

Trabalho 4

Tiago de Paula Alves (187679)
tiagodepalves@gmail.com

31 de março de 2025

1 Introdução

Esteganografia é uma técnica de comunicação secreta, onde a mensagem é escondida em parte de outro texto ou objeto. Dessa forma, apenas o remetente e o destinatário saberiam onde encontrar a mensagem oculta, podendo se comunicar em sigilo.

Isso é diferente da criptografia em que a mensagem continua visível, mas sem sentido algum, exceto para os comunicadores, que conseguem descriptografar a mensagem. Normalmente, a criptografia acaba sendo aplicada na esteganografia, para garantir maior segurança no processo.

Neste trabalho, foram implementados algoritmos simples de esteganografia em imagens digitais, com objetivo de analisar os casos de uso e os limites desse processo. Além disso, foram discutidos alguns vestígios comuns deixados na esteganografia e possíveis soluções para esses problemas, como o posicionamento dos dados de entrada em posições semi-aleatorizadas.

2 O Programa

Todas as ferramentas foram desenvolvidas e testadas para Python 3.6 ou superior. Além da biblioteca padrão da linguagem, foram utilizados também os pacotes OpenCV, para entrada e saída de imagens, e Numpy, para aplicação vetorializada da esteganografia.

2.1 Código-Fonte

Neste trabalho foram elaboradas duas ferramentas, `codificar.py` e `decodificar.py` que fazem todo o processo de esteganografia. Boa parte do código delas é compartilhado pelos arquivos na pasta `lib`, como apresentado a seguir.

codificar.py Ferramenta de codificação de um arquivo em um imagem.

decodificar.py Ferramenta de recuperação do arquivo escondido.

lib Pacote compartilhado pelas ferramentas.

bits.py Serialização e processamento de dados em vetores de bits.

estego.py Ocultação e recuperação de dados em imagens.

permuta.py Repositionamento pseudo-aleatório dos dados binários.

args.py Processamento dos argumentos da linha de comando.

inout.py Tratamento de entrada e saída do programa.

tipos.py Tipos para checagem estática.

Todas as imagens base para o processamento discutido ao longo do texto estão presente na pasta `imagens`. Os textos de exemplo da esteganografia estão na pasta `textos`.

Também existem dois *scripts* em Python utilizados para análise dos resultados: `plano.py`, para extração de planos de bit, e `dist.py`, para medidas de similaridade de imagens. Por fim, o *script* `run.sh` em Bash refaz todos resultados apresentados neste relatório.

2.2 Execução

A execução de ambos os programas deverá ser feita através do interpretador de Python 3.6+. Os exemplos de execuções a seguir funcionam apenas em Python 3.7+, mas o *script* run.sh também funciona na versão 3.6.

Todas as opções a seguir também estão explicadas com o texto de ajuda de cada ferramenta. Ele pode ser acessado com a *flag* --help ou apenas -h.

2.2.1 Codificação

A primeira ferramenta tem apenas um argumento obrigatório: o caminho da imagem, preferencialmente PNG, onde os dados serão ocultados. Um segundo argumento pode ser passado, especificando o arquivo a ser velado. Caso ele não apareça, o arquivo é lido da entrada padrão.

Por padrão, a imagem resultante é apresentada em uma nova janela gráfica. No entanto, a imagem pode ser salva com a opção --output ou -o e o arquivo de salvamento. A seguir temos o exemplo de como guardar o texto “MC920” na imagem baboon.png com o resultado salvo em saída.png.

```
$ echo MC920 | python3 codificar.py imagens/baboon.png -o saída.png
```

Note que o arquivo ocultado pode ser qualquer arquivo, não apenas texto. Além disso, por padrão o arquivo é escrito no plano de bit menos significativo da imagem. Isso pode ser alterado com a opção --bit ou -b. Por exemplo:

```
$ python3 codificar.py imagens/watch.png imagens/monalisa.png -b 2
```

Existe ainda uma opção sobre a ordem de escrita na imagem, -p ou --permuta, que será melhor apresentada na seção 3.1.

