INSTITUTO FEDERAL DE SANTA CATARINA

DANIEL CABRAL CORREA

Implementação do algoritmo de Viterbi para GNU Octave

São José - SC

Dezembro/2025

IMPLEMENTAÇÃO DO ALGORITMO DE VITERBI PARA GNU OCTAVE

Monografia apresentada ao Curso de Engenharia de Telecomunicações do Campus São José do Instituto Federal de Santa Catarina para a obtenção do diploma de Engenheiro de Telecomunicações.

Orientador: Prof. Roberto Wanderley da Nóbrega, Dr.

Coorientador: Prof. Ramon Mayor Martins, Dr.

São José - SC

Dezembro/2025

Daniel Cabral Correa

Implementação do algoritmo de Viterbi para GNU Octave

São José - SC, 23 de Dezembro 2025

Documento assinado digitalmente
ROBERTO WANDERLEY DA NOBREGA
Data: 02/03/2026 13:27:53-0300
Verifique em https://validar.iti.gov.br

Prof. Roberto Wanderley da Nóbrega, Dr.
Orientador
Instituto Federal de Santa Catarina

Documento assinado digitalmente
RAMON MAYOR MARTINS
Data: 02/03/2026 18:45:58-0300
CPF: ***.866.756-**
Verifique as assinaturas em https://v.ufsc.br

Professor Ramon Mayor Martins, Dr.
Coorientador
Instituto Federal de Santa Catarina

Documento assinado digitalmente
DIEGO DA SILVA DE MEDEIROS
Data: 04/03/2026 13:32:28-0300
Verifique em https://validar.iti.gov.br

Professor Diego da Silva de Medeiros, Dr.
Instituto Federal de Santa Catarina

Documento assinado digitalmente
ELEN MACEDO LOBATO
Data: 02/03/2026 22:44:46-0300
Verifique em https://validar.iti.gov.br

Professora Elen Macedo Lobato, Dra.
Instituto Federal de Santa Catarina

AGRADECIMENTOS

Os agradecimentos principais são direcionados à minha esposa Luísa, por todo companheirismo, apoio, dedicação, respeito, estudo e união que tivemos durante esses quase nove anos dentro e fora do curso. Não tenho como agradecer devidamente todos os momentos que passamos juntos até aqui e os que estão por vir.

Agradeço a minha mãe Lucimar, aos meus irmãos Rafael e Gabriel, e a todos meus familiares, por me darem todo amor, apoio, estrutura, incentivo e acreditarem em mim nessa jornada.

À Juliana e especialmente ao Vinícius, por acreditarem em mim, por me incentivarem de duas em duas horas e por terem proporcionado a melhor rede de apoio que eu precisava no momento mais delicado.

A todos os meus amigos, em especial ao Emanuel e ao Raphael, por estarem comigo nos melhores e piores momentos, por me motivarem e acreditarem em mim sempre, por serem meus irmãos. Gostaria de citar também Rafaela e Jean por todo apoio e companheirismo a mim cedido.

Aos meus amigos de graduação, em especial ao Anderson, Adonis, Daniel, Jennefer, Maria Fernanda, Pedro e Vitor, por todos os momentos vividos durante essa graduação, às várias horas de estudos, trabalhos, conquistas e momentos de descontração.

Ao Instituto Federal de Santa Catarina Campus São José que tem me oferecido por mais de 15 anos um ensino de qualidade que dinheiro nenhum no mundo poderia pagar. É uma honra ter sido aluno do IFSC nos cursos técnico integrado e de engenharia de telecomunicações.

Gostaria de agradecer a todos os mestres e professores que tive até aqui, à minha mãe que me ensinou as primeiras letras e números. À professora Tânia Peruffo, a melhor professora de matemática que tive em minha vida. Evandro Cantú, Volney Duarte Gomes, Marcos Moecke, Emerson Ribeiro de Mello, e muitos outros, mestres que me influenciaram a escolher pela engenharia de telecomunicações e mostraram que essa é a área que desejo seguir. Um agradecimento especial ao Roberto Wanderley da Nóbrega e ao Ramon Mayor Martins, que me orientaram nesse trabalho, acreditaram em mim e persistiram em me instruir mesmo nos momentos mais difíceis. Também gostaria de agradecer aos professores Diego e Elen por aceitarem participar da banca de avaliação deste trabalho.

Gostaria de agradecer também às pessoas que perdi ao longo da minha vida, em especial ao meu pai Valdemiro, meus avôs Antônio, Maria, Isabel e Francisco, e minha tia de consideração Dircéia, todos foram muito importantes para ser quem eu sou hoje e de

onde estiverem espero que estejam orgulhosos pela pessoa que me tornei.

Por fim, gostaria de dedicar esse trabalho ao meu filho que está para chegar, Pietro. Nessa reta final ele foi o gás que eu precisava para concluir este projeto.

RESUMO

O presente trabalho tem como objetivo desenvolver a função `vitdec` para o software `GNU Octave`, com o intuito de disponibilizar a função em uma ferramenta gratuita ampliando o acesso ao ensino e à pesquisa. Essa função é a implementação do algoritmo de Viterbi, um método de programação dinâmica que resolve o problema de estimação de sequências em modelos ocultos de Markov. Uma de suas aplicações em comunicação digital é a decodificação de códigos convolucionais, possibilitando a correção de erros de dados após serem transmitidos. Neste documento será contextualizado o estudo realizado sobre códigos convolucionais e o algoritmo de Viterbi. Além disso, será apresentado o desenvolvimento do projeto. Por fim, analisamos os resultados obtidos e traçamos as conclusões do trabalho.

Palavras-chave: Algoritmo de Viterbi, Vitdec, Decodificação, GNU Octave, Códigos convolucionais.

ABSTRACT

This work aims to develop the vitdec function for GNU Octave, providing this tool in a free software environment to broaden access to education and research. This function implements the Viterbi algorithm, a dynamic programming method used to solve sequence estimation problems in Hidden Markov Models (HMM). In digital communications, one of its primary applications is the decoding of convolutional codes, which allows for error correction in transmitted data. This document provides the context for the study of convolutional codes and the Viterbi algorithm, followed by a description of the project's development. Finally, the results are analyzed, and conclusions are presented.

Keywords: Viterbi algorithm, Vitdec, Decoding, GNU Octave, Convolutional Codes.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

AWGN *Additive White Gaussian Noise.*

BER *Bit Error Rate.*

CDMA *Code Division Multiple Access.*

GSM *Global System for Mobile Communications.*

HMM *Hidden Markov Model.*

LTE *Long Term Evolution.*

SUMÁRIO

1 INTRODUÇÃO

Em um sistema de comunicação digital um dos pontos mais importantes a serem tratados é a confiabilidade na transmissão de informações. Para garantir essa questão, em um meio ruidoso e com interferências, são utilizadas técnicas de codificação e decodificação de erro. Uma das técnicas que se destaca é a codificação convolucional com decodificação pelo algoritmo de Viterbi. Essa combinação gera redundância controlada na transmissão com uma estratégia ótima de recuperação de dados no receptor.

O algoritmo de Viterbi é um método de programação dinâmica que resolve o problema de estimação de sequências em *Hidden Markov Model* (HMM). Ele opera por meio de uma treliça de estados, calculando métricas de distância (ex.: Hamming) entre os símbolos recebidos e todas as possíveis sequências codificadas. A cada etapa, mantém apenas o caminho de menor custo acumulado (sobrevivência de trajetórias), descartando combinações subótimas. Com isso, reduz-se a complexidade computacional, evitando que de fato se analise todas as combinações, garantindo eficiência mesmo em cenários de alto ruído.

A importância do algoritmo de Viterbi se deve à sua eficiência em encontrar a sequência mais provável transmitida, mesmo em ambientes ruidosos, o que o torna fundamental em aplicações de telecomunicações e armazenamento digital. Em telecomunicações, tal algoritmo é utilizado em uma diversa gama de aplicações como tecnologias celulares *Long Term Evolution* (LTE), *Code Division Multiple Access* (CDMA), *Global System for Mobile Communications* (GSM), *modem*, transmissão via satélite, *wireless* 802.11 até comunicações em espaço profundo (SHPIGELBLAT, 2009).

Desta forma, ferramentas como o `MATLAB` e o `GNU Octave` são amplamente utilizadas no ambiente acadêmico e de pesquisa para simulação e validação desses algoritmos. Por se tratar de uma tecnologia proprietária, o `MATLAB` disponibiliza diversas funções prontas, como a função `vitdec` desenvolvida para realizar a decodificação utilizando o algoritmo de Viterbi, contribuindo para o desenvolvimento de protótipos e experimentos (MathWorks, 2024c). Entretanto, sua licença possui alto custo, limitando o livre acesso e a replicação científica dos experimentos, sobretudo nos casos de instituições públicas e projetos de código aberto. Em contrapartida, o `GNU Octave` é uma solução livre e compatível com grande parte da sintaxe do `MATLAB`, embora ainda esteja defasado em relação às implementações nativas para funções avançadas, como ocorre com o `vitdec`. Isso dificulta a migração de projetos e restringe o potencial de inclusão digital e colaboração aberta.

