas e colunas. Depois disso, varremos todos os pixels conhecidos, que são os índices da matriz (sem varrer os índices da moldura) e manuseamos a conta normalmente, pois ao referenciarmos índices que estariam fora da matriz, vamos acabar referenciando índices da moldura, que por serem os mesmos da primeira e última linhas e colunas, faz com que automaticamente estejamos utilizando o método de diferença lateral. Depois de calcularmos uma série de derivadas parciais e mistas, calculamos os coeficientes do polinômio e, usando uma estratégia similar a do caso bilinear (porém sem necessidade de modificar a fórmula, pois ela já permite isso), adicionamos mais linhas ao vetor X e mais colunas ao vetor Y para podermos calcular todos os pontos dentro do quadrado de uma vez só. Com isso, posicionamos o quadrado na matriz que se tornará a imagem interpolada, de modo idêntico ao bilinear.

De forma semelhante ao visto no bilinear, como nosso algoritmo passa por cada um dos pixels, então podemos dizer que o tempo consumido é $\mathcal{O}(p^2)$, onde p é a quantidade de pixels em na horizontal (ou vertical) da imagem original.

3.1 Resultados no Zoológico

Nossos experimentos no zoológico foram parecidos com os experimentos da interpolação bilinear por partes. Utilizamos as mesmas seis funções indicadas lá, porém, modificamos o valor de h , já que aqui quanto menor h melhor

será nossa interpolação (pois a aproximação da derivada será melhor).

Fizemos os testes para $k = 1$ e $k = 5$, e verificamos que a alteração do valor de h só tem impacto significativo para alguns testes com $k = 1$ (erro diminuiu aproximadamente 0.0001%).

Abaixo, temos as imagens dos resultados das 6 funções com $h = 10^{-4}$ com os seus respectivos erros:

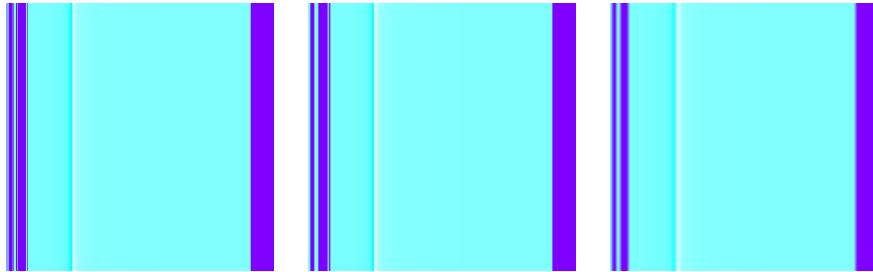


Figure 14: Aqui temos a imagem original gerada pela função 1, a imagem após o processo com $k = 1$ (erro de 5.6%) e a imagem após o processo com $k = 5$ (erro de 7.33%).

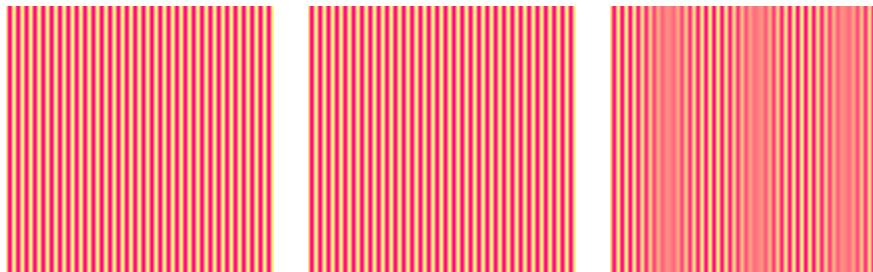


Figure 15: A imagem original gerada pela função 2, a imagem após o processo com $k = 1$ (erro de 5.3%) e a imagem após o processo com $k = 5$ (erro de 12.5%).