2.2.2 Decodificação

A ferramenta de decodificação é mais simples, recebendo apenas a imagem já esteganografada. Após a decodificação, o arquivo resultante é escrito na saída padrão. Um segundo argumento pode ser especificado para armazenar o resultado.

Note que, para funcionamento correto da ferramenta, é necessário que a decodificação seja feita no mesmo plano de bit em que o arquivo foi embutido. Para isso, existe a *flag* --bit ou -b, assim como na codificação. Por exemplo:

```
$ echo MC920 | python3 codificar.py imagens/peppers.png -o saída.png -b 1  
$ python3 decodificar.py saída.png -b 1  
MC920
```

3 Implementação

O primeiro passo no algoritmo de codificação implementado da esteganografia foi transformar os dados do arquivo de entrada em um vetor de bits. Assim, a mensagem “IC”, dada pelos bytes 0x49 e 0x43, seria representada pelo vetor [1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0]. Para que seja possível a recuperação, também é adicionado um vetor de 64 bits antes do vetor de bits da imagem, representando o tamanho desse vetor.

Para injetar esse vetor na imagem são feitas algumas manipulações binárias simples. Para cada byte da imagem é aplicada uma máscara responsável por zerar o plano de bit especificado. Após isso, o bit que será inserido é deslocado (por *bitshift*) para a posição certa e aplicado por um *bitwise or* no byte da imagem novamente. Isso é feito até encerrar o vetor de bits.

No código-fonte entregue, o mesmo processo é feito de forma vetorizada, utilizando uma matriz booleana para indexação.

Para a decodificação o processo é ainda mais simples. Basta extrair o plano de bit correto em um vetor de bits, recuperar o tamanho do arquivo nos primeiros 64 bits e usar isso para recuperar o vetor dos bits que realmente fazem parte do arquivo. O resultado é uma mera aglutinação desses bits em caracteres ou bytes.

3.1 Permutação

O modelo anterior insere os bits sempre do começo pro final da imagem, o que pode gerar marcas bem claras na imagem como veremos na próxima seção. Uma maneira de evitar isso é distribuindo os dados do arquivo por toda a imagem, como é feito na ferramenta com a opção --permuta ou -p.

Nesse método basicamente é aplicada uma permutação da matriz booleana de indexação, inserindo os dados em posições distribuídas pela imagem. Para evitar o surgimento de algum padrão de bits, os dados da imagem também são permutados antes da inserção.

De forma a tornar o método reversível, mas imprevisível, em cada execução é gerada uma chave aleatória. Essa chave é usada como semente para o gerador pseudo-aleatório, sendo depois inserida nos últimos 64 bits do plano especificado da imagem. Todo esse método de permutação funciona apenas com a versão 1.17 ou superior de Numpy.

Com o tamanho fixado nos primeiros 64 bits e a chave nos últimos 64 bits, os dados dispersos na imagem podem ser facilmente recuperados, revertendo o processo anterior. Para indicar se houve a permutação na esteganografia ou não, a ferramenta usa o primeiro bit do plano de bits, efetivamente reduzindo o tamanho representável para 63 bits. Assim, a ferramenta de decodificação consegue recuperar os dois métodos, sem necessidade de indicação do usuário.

4 Resultados

4.1 Assinatura

Podemos ver pela figura 1 que assinatura digitais, normalmente com menos de caracteres, são imperceptíveis na imagem como um todo. No plano de bit específico, a diferença também não é visível ao olho humano. A medida RMSE¹ entre o resultado e a original é 0.03, com um UIQ² muito próximo de 1.

O resultado na figura 1a contém o texto “Realizzato da Leonardo da Vinci™”, representado em UTF-8 por 35 bytes. Juntamente com o cabeçalho de 8 bytes, a assinatura ocupa 344 bits do plano menos significativo. Isso não chega a metade da primeira linha (256×3 bits de capacidade).