1.1 Definição do Problema

Apesar da importância do algoritmo de Viterbi e da popularidade do `GNU Octave` como ferramenta de simulação, não existe atualmente uma implementação nativa da função `vitdec` no `GNU Octave`. Isso pode criar uma barreira para pesquisadores, estudantes e desenvolvedores que dependem de soluções livres para validar algoritmos de comunicação digital, especialmente em contextos onde o acesso ao `MATLAB` é inviável por questões financeiras ou de licenciamento. A ausência dessa função compromete a reprodutibilidade de experimentos, dificulta o ensino e restringe o avanço de pesquisas colaborativas baseadas em *software* livre.

1.2 Objetivos

O principal objetivo deste estudo é disponibilizar em uma ferramenta livre, como o `GNU Octave`, uma função baseada no algoritmo de Viterbi, como o `vitdec` do `MATLAB`, de modo a facilitar o acesso da comunidade científica e acadêmica, além de ampliar a capacidade da ferramenta. O desenvolvimento de uma implementação da função `vitdec` no `GNU Octave` visa preencher uma lacuna relevante, promovendo a democratização do acesso à tecnologia e a reprodutibilidade dos experimentos em comunicações digitais.

1.2.1 Objetivo geral

Desta forma, o principal objetivo desse trabalho é desenvolver pelo menos uma das variantes da função `vitdec` para o `GNU Octave`, simulando o algoritmo de Viterbi para decodificação de códigos convolucionais.

1.2.2 Objetivos específicos

Além disso, os seguintes objetivos específicos são estabelecidos:

- Estudar os fundamentos teóricos dos códigos convolucionais e do algoritmo de Viterbi;
- Analisar a documentação da função `vitdec` do `MATLAB` e identificar requisitos para sua implementação no `GNU Octave`;
- Desenvolver e validar a função no `GNU Octave`, comparando seus resultados com os do `MATLAB`;
- Realizar testes e análise de desempenho para garantir a confiabilidade da solução.

1.3 Escopo do Estudo

O escopo deste trabalho está restrito à implementação da versão *hard-decision* da função `vitdec`, utilizando apenas recursos disponíveis no `GNU Octave` e validando os resultados por meio de comparações com o `MATLAB`. Não serão abordadas, neste momento, as variantes *soft-decision* ou *unquant* da função, nem a integração com outros pacotes externos. O estudo se concentra em canais binários e estruturas de treliça compatíveis com as funções já existentes no `GNU Octave`, visando garantir a compatibilidade e a facilidade de uso para a comunidade.

1.4 Significância da Pesquisa

Esta pesquisa contribui para a área de telecomunicações, em específico comunicações digitais, ao preencher uma lacuna importante no ecossistema de *software* livre, promovendo a inclusão digital e a democratização do acesso a ferramentas de simulação avançadas. Ao disponibilizar uma implementação aberta e validada da função `vitdec` para o `GNU Octave`, o trabalho facilita a reprodutibilidade científica, estimula o desenvolvimento colaborativo e reduz a dependência de soluções proprietárias, beneficiando tanto o ensino quanto a pesquisa em engenharia e tecnologia da informação. Além disso, ao fortalecer o pacote de comunicações do `GNU Octave`, a pesquisa abre caminho para futuras extensões e aprimoramentos, contribuindo para o avanço tecnológico e científico do país.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda a fundamentação teórica do projeto desenvolvido. Particularmente, discutimos sobre a modelagem matemática dos códigos convolucionais e suas representações gráficas e, finalmente, sobre o algoritmo de Viterbi. Além disso, também discutiremos sobre os ambientes de computação numérica e as funções relacionadas ao algoritmo de Viterbi e códigos convolucionais.

2.1 Códigos Convolucionais: Modelagem Matemática

A teoria da codificação de erros é um dos pilares para garantir a integridade dos dados transmitidos em sistemas digitais. Entre as técnicas mais relevantes, destacam-se os códigos convolucionais, introduzidos por Peter Elias em 1955, que utilizam memória e operações de convolução para gerar sequências codificadas com redundância, permitindo a detecção e correção de erros em canais ruidosos (HAYKIN, 2007). O codificador convolucional pode ser modelado como uma máquina de estados finitos, composta por k entradas, por M registradores de deslocamento e n somadores módulo 2, sendo a taxa de código r definida pela razão entre o número de bits de entrada kL e de saída $n(L+M)$, expressa por

$$r = \frac{k.L}{n(L+M)} \ . \tag{2.1}$$

Considerando que $L \gg M$, podemos simplificá-la para

$$r = \frac{k}{n} \ . \tag{2.2}$$

O comprimento de restrição de um código convolucional pode ser descrito como a quantidade de deslocamentos necessários para que um único bit de mensagem não influencie mais na saída do codificador. Sendo assim, o comprimento de restrição K é definido por

$$K = M + 1 \tag{2.3}$$

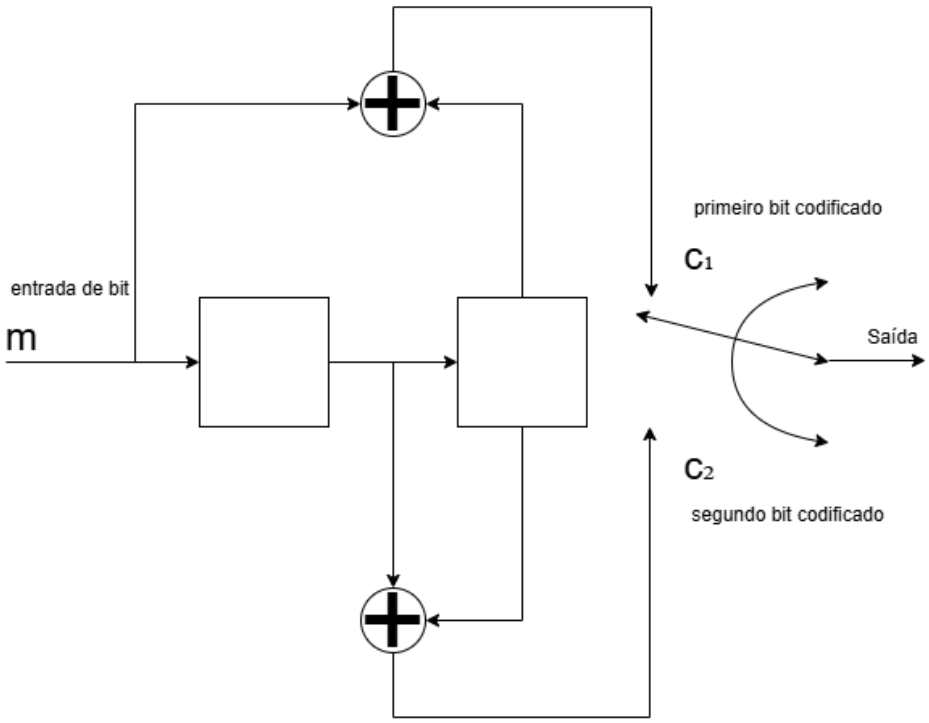
considerando que $k = 1$. Caso $k > 1$, então o comprimento de restrição deixa de ser um valor único e passa a ser um vetor de k elementos, tendo um valor K para cada bit de entrada.

Segundo Haykin (2007), podemos caracterizar cada caminho entre a entrada e a saída de um codificador convolucional a partir de sua resposta ao impulso, com cada registrador fixado no estado zero inicialmente. Deste modo, podemos determinar cada percurso em termos de um polinômio gerador, definido como a transformada de

retardo unitário da resposta ao impulso. A sequência geradora, descrita pelos coeficientes $g_0^{(i)}, g_1^{(i)}, g_2^{(i)}, ..., g_M^{(i)}$, define a resposta ao i-ésimo percurso com $i = 1, \ldots, n$, onde n indica o número de somadores (saídas) do codificador. Tais coeficientes assumem valores binários (0 ou 1), onde o valor unitário indica uma conexão ativa do registrador com o i-ésimo somador. Deste modo, o polinômio gerador do i-ésimo percurso é representado por

$$g^{(i)}(D) = G_0^{(i)} + G_1^{(i)} D + G_2^{(i)} D^2 + \cdots + G_M^{(i)} D^M \tag{2.4}$$

onde D indica a variável de retardo unitário. O codificador convolucional completo é descrito pelo conjunto de polinômios geradores $g^{(1)}(D), g^{(2)}(D), ..., g^{(n)}(D)$.



Fonte: Elaborada pelo autor.