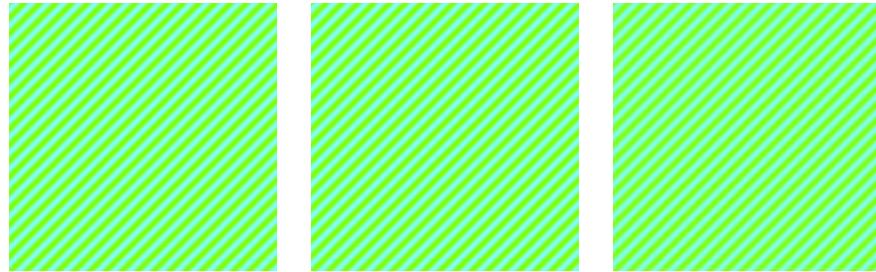


Figure 16: A imagem original gerada pela função 3, a imagem após o processo com $k = 1$ (erro de 0.1%) e a imagem após o processo com $k = 5$ (erro de 2.6%).

Para a função 3, alterando h obtivemos um erro menor (0.12020% para $h = 1$, 0.12018% para $h = 10^{-4}$, 0.12016% para $h = 10^{-6}$).

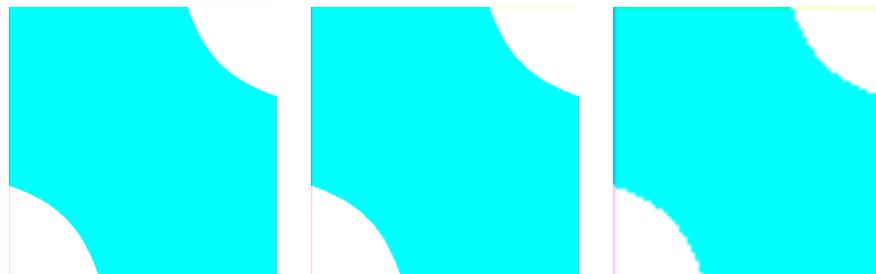


Figure 17: A imagem original gerada pela função 4, a imagem após o processo com $k = 1$ (erro de 0.8%) e a imagem após o processo com $k = 5$ (erro de 2.1%).

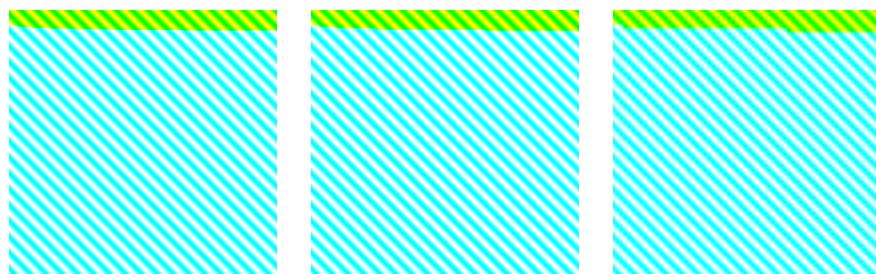


Figure 18: A imagem original gerada pela função 5, a imagem após o processo com $k = 1$ (erro de 0.6%) e a imagem após o processo com $k = 5$ (erro de 3.8%).

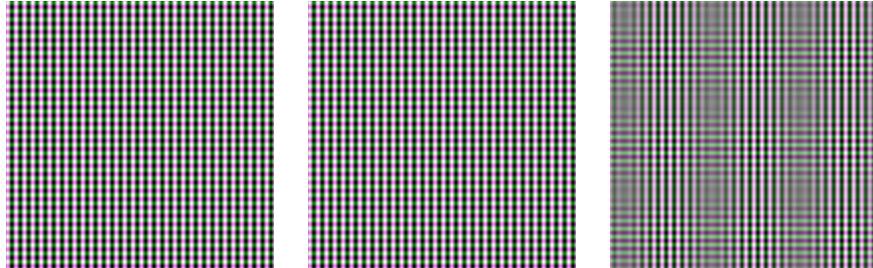


Figure 19: A imagem original gerada pela função 6, a imagem após o processo com $k = 1$ (erro de 0.8%) e a imagem após o processo com $k = 5$ (erro de 3.1%).