A imagem pode ser produzida por:

```
$ echo Realizzato da Leonardo da Vinci™ | python3 codificar.py
↪ imagens/monalisa.png -o saida.png
```

E verificada com:

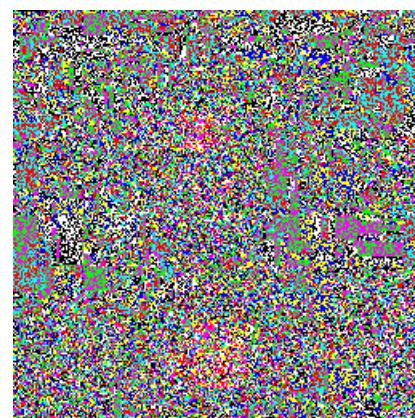
```
$ python3 decodificar.py saida.png
Realizzato da Leonardo da Vinci™
```

¹ Root Mean Square Error. URL: https://en.wikipedia.org/wiki/Root-mean-square_deviation.

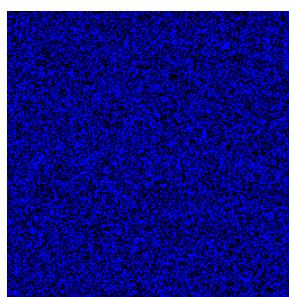
² Zhou Wang e A. C. Bovik. “A universal image quality index”. Em: *IEEE Signal Processing Letters* 9.3 (2002), pp. 81–84. DOI: 10.1109/97.995823.



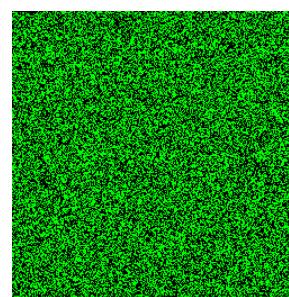
(a) Imagem com mensagem oculta.



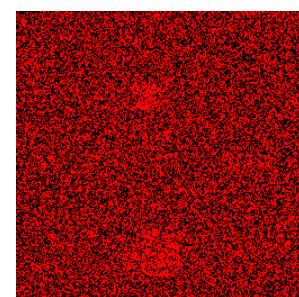
(b) Plano 0.



(c) Plano 0, canal azul.



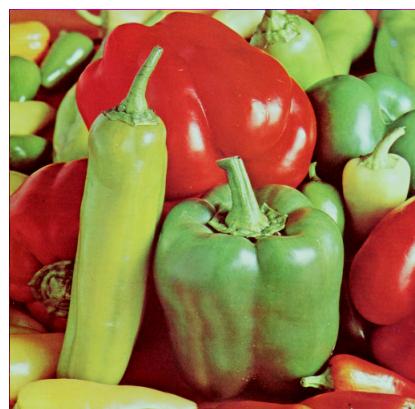
(d) Plano 0, canal verde.



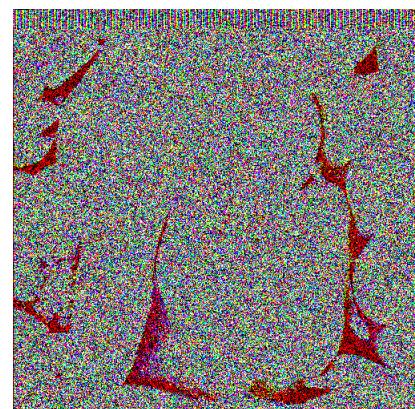
(e) Plano 0, canal vermelho.

Figura 1: monalisa.png com uma assinatura de 33 caracteres.