Figura 1 – Exemplo de codificador convolucional

Vamos exemplificar o que vimos até aqui, tendo como base a Figura 1. Na figura temos um codificador convolucional de $k = 1$, $n = 2$ e $K = 3$. Tal codificador possui taxa de código de 1/2, operando de forma sequencial sobre a sequência de mensagem de entrada, isto é, um bit por vez. Neste exemplo temos dois percursos onde, para o primeiro (superior) a resposta ao impulso do percurso é (1,0,1). A partir disso, o polinômio gerador correspondente é dado por

$$g^{(1)}(D) = 1 + D^2 \ .$$

A resposta ao impulso do outro percurso é (0,1,1). Então, o polinômio gerador correspondente pode ser representado por

$$g^{(2)}(D) = D + D^2 \ .$$

Para a sequência de mensagem (1011011), temos a seguinte representação polinomial

$$m(D) = 1 + D^2 + D^3 + D^5 + D^6 \ .$$

À semelhança da transformada de Fourier, a convolução no domínio de tempo é transformada em multiplicação no domínio D (HAYKIN, 2007). Então, o polinômio de saída do percurso 1 é

$$\begin{aligned}
c^{(1)}(D) &= g^{(1)}(D)m(D) \\
c^{(1)}(D) &= (1 + D^2)(1 + D^2 + D^3 + D^5 + D^6) \\
c^{(1)}(D) &= 1 + D^2 + D^3 + D^5 + D^6 + D^2 + D^4 + D^5 + D^7 + D^8 \\
c^{(1)}(D) &= 1 + D^3 + D^4 + D^6 + D^7 + D^8 \ .
\end{aligned}$$

Com isso deduzimos que a sequência de saída do percurso superior é 100110111. Seguindo para o segundo percurso

$$\begin{aligned}
c^{(2)}(D) &= g^{(2)}(D)m(D) \\
c^{(2)}(D) &= (D + D^2)(1 + D^2 + D^3 + D^5 + D^6) \\
c^{(2)}(D) &= D + D^3 + D^4 + D^6 + D^7 + D^2 + D^4 + D^5 + D^7 + D^8 \\
c^{(2)}(D) &= D + D^2 + D^3 + D^5 + D^6 + D^8 \ .
\end{aligned}$$

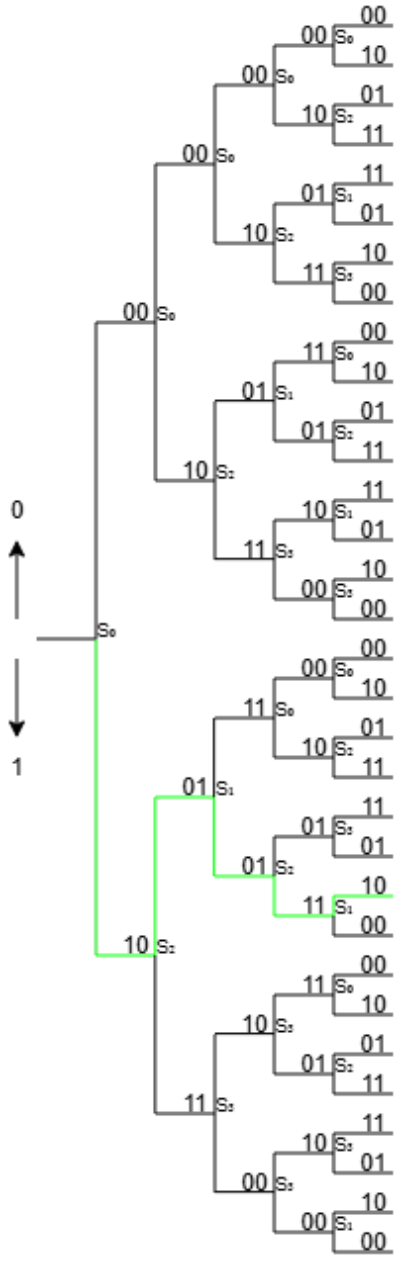
Logo, a sequencia de saída do percurso inferior é 011101101. Para termos a sequência codificada devemos multiplexar as sequências dos dois percursos. Então, teremos

$$c = 100101111001111011 \ .$$

Após a passagem no codificador, a informação de tamanho $L = 7$ bits gera uma sequência de tamanho $n(L + K - 1) = 18$ bits. Isso se deve ao fato de que, ao codificar a mensagem, o registrador de deslocamento deve voltar ao estado zero, sendo assim, uma sequência de zeros de tamanho $K-1$ deve ser anexada ao fim da mensagem. Essa sequência para a limpeza dos registradores é conhecida como *cauda* da mensagem.

2.1.1 Representações gráficas

Geralmente um codificador convolucional tem suas propriedades estruturais retratadas em três diagramas: (1) árvore de códigos, de (2) treliça e de (3) estados. Para facilitar o entendimento utilizaremos o codificador convolucional da Figura 1 para esclarecer o que cada um desses diagramas pode oferecer.



Fonte: Elaborada pelo autor.

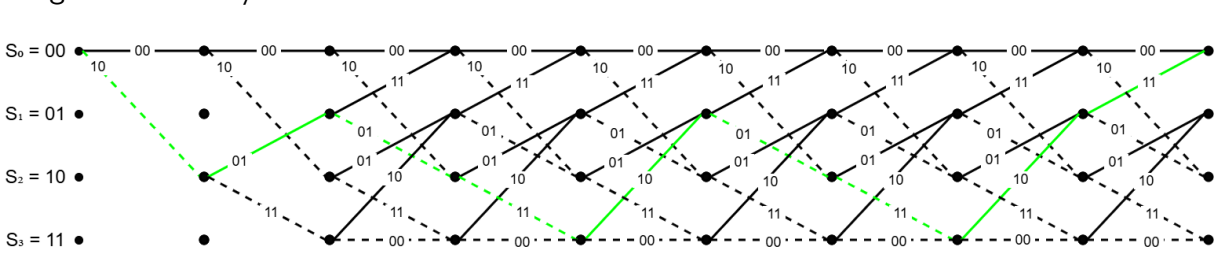
Figura 2 – Árvore de códigos parcial correspondente ao codificador convolucional da Figura 1

Diagrama de árvore de códigos

A Figura 2 exibe um diagrama de árvore de códigos, que utiliza cada ramo para representar um bit de entrada com o par de bits de saída correspondente indicado no ramo (HAYKIN, 2007). Por convenção, ao receber um bit 0 na entrada do codificador convolucional seguiremos na parte superior da bifurcação, indo para a parte inferior quando recebemos um bit 1. Para identificarmos uma mensagem codificada, devemos acompanhar o percurso da árvore (da esquerda para a direita) de acordo com a sequência de entrada,

os símbolos correspondentes das ramificações formam a mensagem codificada. Recordando a sequência do exemplo citado a pouco, teremos a sequência de entrada (1011011) e considerando apenas os 5 primeiros dígitos. Se seguirmos as ramificações (caminho verde) obtemos a sequência codificada correspondente (10, 01, 01, 11, 10), que está de acordo com o exemplo.

Analisando a Figura 2, pode-se notar que a árvore começa a se repetir após os três primeiros ramos. Isso está associado à sua memoria, neste caso $M = K - 1 = 2$ bits de mensagem. Logo, quando estamos inserindo o terceiro bit no codificador, o primeiro deixa de ter influência sobre ele. Por conta disso, diferentes sequências de mensagem que compartilham os mesmos dois bits mais recentes passam a produzir exatamente os mesmos símbolos-código. Assim, os nós da árvore que possuem o mesmo rótulo representam o mesmo estado interno do codificador e podem ser unificados. Esse agrupamento leva à forma reduzida da árvore de códigos, apresentada na Figura 3.



Fonte: Elaborada pelo autor.

Figura 3 – Treliça correspondente ao codificador convolucional da Figura 1

A treliça recebe esse nome por conta de sua estrutura em forma de árvore cujos ramos se entrelaçam. Por convenção, a forma para diferenciar seus ramos se dá pelo seguinte: caso o ramo seja gerado por uma entrada de bit 0 ele será desenhado com uma linha cheia; caso seja por um bit 1 será uma linha tracejada. Quanto ao exemplo do codificador da Figura 1, conseguimos visualizar na Figura 3 que, ao inserirmos a sequência de entrada (1011011) no codificador, obtemos a sequência codificada correspondente (10,01,01,11,10,01,11), o que está de acordo com o exemplo.