É fácil perceber que as funções de classe C^2 tiveram um resultado bem melhor que as funções que não são. Isso se dá pelo fato de que essa interpolação considera as derivadas, garantindo, então, uma melhor aproximação para aquelas funções que são suaves e aumentando o erro daquelas que não são. Também vale notar que o erro aumenta ao passo que aumentamos o valor da compressão.

Experimentamos, também, comprimir e descomprimir a imagem com $k = 7$ (obtemos erro de 54%) e comprimir a imagem três vezes (erro de 55%). O método bilinear obtém um erro menor do que esse para esses dois experimentos.

Também percebemos que o erro não é alterado conforme diminuimos h pela quantidade de informação que perdemos ao longo da compressão.

Abaixo temos as imagens obtidas.

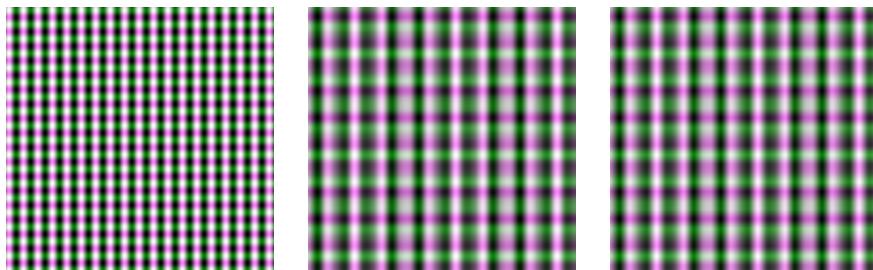


Figure 20: A imagem original gerada pela função 6, a imagem após o processo com $k = 7$ e a imagem após o processo de compressão três vezes e descompressão três vezes.

3.2 Resultados na Selva

Utilizamos, para os experimentos, as mesmas imagens utilizadas para o método bilinear. Da mesma forma que no zoológico, aqui não obtivemos diferenças quando variamos o valor de h . Para os testes, utilizamos apenas $k = 1$ e $h = 0.0001$. Os resultados seguem abaixo.



Figure 21: À direita a imagem original e à esquerda a imagem após o processo.
O erro obtido foi de 1.3%.



Figure 22: À direita a imagem original e à esquerda a imagem após o processo.
O erro obtido foi de 1%.



Figure 23: À direita a imagem original e à esquerda a imagem após o processo.
O erro obtido foi de 1.3%.



Figure 24: À direita a imagem original e à esquerda a imagem após o processo.
O erro obtido foi de 1%.

A qualidade da imagem é muito parecida com a do método bilinear, garantindo um erro pequeno tanto para imagens preto e branco quanto para imagens coloridas (com erro igual para os dois).

Conforme feito no Zoológico, executamos o programa com $k = 7$ e também com $k = 1$ três vezes. Os resultados mostram um erro semelhante aos do caso bilinear (5%). Abaixo temos as imagens obtidas.



Figure 25: À direita a imagem original, ao centro a imagem após uma compressão e descompressão com $k = 7$ e à esquerda a imagem após três compressões e três descompressões com $k = 1$.

4 Conclusão

Com os testes tanto da selva quanto do zoológico, podemos concluir que as duas interpolações são suficientes para imagens com certas características, ou seja, quando precisamos ampliar uma imagem que tenha características mais curvas (aqui podemos dizer que a função geradora dessa imagem seria algo próximo de uma função de classe C^2), é mais interessante escolher o método bicúbico, já que os resultados obtidos para essa classe de funções foi bem melhor (um exemplo é a função 6 do zoológico, que nos mostrou um erro de aproximadamente 4% para o método bilinear contra 0.8% no bicúbico). Já para imagens que não tem características curvas, podemos utilizar o método bilinear, que é mais simples e relativamente mais rápido.