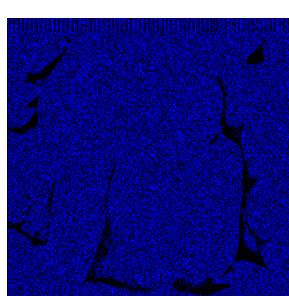
4.2 Texto Pequeno



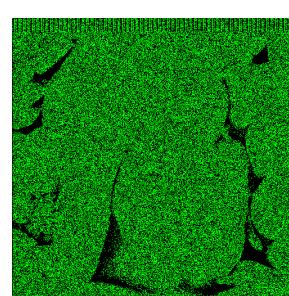
(a) Imagem com mensagem oculta.



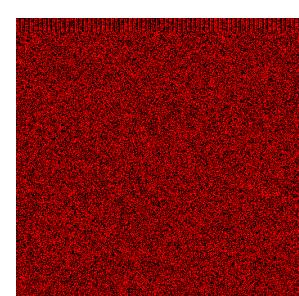
(b) Plano 0.



(c) Plano 0, canal azul.



(d) Plano 0, canal verde.



(e) Plano 0, canal vermelho.

Figura 2: peppers.png com o enunciado .md.

Nesse caso, temos a figura 2a esteganografada com um texto curto, de aproximadamente 600 palavras. O texto, `enunciado.md`, é basicamente uma transcrição do enunciado do trabalho 4 para Markdown, composto de 4.5 KiB de dados.

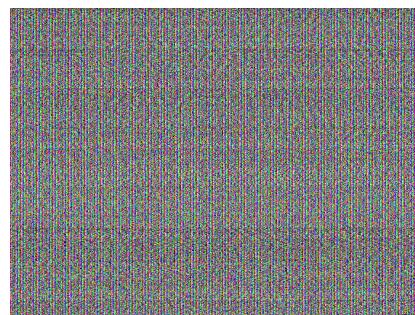
Aqui, a presença do conteúdo já começa a se tornar perceptível no plano de bit menos significativo. Na imagem como um todo, no entanto, ainda não é diferenciável. Os índices confirmam isso com RMSE igual a 0.15 e UIQ também muito próximo a 1.

```
$ python3 codificar.py imagens/peppers.png textos/enunciado.md -o saida.png
# verificação
$ python3 decodificar.py saida.png | diff -qs - textos/enunciado.md
Files - and textos/enunciado.md are identical
```

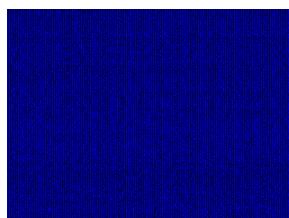
4.3 Texto Grande



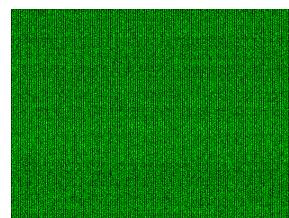
(a) Imagem com mensagem oculta.



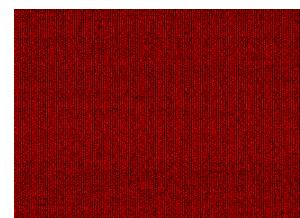
(b) Plano 0.



(c) Plano 0, canal azul.



(d) Plano 0, canal verde.



(e) Plano 0, canal vermelho.

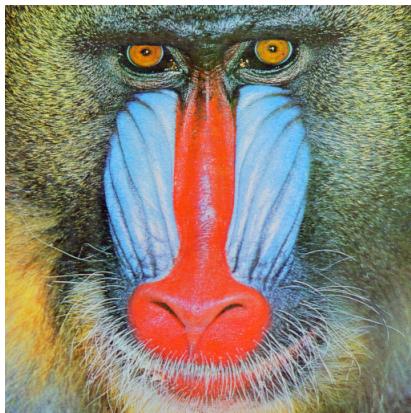
Figura 3: `watch.png` com 28 mil linhas de `words.txt`.