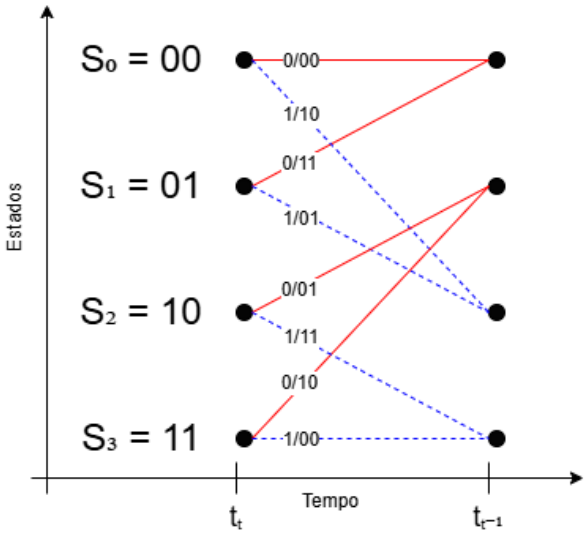
Comparando a treliça com a árvore de códigos, percebemos que a primeira é muito mais informativa, além de expor que o codificador convolucional está associado a uma máquina de estados finitos. Para determinar a quantidade de estados, considera-se 2^M, sendo M a quantidade de registradores. No exemplo da Figura 1, temos $M = 2$. Então, o estado que esse codificador pode assumir é um dos quatro valores possíveis como consta na Tabela 1. Normalmente a treliça possui $(L+K)$ níveis (também chamado de profundidade), onde L é o comprimento da sequência de mensagem de entrada e K, o comprimento de

restrição do código. Os primeiros e últimos M níveis correspondem, respectivamente, ao afastamento e ao retorno ao estado inicial do codificador. Por conta disso, não é possível alcançar todos os estados nesses trechos. Isso se difere da parte central onde é possível alcançar qualquer estado. Além disso, podemos observar uma estrutura periódica fixa na parte central como é exibida na Figura 4.

Tabela 1 – Tabela de estados para o codificador convolucional da Figura 1

Estado	**Descrição binária**
S_0	00
S_1	01
S_2	10
S_3	11

Fonte: Elaborada pelo autor.

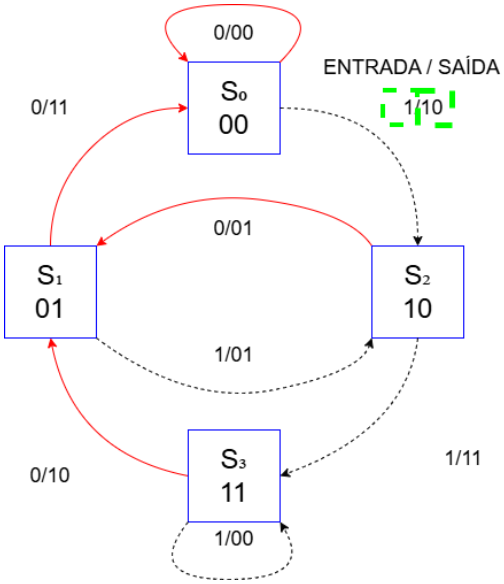


Fonte: Elaborada pelo autor.

Figura 4 – Recorte central da treliça correspondente ao codificador convolucional da Figura 1

Diagrama de máquina de estados

Observe novamente a Figura 4, que exibe um trecho da parte central da treliça. Apenas essa imagem não é suficiente para determinar o estado que o codificador deveria estar, sendo possível que ele esteja em qualquer um deles. Os nós à esquerda representam os estados atuais possíveis, enquanto os da direita representam os próximos estados possíveis. Assim sendo, podemos juntar os nós da esquerda e da direita obtendo o diagrama de estados do codificador, exibido na Figura 5. Tal figura representa os quatro estados possíveis do codificador. Cada nó possui duas ramificações de entrada e saída. Para representar as transições de estado, quando o codificador recebe o bit 0, o estado é alterado seguindo a linha cheia. Caso contrário, segue a ramificação tracejada.



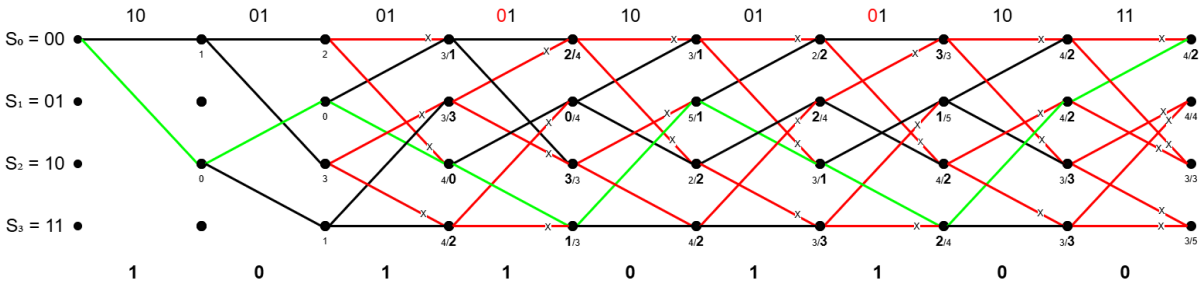
ESTADO ATUAL	ENTRADA	PROXIMO ESTADO	SAÍDA
S_0 00	0	S_0	00
	1	S_2	10
S_1 01	0	S_0	11
	1	S_2	01
S_2 10	0	S_1	01
	1	S_3	11
S_3 11	0	S_1	10
	1	S_3	00

Fonte: Elaborada pelo autor.

Figura 5 – Máquina de estados correspondente ao codificador convolucional da Figura 1

Com o diagrama de estados, conseguimos indicar facilmente a informação codificada. Iniciaremos pelo estado S_0 e avançamos de acordo com a sequência de mensagem (1011011), percorrendo o caminho $S_0, S_2, S_1, S_2, S_3, S_1, S_2, S_3$ e , portanto, obtendo a sequência (10,01,01,11,10,01,11), completando a relação entre a entrada e a saída do codificador convolucional.

2.2 Algoritmo de Viterbi: Otimização Dinâmica



Fonte: Elaborada pelo autor.

Figura 6 – Correção da mensagem recebida utilizando o algoritmo de Viterbi com o codificador convolucional da Figura 1

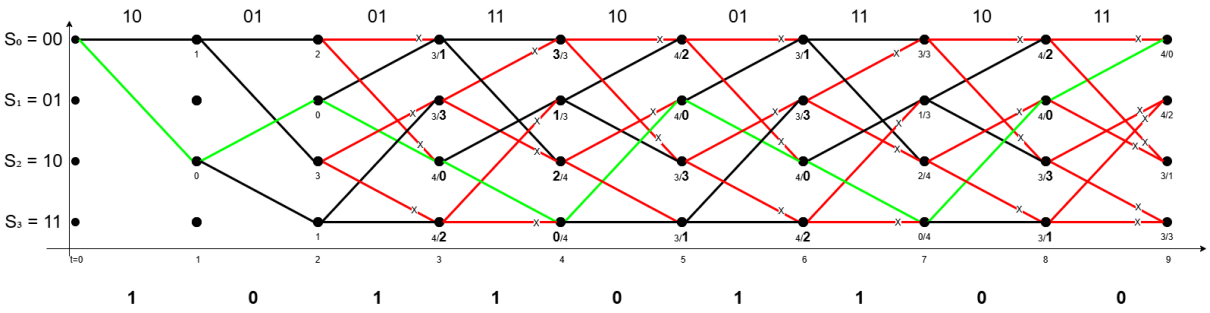
A decodificação ótima dos códigos convolucionais foi viabilizada a partir de 1967 com o desenvolvimento do algoritmo de Viterbi por Andrew Viterbi (HAYKIN, 2007). O algoritmo utiliza uma estrutura de treliça para encontrar, via programação dinâmica, o caminho mais provável percorrido pela sequência transmitida, como vemos na Figura 6,

minimizando a métrica de distância (como a distância de Hamming, no caso de decisão rígida). Viterbi, então, tornou-se padrão em diferentes aplicações como, por exemplo, comunicações via satélite, redes móveis e armazenamento digital, devido à sua eficiência e desempenho próximo ao ótimo.

No algoritmo de Viterbi utilizamos o diagrama de treliça como representação do codificador convolucional, devido ao fato de que o diagrama de árvore de código cresce exponencialmente de acordo com o crescimento da mensagem de entrada, enquanto o diagrama de treliça mantém os ramos em 2^{K-1}.

A Figura 7 traz a representação de como utilizamos o algoritmo de Viterbi, usando como base a mensagem enviada no exemplo anterior (10,01,01,11,10,01,11). Como o codificador convolucional deste exemplo (Figura 1) tem $K = 3$, no instante $t = 3$ é possível observar a chegada de dois percursos em cada um dos estados. Nesse ponto, o decodificador deve tomar a decisão de qual dos dois percursos deve se manter. Isso ocorrerá novamente em todos os outros pontos até o fim da mensagem. Isto é, justamente, o funcionamento do algoritmo de Viterbi: percorre toda a treliça, decidindo qual percurso deve ser mantido.

O algoritmo computa as métricas de todo o percurso na treliça. A métrica é a diferença entre um determinado percurso e a sequência recebida. Tal comparação é denominada distância de Hamming. Em seguida, é realizada a comparação entre dois caminhos que chegam ao mesmo nó e o percurso com a maior métrica é descartado, como vimos na Figura 7 (os caminhos em vermelho). Os caminhos que são mantidos, juntamente com suas métricas, são chamados de percursos sobreviventes ou ativos, que estão em preto ou verde. Esse processo continua até o final da mensagem, quando a treliça deve retornar ao estado 0. No caso em que as métricas de dois ramos são idênticas, o algoritmo seleciona arbitrariamente qual dos percursos deve ser mantido.



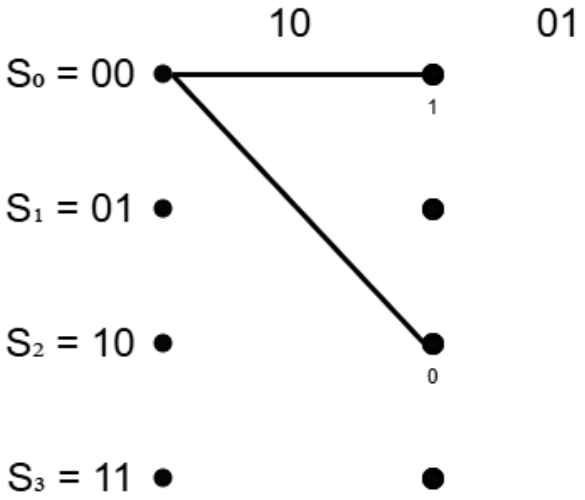
Fonte: Elaborada pelo autor.

Figura 7 – Exemplo de uso do algoritmo de Viterbi do codificador convolucional da Figura 1

O Comportamento do algoritmo de Viterbi pode ser descrito nos passos a seguir.

Inicialização

Inicie identificando o estado 0 da treliça, pois consideramos que as memórias estarão limpas. Sendo assim, o estado inicial sempre será 0.

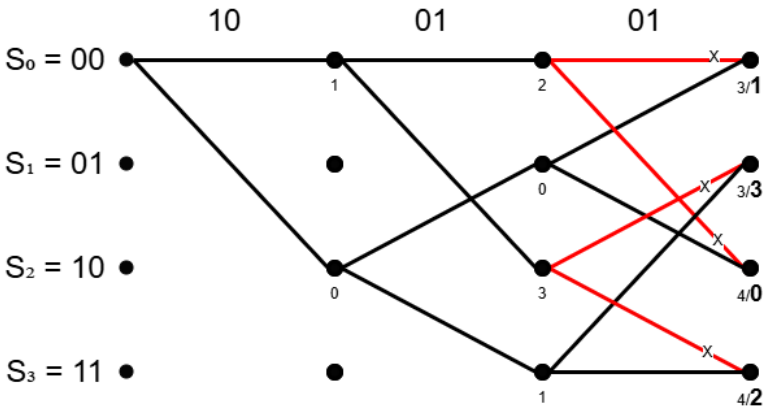


Fonte: Elaborada pelo autor.

Figura 8 – Inicialização - Algoritmo de Viterbi

Computação ao longo da treliça

Neste ponto é importante que tenhamos algumas informações: quais são os percursos sobreviventes até aqui e se as métricas de cada nó estão armazenadas. Então, neste nível, identifique todos os percursos, recalcule as métricas em cada nó somando a métrica dos ramos de entrada na métrica do percurso sobrevivente. Por fim, em cada estado, verifique o caminho com menor valor e descarte os demais.

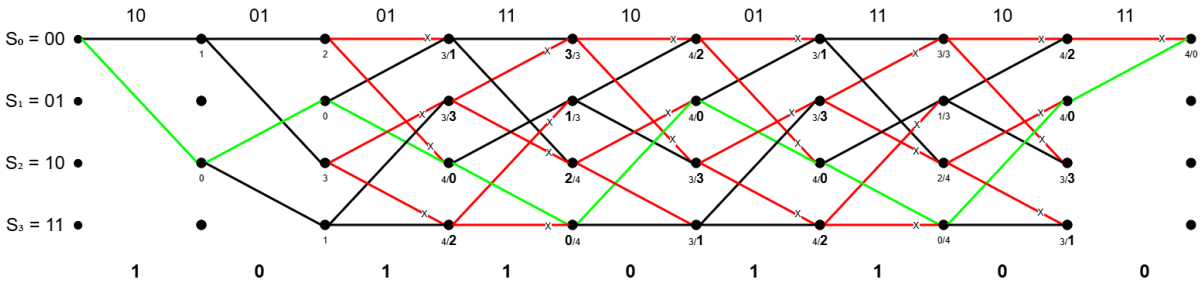


Fonte: Elaborada pelo autor.

Figura 9 – Computação - Algoritmo de Viterbi

Passo Final

Continue com a computação até que o algoritmo finalize sua busca progressiva ao longo da treliça e atinja o nó de finalização (estado 0).



Fonte: Elaborada pelo autor.

Figura 10 – Finalizando - Algoritmo de Viterbi

2.3 Ambientes de Computação Numérica

No contexto de ferramentas computacionais, o `MATLAB` consolidou-se como ambiente padrão para simulação e prototipagem de sistemas de comunicação, oferecendo funções prontas como `vitdec` para decodificação convolucional (MathWorks, 2024c). Por outro lado, o `GNU Octave`, embora compatível com a sintaxe do `MATLAB`, ainda não dispõe de uma implementação nativa da função `vitdec`, o que limita sua adoção em projetos que exigem reprodutibilidade e acesso aberto (SourceForce, 2025).

Comparações entre `MATLAB` e o `GNU Octave`, mostram que, apesar do `GNU Octave` apresentar desempenho inferior em algumas operações, ele se destaca como alternativa viável e livre para pesquisa e ensino, especialmente quando há esforços contínuos para expandir seu conjunto de funções. Porém, ele possui uma lacuna significativa: a ausência de uma implementação nativa de algumas funções, como o `vitdec`. Essa limitação impede que pesquisadores e estudantes que dependem exclusivamente de *software* livre possam replicar experimentos e validar algoritmos sem recorrer a soluções proprietárias.

2.4 Funções Auxiliares

Nesta seção serão apresentadas as funções utilizadas para realizar os testes em conjunto com a função `vitdec`.

2.4.1 Função `poly2trellis`

Função utilizada para gerar a treliça que será utilizada na função `vitdec` (MathWorks, 2024a). Ela possui implementação em ambos os *softwares* comentados até aqui. Possui

dois parâmetros de entrada, sendo eles o comprimento de restrição K e os polinômios geradores do circuito convolucional, que devem ser convertidos para octal. Sua saída gera uma estrutura com os seguintes elementos:

- **`numInputSymbols`**: número de símbolos possíveis na entrada, dado por 2^k;
- **`numOutputSymbols`**: número de símbolos possíveis na saída, dado por 2^n;
- **`numStates`**: quantidade de estados;
- **`nextStates`**: matriz de transição de estados, sendo que a linha indica o estado, e a coluna a entrada.
- **`outputs`**: matriz com as saídas, sendo que a linha indica o estado, e a coluna a entrada.

2.4.2 Função `convenc`

Essa função utiliza códigos convolucionais para realizar a codificação da mensagem (MathWorks, 2024b). A mesma possui implementação tanto no `GNU Octave`, quanto no `MATLAB`. A função possui dois parâmetros de entrada, sendo eles a mensagem a ser codificada e a estrutura gerada pela função `poly2trellis`. A saída nada mais é que a mensagem codificada.

2.5 Função `vitdec`

A função `vitdec`, como apresentada anteriormente, trata-se da implementação do algoritmo de Viterbi no software `MATLAB`. Sua documentação detalha seus parâmetros, além de dar alguns exemplos de uso (MathWorks, 2024c).

Sintaxe

Para implementação dessa função seguimos a documentação citada anteriormente. Tal documento retrata várias variações da função em questão. Para esse projeto, iremos trabalhar com a primeira versão. Essa versão possui cinco parâmetros:

- **`msg`**: mensagem a ser decodificada;
- **`trellis`**: treliça com os parâmetros do codificador convolucional;
- **`tbdepth`**: profundidade de rastreamento;
- **`opmode`**: modo de operação;
- **`dectype`**: tipo de decisão.

2.5.1 Modo de operação - `opmode`

Este parâmetro indica ao decodificador como o codificador operou.

- **`cont`** - modo de operação contínuo, assume-se que ao iniciar a transmissão, as memórias estão zeradas. Um atraso igual ao número `tbdepth` de símbolos de entrada ocorre antes que o primeiro símbolo decodificado apareça na saída. O decodificador irá retroceder a partir do estado com a melhor métrica.
- **`term`** - modo de operação terminado, ou seja, as memórias iniciam zeradas e terminam também zeradas. Este modo não acarreta atraso.
- **`trunc`** - modo de operação truncado, nesse modo entende-se que a codificação começou em zero, porém não se tem certeza que a mesma terminou com a memória zerada.

2.5.2 Profundidade de rastreamento - `tbdepth`

Segundo a documentação da própria função, a variável `tbdepth` representa a profundidade do rastreamento, que influencia o atraso de decodificação. O atraso de decodificação é o número de símbolos zero que precedem o primeiro símbolo decodificado na saída. Em relação aos modo de operação, quando o modo for truncado ou terminado, não existe atraso na decodificação. Sendo assim, a variável deve ser menor ou igual ao tamanho da entrada. Já no modo contínuo, o atraso da decodificação é o número de símbolos de profundidade de rastreamento.