Novamente, mais uma imagem indistinguível da original, apesar de ter um RMSE de 0.7 e UIQ 0.9998. O plano 0 aqui, por estar quase todo preenchido, é um pouco menos suspeito para um observador desabituado. No entanto, pelo padrões de linhas verticais, ele ainda seria facilmente detectável.

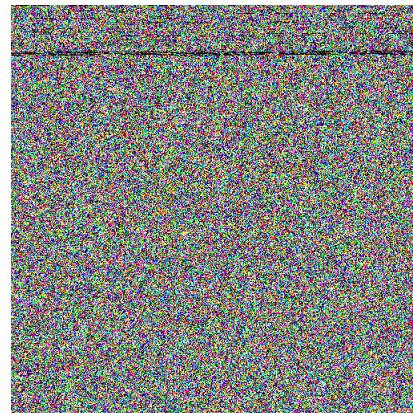
O texto nesse caso é composto das 28700 primeiras palavras do `words.txt`.

```
$ head -n 28700 textos/words.txt | python3 codificar.py imagens/watch.png -o
→ saida.png
# verificação
$ python3 decodificar.py saida.png | diff -qs - <(head -n 28700
→ textos/words.txt)
Files - and /proc/self/fd/12 are identical
```

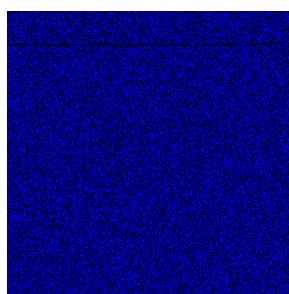
4.4 Arquivos Binários



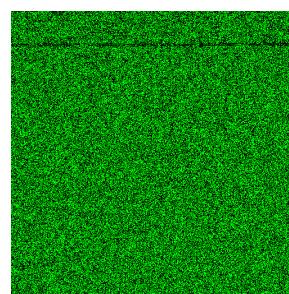
(a) Imagem com mensagem oculta.



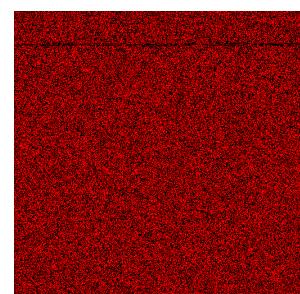
(b) Plano 0.



(c) Plano 0, canal azul.



(d) Plano 0, canal verde.

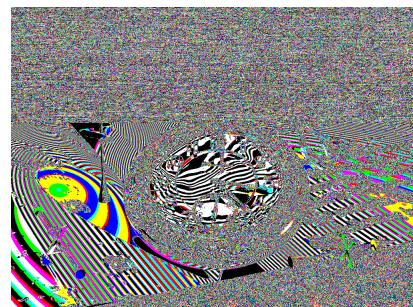


(e) Plano 0, canal vermelho.

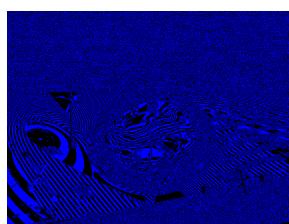
Figura 4: baboon.png com código-fonte comprimido.



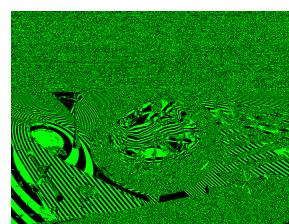
(a) Imagem com mensagem oculta.



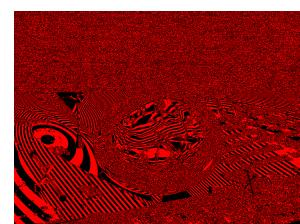
(b) Plano 0.



(c) Plano 0, canal azul.



(d) Plano 0, canal verde.



(e) Plano 0, canal vermelho.

Figura 5: watch.png com monalisa.png oculta.

Para arquivos não-textuais, como arquivos *zip* ou PNG, os padrões gerados na imagens são menos reconhecíveis que as linhas verticais geradas pelos caracteres advindos do padrão ASCII.