2.5.3 Tipo de decisão - `dectype`

Este parâmetro serve para informar ao decodificador qual tipo de entrada ele irá receber. Existem três tipos de parâmetros:

- `hard` - valores de entrada binários;
- `soft` - valores de entrada no intervalo de 0 a $2^{nsdec-1}$, onde `nsdec` é um valor entre 1 e 13. Nesse caso o algoritmo considera o 0 como sendo o 0 mais confiável e o valor de $2^{nsdec-1}$ como o valor de 1 mais confiável.
- `unquant` - valores numéricos com sinal, onde valores negativos correspondem a um valor lógico 1 e os positivos valor lógico 0;

Segundo (HAYKIN, 2007), os tipos de decisão `soft` e `unquant` podem ter um ganho de desempenho de até 3 dB em comparação ao `hard`. Isso se deve ao fato do tipo de entrada de cada um deles. Os dois primeiros preservam a informação de confiabilidade do

sinal com mais exatidão, porém, com isso, temos uma maior complexidade no decodificador por aceitarem entradas analógicas.

3 DESENVOLVIMENTO

O trabalho propõe o desenvolvimento da função `vitdec`, responsável por decodificar uma mensagem para recuperar a informação transmitida utilizando o algoritmo de Viterbi. Esta função será construída para o `GNU Octave`, *software* livre que não possui essa implementação. O desenvolvimento do trabalho respeitará a documentação da função de mesmo nome do `MATLAB`, *software* que possui direitos autorais e requer licença para o uso do mesmo e do pacote `communications`, que inclui a função em questão. O software `MATLAB` será usado apenas para comparação e análise dos resultados da nova função.

3.1 Desenvolvimento da função `vitdec`

Após verificar a documentação, foi determinado inicialmente que seria desenvolvida apenas a vertente da função utilizando o modo de operação terminado com o tipo de decisão *hard*, o que mais se assemelha com o que é visto em livros.

Além disso, no projeto foi escolhido realizar o armazenamento de todas as informações na memória, priorizando a simplicidade do código em relação à eficiência da memória. Sendo assim, a variável `tbdepth` não é utilizada de fato. Embora a escolha pareça custosa em memória, ela irá garantir a decodificação de máxima verossimilhança.

Na prática é como se a variável estivesse preenchida com o tamanho da informação. A função irá funcionar normalmente mas dependendo do tamanho da informação e da quantidade de memória do codificador, a execução levará mais tempo se comparada ao `MATLAB`.

Iniciando, foi criado um diretório com dez arquivos, o primeiro intitulado `vitdec`.m como exibido no apêndice Código 1. Este é o arquivo responsável por chamar toda a função de fato. Ele foi elaborado da seguinte forma: existe uma estrutura de *switch* com o modo de operação e dentro de cada alternativa existe outro *switch* com o tipo de decisão no codificador. Cada "ramo" irá chamar uma outra função, essa com a devida identificação, `vitdec_opmode_dectype.m`. Apenas a versão `vitdec_term_hard.m` foi elaborada. Essa função possui apenas os três primeiros parâmetros da versão original.

Em seguida foi criada uma outra função, denominada `trellisPreviousStates`, que está exibida no Código 2. Esta função tem um comportamento oposto à propriedade "`nextStates`" da treliça que, ao invés de apontar os próximos estados, indica os estados anteriores.

Concluída a confecção desta última, iniciou-se a implementação da função `vitdec_term_hard.m` apresentada no apêndice Código 3. Para facilitar a compreen-

são, o código foi segmentado em três partes: a primeira sendo a inicialização de métricas e estados, exibida na Código 3.1, a segunda o ciclo de comparação visto no Código 3.2 e, por fim, é apresentado no Código 3.3 a reconstrução da mensagem.

Código 3.1 – vitdec_term_hard_parte1.m

```
function decodedout = vitdec_term_hard(codedin, trellis, tbdepth)
estadoInicial = 1;
estadoFinal = 1;
tamanhoEntradaInd = log2(trellis.numInputSymbols);
tamanhoSaidaInd = log2(trellis.numOutputSymbols);
quantidadeSimbolosSaida = length(codedin)/tamanhoSaidaInd;
tamanhoInfo = quantidadeSimbolosSaida * tamanhoEntradaInd;
tamanhoMem = quantidadeSimbolosSaida + 1;
metricas = inf(trellis.numStates,tamanhoMem);
metricas(estadoInicial,1) = 0;
caminhos = inf(trellis.numStates, 1);
caminhos(estadoInicial,1) = 1;
caminhos_old = caminhos;
infos = reshape(codedin,  tamanhoSaidaInd, [])';
estadoAnteriores = trellisPreviousStates(trellis.nextStates);
```

Na inicialização, o código define o estado inicial e final com 1, já que estamos utilizando o modo de operação terminado, esse valor é definido como 1 pois o `GNU Octave` define a primeira posição de qualquer elemento como 1. Em seguida, são resgatadas algumas informações sobre a mensagem enviada e do codificador como, por exemplo, o tamanho da entrada e saída do codificador, número de símbolos enviados e o tamanho da mensagem recuperada. Por fim, iniciamos a matriz de métricas, responsável por armazenar a distância de Hamming. Como já sabemos que a transmissão iniciou com as memórias vazias, definimos toda matriz como sendo infinita (com exceção da primeira posição, que é 0). Também iniciamos a matriz `caminhos` e `caminhos_old`, que irão guardar o histórico dos estados percorridos. E antes da ida para a próxima etapa é chamada a função `trellisPreviousStates`, função que informa os estados anteriores do codificador e redimensionamos a informação codificada em uma matriz de símbolos.

Código 3.2 – vitdec_term_hard_parte2.m

```
for simbolo = 1:quantidadeSimbolosSaida
  possibilidades = [];
  diferenca = [];
  for entrada = 1:trellis.numStates
    caminhosPoss = [];
    for previousStateInput = 1:trellis.numInputSymbols
      caminhoPredecessor = caminhos(estadoAnteriores(entrada,previousStateInput),1:
    simbolo);
      caminhosPoss = [caminhosPoss; caminhoPredecessor entrada];
```

```
    endfor
    segmentoFinalCaminho = caminhosPoss(:,end-1:end);
    diferencaPasso = [];
    possivelOutput = [];
    for indiceCandidatos = 1:size(segmentoFinalCaminho,1)
      if(segmentoFinalCaminho(indiceCandidatos,1) == Inf)
        diferencaPasso = [diferencaPasso ; Inf];
      else
        indiceInputAssociado = find(trellis.nextStates(segmentoFinalCaminho(
    indiceCandidatos,1),:) == segmentoFinalCaminho(indiceCandidatos,2)-1);
        Output = de2bi(trellis.outputs(segmentoFinalCaminho(indiceCandidatos,1),
    indiceInputAssociado),'left-msb', tamanhoSaidaInd);
        diferencaPasso = [diferencaPasso ; sum(Output ~= infos(simbolo,:)) +
    metricas(segmentoFinalCaminho(indiceCandidatos, 1),simbolo)];
      endif
    endfor
    [~,linhaDifMin] = min(diferencaPasso);
    possibilidades = [possibilidades; caminhosPoss(linhaDifMin,:)];
    diferenca = [diferenca; diferencaPasso(linhaDifMin)];
  endfor
  caminhos_old = caminhos;
  caminhos = possibilidades;
  metricas(:,simbolo+1) = diferenca;
endfor
caminho = caminhos(estadoFinal,:);
decodedout = [];
```

Em seguida, na etapa do ciclo de comparação, percorremos cada símbolo recebido, verificando os possíveis estados atuais e checando os caminhos anteriores até ali, assim percorrendo os possíveis caminhos e calculando as métricas até aquele ponto. Então, realiza-se a comparação e atualiza-se as tabelas de métricas e caminhos.

Código 3.3 – vitdec_term_hard_parte3.m

```
for x = 1:size(caminho,2)-1
  indiceInputRecuperado = find(trellis.nextStates(caminho(x),:) == caminho(x+1)-1);
  bitsDecodificados = de2bi(indiceInputRecuperado - 1, 'left-msb',log2(trellis.
    numInputSymbols));
  decodedout = [decodedout bitsDecodificados];
endfor
```

Após percorrer toda matriz de símbolos, o código vai para a última etapa, escolhendo o caminho de menor custo, que finaliza no estado 0 e, então, percorre esse caminho e retorna a informação decodificada.

4 ANÁLISE DOS RESULTADOS

Para os testes efetuados e descritos nessa seção, utilizamos um *notebook* com sistema operacional `Windows`. Tendo como configuração o processador I5-7200U com arquitetura Dual-Core e 4 Threads, operando a uma frequência base de 2.50 GHz, 8GB de memória RAM, com um SSD de 500GB e uma de placa de video de 2GB de memória. Foram utilizados os softwares MATLAB na versão 2024b e o GNU Octave na versão 10.3.0.

Todos os experimentos e dados que foram utilizados para esta etapa foram documentados e disponibilizados no *GitHub*: <https://github.com/Danehko/octave-communications>.

Após a elaboração da função `vitdec`, realizamos alguns testes de validação. A intenção foi comparar os resultados da função elaborada no `GNU Octave` com a função nativa do `MATLAB`. Nos experimentos aferimos a saída, a mensagem decodificada e o tempo para execução da operação em relação ao `MATLAB`. Dessa forma, escolhemos 5 codificadores que já estavam sendo estudados e suas informações estão na Tabela 2. A escolha se baseou pela diferença no tamanho do comprimento de restrição. Além disso, também foram escolhidos codificadores com mais de um bit de entrada. Foram realizadas quatro simulações para cada codificador, com exceção do primeiro (que foram realizadas apenas duas simulações por ser um codificador mais simples que outros). Nessas variações foram alteradas o tamanho da mensagem, além de inserir erros para acompanhar o comportamento.

Tabela 2 – Detalhes dos codificadores utilizados para teste

Codificador	**Bits de entrada (k)**	**Bits de saída (n)**	**Comprimento de restrição (K)**	**Polinômios geradores**
1	1	2	2	[2 3]
2	1	2	3	[6 7]
3	2	3	2, 2	[2 3 0 ; 1 3 2]
4	1	2	7	[171 133]
5	2	3	5, 4	[23 35 0 ; 0 5 13]

Fonte: Elaborada pelo autor.

Por fim, será realizado um novo teste de desempenho utilizando os codificadores 2, 4 e 5 citados anteriormente. Nesse caso, simularemos uma transmissão por um canal ruidoso *Additive White Gaussian Noise* (AWGN). No experimento, após a mensagem ser codificada, ela será convertida através de uma sinalização polar, onde o bit 0 é representado por -1 e o bit 1 por $+1$. Será adicionado ruído em diferentes níveis de potência, parametrizado pela relação E_b/N_0, para estimar a eficiência da decodificação em termos de taxa de erro de bit *Bit Error Rate* (BER). Assim podemos validar a implementação através da comparação

das curvas de BER vs. E_b/N_0 obtidas pela função elaborada no GNU Octave e pela função nativa do MATLAB, juntamente ao desempenho teórico esperado.

4.1 Resultados

Na primeira série de experimentos, foram comparadas a mensagem de entrada, (após ser codificada e com ruído inserido de forma não aleatória), a saída da função elaborada e o tempo de execução com a função do MATLAB. Foi observado, então, que a função *vitdec* desenvolvida para GNU Octave obteve o mesmo resultado em todos os termos analisados.

A seguir, é exibido o Código 4.1, executado no GNU Octave, aqui a mensagem é gerada randomicamente, codificada e é adicionado um erro na mensagem codificada. Então, utilizamos a função vitdec para decodificar e recuperar a mensagem original. Por fim, é feita a comparação da mensagem inicial com a mensagem decodificada para avaliar se o algoritmo conseguiu recuperar a informação corretamente.

Código 4.1 – experimento4.m

```
clear all;
close all;
clc;

pkg load communications;

addpath('../functions');

rng(76661);
u = randi([0 1],1,30);
M = 7;
u = [u zeros(1,M)];
G = [171 133];

trellis = poly2trellis(M, G);
c = convenc(u, trellis);
%simulando Erro
codedin = c;
codedin(3) = 1 - codedin(3);
tbdepth = length(u);
opmode = 'term';
dectype = 'hard';
decodedout = [];
tic;
decodedout = vitdec(codedin, trellis, tbdepth, opmode, dectype);
tempo = toc;
comparacao = isequal(u,decodedout);
```

Para realizar a comparação da função nativa em MATLAB com a função customizada em GNU Octave, realizamos algumas alterações no código para ser executado no MATLAB. Tais alterações constam no Código 4.2. Primeiramente, colocamos uma verificação do pacote communications do MATLAB que, caso não seja encontrado, aponta um erro. Em seguida, para realizar a comparação, a mensagem deve corresponder à mesma da execução do GNU Octave. Por conta disso, ela é escrita à mão (*hard coded*), e não gerada de maneira aleatória.

Código 4.2 – Diferença do experimento 4 - MATLAB

```
if ~license('test', 'communication_toolbox')
    error('Este script requer o Communications Toolbox do MATLAB.');
end

u = '11110000011010100011111000010000000000';
u = u - '0';
```

Nos Código 4.3 e Código 4.4 são exibidos os dados retirados da execução do experimento do Código 4.1, sendo respectivamente os dados da execução no GNU Octave e no MATLAB. Foram coletadas a mensagem transmitida (u), a mensagem codificada (c), a mensagem codificada com a inserção do erro (*codedin*), a mensagem decodificada (*decodedout*), a comparação entre u e *decodedout* (*comparacao*) e o tempo de execução (*tempo*). Nas imagens notamos que todos os dados são iguais com exceção do tempo de execução, comprovando assim o funcionamento da função.

Código 4.3 – Resultados do experimento4 executado no GNU Octave

```
Executado no GNU Octave
u = 11110000011010100011111000010000000000
c = 11011001100101101000010111010111001000000101011110011001011111000111000000
codedin =
    11111001100101101000010111010111001000000101011110011001011111000111000000
decodedout = 11110000011010100011111000010000000000
comparacao = 1
tempo = 1.6996
```

Código 4.4 – Resultados do experimento4 executado no *MATLAB*

```
Executado no Matlab
u = 11110000011010100011111000010000000000
c = 11011001100101101000010111010111001000000101011110011001011111000111000000
codedin =
    11111001100101101000010111010111001000000101011110011001011111000111000000
decodedout = 11110000011010100011111000010000000000
```

```
comparacao = 1
tempo = 0.044470
```

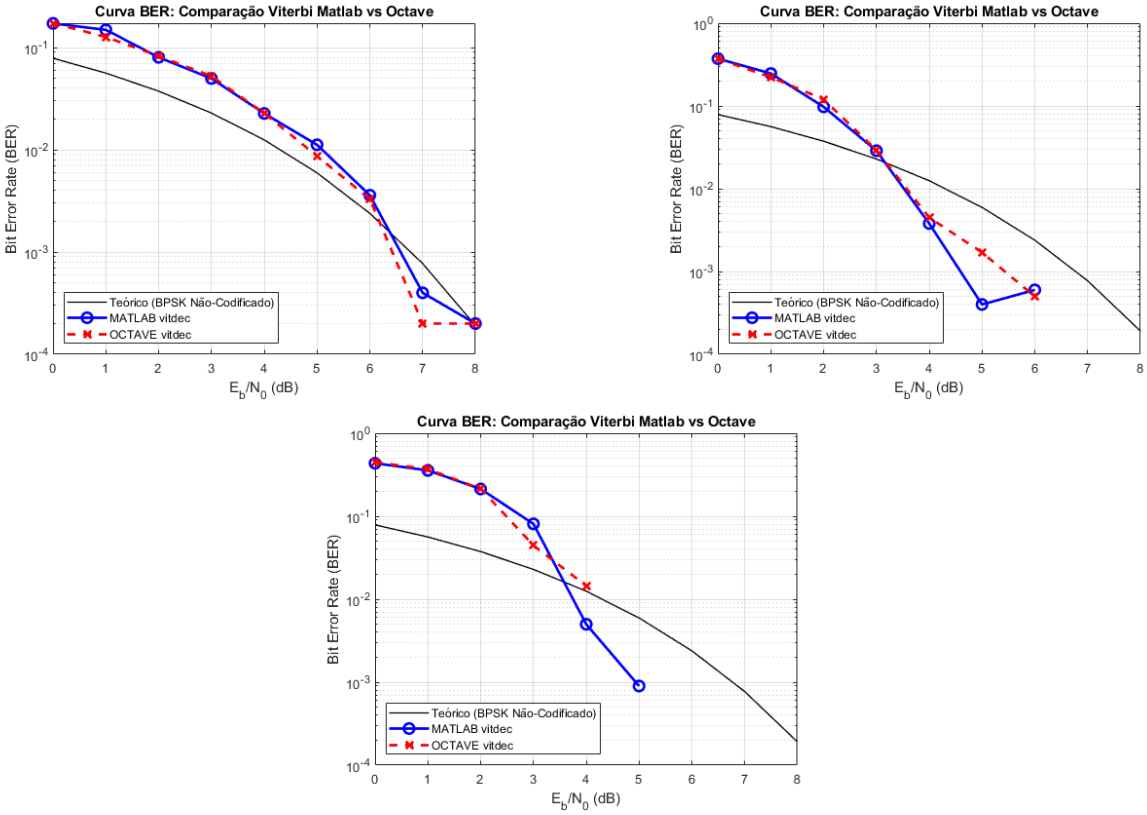
Na Tabela 3 é exibido uma comparação, utilizando os codificadores descritos anteriormente na Tabela 2. Foi transmitido a mesma mensagem, porém foi inserindo erros nesta de acordo com o comprimento de restrição de cada codificador para validar a recuperação da mensagem. A execução foi satisfatória, conseguindo recuperar a informação transmitida inicialmente. Porém um ponto chamou bastante atenção o tempo de computação. A função nativa do `MATLAB` se mostrou muito mais eficiente, executando os experimentos em cerca de um segundo. A performance do `MATLAB` é obtida graças ao fato da função `vitdec` ser compilada em C. Já a função desenvolvida nesse trabalho foi implementada no próprio `GNU Octave` e, por tratar-se de um arquivo do tipo .m, precisa ser interpretada pelo *software*. Um outro ponto a ser considerado é que a função desenvolvida não foi otimizada ao máximo, podendo ganhar algum desempenho ao realizar alguns ajustes.