Ainda assim, podemos ver um risco horizontal na figura 4 onde o arquivo oculto se encerra, mesmo com um ruído forte no plano 0 da imagem original. Podemos ver também onde a

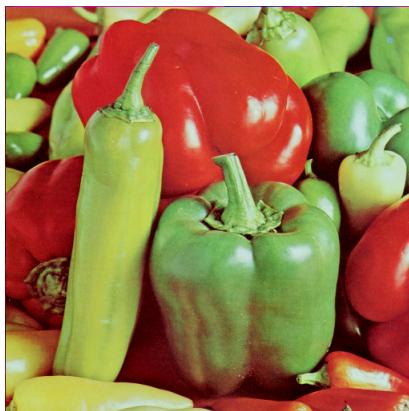
`monalisa.png` se encerra na figura 5, mas isso principalmente porque a `watch.png` tem baixo ruído no plano menos significativo.

```
# ocultação do código fonte comprimido
$ zip -9q codigo.zip lib/*.py *.py textos/enunciado.md
$ python3 codificar.py imagens/baboon.png codigo.zip

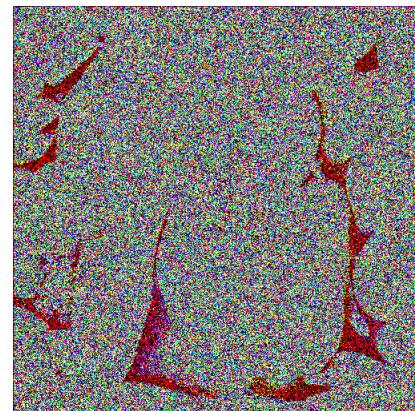
# ocultação da monalisa
$ python3 codificar.py imagens/watch.png monalisa.png
```

4.5 Permutação

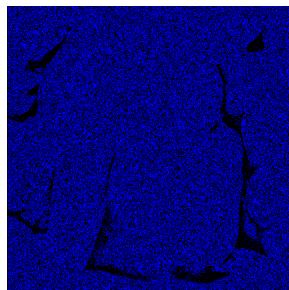
Os dados do arquivo a ser ocultado, assim como discutido na seção 3.1, também podem ser inseridos em posições pseudo-aleatórias da imagem, de forma a redistribuir os bits e ocultar os padrões que podem aparecer. Podemos ver que para a figura 6, o padrão do `enunciado.md` fica praticamente escondido no ruído do plano 0. As medidas de similaridade, RMSE, UIQ e outras, em geral não têm diferenças significativas do algoritmo mais simples, pois não levam em conta a posição do pixel.



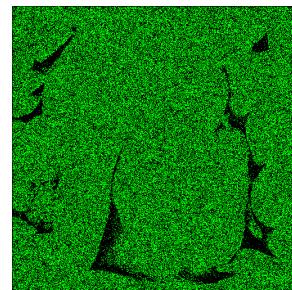
(a) Imagem com mensagem oculta.



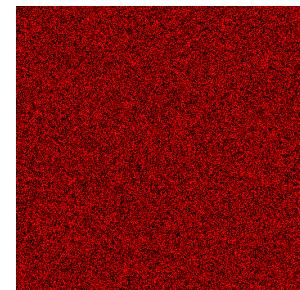
(b) Plano 0.



(c) Plano 0, canal azul.



(d) Plano 0, canal verde.



(e) Plano 0, canal vermelho.

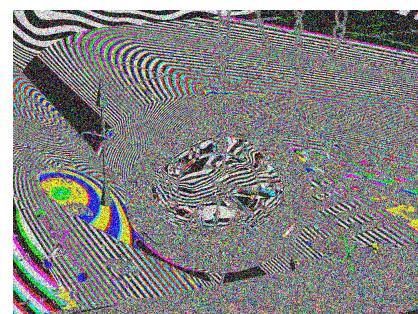
Figura 6: figura 2a com permutação dos dados.

Já na figura 7, podemos ver um outro problema. A imagem original, `watch.png`, não apresenta muito ruído do plano menos significativo. Por conta disso, o pseudo-ruído gerado pela ocultação da imagem `monalisa.png` acaba ficando um pouco estranho. Ainda assim, é argumentável que isso é menos chamativo do que o que aparece em vários outros resultados do algoritmo mais simple.

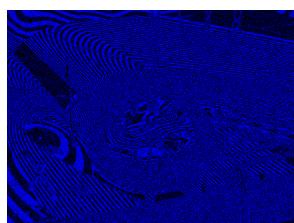
As imagens aqui foram geradas com os mesmos comandos encontrados na seção 4.2 e seção 4.4. A única diferença é a adição da flag `-p`, apresentada na seção 3.1.



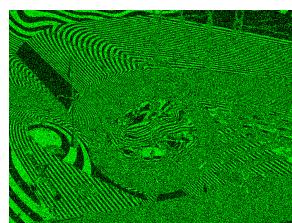
(a) Imagem com mensagem oculta.



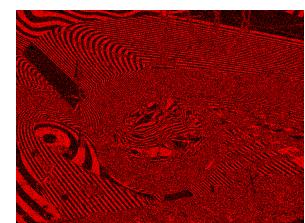
(b) Plano 0.



(c) Plano 0, canal azul.



(d) Plano 0, canal verde.



(e) Plano 0, canal vermelho.

Figura 7: figura 5a com permutação dos dados.

5 Outros Planos de Bit

A influência do plano de bit para a esteganografia é muito importante. Normalmente, em imagens mais naturais, como fotos, os 3 planos menos significativos (0, 1 e 2) causam um impacto quase imperceptível ao olho humano. Para os outros planos, a interferência na imagem original pode acabar sendo percebida. Isso pode ser visto com a `monalisa.png`, que ainda parece com a origina na figura 8e, mas fica claramente alterada na figura 8f.



(a) Original.



(b) Ocultação no plano 0.



(c) Ocultação no plano 1.



(d) Ocultação no plano 2.



(e) Ocultação no plano 3.



(f) Ocultação no plano 4.

Figura 8: Ocultação de `enunciado.md` em `monalisa.png` utilizando alguns planos diferentes. A imagem original acompanha para comparativo.

Em figuras mais geométricas e com menor variação de iluminação, por exemplo, os efeitos podem ser mais aparentes. Isso acontece na figura 9, em que o plano 3 já começa a indicar algum artefato suspeito na imagem.

As medidas de similaridade, como esperado, pioram quanto maior for o plano de esteganografia. No entanto, a medida RMSE apresentou padrão exponencial, dobrando a cada incremento no número do plano. Por exemplo, a figura 8b teve um RMSE de 0.306, a figura 8c foi 0.611, até a figura 8f com $4.93 \approx 0.306 \cdot 2^4$. O padrão também apareceu nos planos mais significativos e na `peppers.png`.



Figura 9: Ocultação das primeiras 5 mil linhas de `words.txt` em `peppers.png`.

6 Conclusão

Pelos resultados da seção anterior, podemos ver que a esteganografia é bem útil quando o objetivo é comunicação sem chamar atenção. Usando os planos de bits menos significativos das imagens é quase impossível que um humano a olho nu suspeite de alguma coisa.

No entanto, algum programa ou programador buscando algo estranho, pode facilmente achar no plano de bit certo. Isso pode ficar bem claro, especialmente com mensagens ou arquivo textuais. Nesse caso, a redistribuição dos dados, discutida na seção 3.1, ou a criptografia dos dados pode acabar ajudando.

Também para arquivos de texto, os traços normalmente são bem marcantes. Isso é evitável também com algum tipo de compressão, como `zip`, que também aumenta a capacidade de armazenamento ou transmissão.