Tabela 3 – Comparação entre a função desenvolvida com a função `vitdec` nativa do `MATLAB`

Codificador	**Tamanho da mensagem (bit)**	**Erros introduzidos**	**Tempo Vitdec (`MATLAB`)**	**Tempo função implementada**
1	10002	1000	$0.018369s$	$31.2215s$
2	10003	667	$0.014514s$	$45.7814s$
3	10004	500	$0.014542s$	$41.03s$
4	10007	286	$0.056532s$	$1122.82s$
5	10010	223	$0.043145s$	$1581.94s$

Fonte: Elaborada pelo autor.

Os resultados do segundo experimento estão exibidos na Figura 11. Ao analisarmos os gráficos percebemos que, apesar da pequena variação após a marca de 6dB, devido a uma flutuação estatística, o comportamento da função `vitdec` elaborada para `GNU Octave` tem um desempenho muito próximo ao da função `vitdec` nativa do `MATLAB`. Com isso, certificamos que a função elaborada funciona perfeitamente.



Fonte: Elaborada pelo autor.

Figura 11 – Curvas BER vs. E_b/N_0 dos experimentos 2, 4 e 5

5 CONCLUSÃO

O principal objetivo deste estudo foi desenvolver e disponibilizar a função `vitdec`, função essa baseada no algoritmo de Viterbi, para um ambiente de software livre como o `GNU Octave`. Essa implementação contribui para o desenvolvimento do ensino e pesquisa em telecomunicações.

A primeira etapa deste projeto foi a realização de pesquisa e estudos teóricos sobre códigos convolucionais e algoritmo de Viterbi. Com esta etapa finalizada, iniciou-se o desenvolvimento da variante do método `vitdec` com modo de operação terminado, ou seja, a transmissão começa e termina com os registradores zerados e com tipo de decisão *hard*, seguindo a especificação de parâmetros da documentação do software do `MATLAB`.

A função desenvolvida se mostrou funcional e obteve resultados equivalentes à função da ferramenta `MATLAB`, com exceção do tempo de execução, onde a função do `MATLAB` é muito rápida. A decisão de projeto de simplificar o código, não utilizando a variável `tbdepth`, é justificada tendo em vista garantir a máxima verossimilhança na decodificação, priorizando simplicidade de implementação ao invés de desempenho.

Ao analisar os resultados dos testes, observou-se que a função teve um tempo de execução maior em comparação à função do `MATLAB`, devido ao fato da primeira ser implementada na ferramenta `GNU Octave`, enquanto a função nativa do `MATLAB` é compilada em C e otimizada para tal linguagem. Em relação aos resultados, a função desenvolvida neste trabalho obteve os mesmos resultados, validando sua utilização no ambiente de livre acesso (`GNU Octave`).

Sendo assim, foi visto que a função desenvolvida mostrou-se funcional, e está disponível para a comunidade científica e acadêmica de forma gratuita no repositório, a fim de proporcionar seu livre uso e possíveis melhorias.

5.1 Trabalhos futuros

Tendo em vista as escolhas desse trabalho e levando em consideração a conclusão, fica como sugestão para trabalhos futuros o desenvolvimento das outras vertentes da função `vitdec`. A elaboração atual é modular e foi pensada para esse propósito. É sugerido, então, que trabalhos futuros iniciem com alterações apenas no tipo de decisão do decodificador, pois parte da função desenvolvida pode ser aproveitada.

Além disso, também é proposto uma otimização da versão desenvolvida nesse trabalho, por exemplo desenvolvê-la em C, C++ ou Fortran, com o intuito de melhorar o

desempenho.

Outro ponto a ser trabalhado é o uso da variável `tbdepth`, refinando assim o gerenciamento de memória através da técnica de janela deslizante, que reduzirá o consumo de memória e processamento.

Por fim, outra sugestão de trabalho futuro é o desenvolvimento de outras funções desse módulo, como as funções `bchenc` e `bchdec`, ou de outro pacote que ainda não tenha uma implementação no `GNU Octave`. Quanto maior for o acesso às funções como o `vitdec`, por exemplo, maior será a possibilidade de replicações por parte da sociedade acadêmica e melhor será o entendimento dos estudantes.

REFERÊNCIAS

HAYKIN, S. *Sistemas de Comunicação Analógicos e Digitais*. 4. ed. São Paulo: Bookman, 2007. 683–693 p. 25, 27, 28, 31, 36

MathWorks. *convenc: Convolutionally encode binary message*. 2024. Acesso em: 01 Dex. 2025. Disponível em: <https://www.mathworks.com/help/comm/ref/convenc.html>. 34

MathWorks. *poly2trellis: Convert convolutional code polynomials to trellis description*. 2024. Acesso em: 01 Dez. 2025. Disponível em: <https://www.mathworks.com/help/comm/ref/poly2trellis.html>. 35

MathWorks. *vitdec: Convolutionally decode binary data by using Viterbi algorithm*. 2024. Acesso em: 22 abr. 2024. Disponível em: <https://www.mathworks.com/help/comm/ref/vitdec.html>. 21, 34, 35

SHPIGELBLAT, S. *Implementing the Viterbi Algorithm in Today's Digital Communications Systems*. 2009. <https://www.design-reuse.com/articles/21107/viterbi-algorithm.html>. Acesso em: 22 abr. 2025. 21

SourceForce. *Octave Forge communications - A collection of packages providing extra functionality for GNU Octave*. 2025. Acesso em: 12 Dez. 2025. Disponível em: <https://sourceforge.net/p/octave/communications/ci/default/tree/>. 34

Apêndices

Código 1 – vitdec.m

```
function decodedout = vitdec(codedin, trellis, tbdepth, opmode, dectype)
  switch opmode
    case 'term'
      switch dectype
        case 'soft'
          disp('Ainda nao implementado');
##          decodedout = vitdec_term_soft(codedin, trellis, tbdepth)
        case 'unquant'
          disp('Ainda nao implementado');
##          decodedout = vitdec_term_unquant(codedin, trellis, tbdepth)
        case 'hard'
          decodedout = vitdec_term_hard(codedin, trellis, tbdepth);
        end
    case 'cont'
      switch dectype
        case 'soft'
          disp('Ainda nao implementado');
##          decodedout = vitdec_cont_soft(codedin, trellis, tbdepth)
        case 'unquant'
          disp('Ainda nao implementado');
##          decodedout = vitdec_cont_unquant(codedin, trellis, tbdepth)
        case 'hard'
          disp('Ainda nao implementado');
##          decodedout = vitdec_cont_hard(codedin, trellis, tbdepth)
        end
    case 'trunc'
      switch dectype
        case 'soft'
          disp('Ainda nao implementado');
##          decodedout = vitdec_trunc_soft(codedin, trellis, tbdepth)
        case 'unquant'
          disp('Ainda nao implementado');
##          decodedout = vitdec_trunc_unquant(codedin, trellis, tbdepth)
        case 'hard'
          disp('Ainda nao implementado');
##          decodedout = vitdec_trunc_hard(codedin, trellis, tbdepth)
        end
    end
end
```

Código 2 – trellisPreviousStates.m

```
function out = trellisPreviousStates(nextStates)
min = min(reshape(nextStates,1,[]));
max = max(reshape(nextStates,1,[]));
out = zeros(size(nextStates));
```

```
for state = min:1:max
  linha = [];
  for posicaoNextStates = 1:numel(nextStates)
    if(state == nextStates(posicaoNextStates))
      posicaoAparicao = mod(posicaoNextStates,size(nextStates,1));
      if(posicaoAparicao == 0)
        linha = [linha size(nextStates,1)];
      else
        linha = [linha posicaoAparicao];
      endif
    endif
  endfor
  out(state+1,:) = sort(linha);
endfor
end
```

Código 3 – vitdecTermHard.m

```
function decodedout = vitdec_term_hard(codedin, trellis, tbdepth)
estadoInicial = 1;
estadoFinal = 1;
tamanhoEntradaInd = log2(trellis.numInputSymbols);
tamanhoSaidaInd = log2(trellis.numOutputSymbols);
quantidadeSimbolosSaida = length(codedin)/tamanhoSaidaInd;
tamanhoInfo = quantidadeSimbolosSaida * tamanhoEntradaInd;
tamanhoMem = quantidadeSimbolosSaida + 1;
metricas = inf(trellis.numStates,tamanhoMem);
metricas(estadoInicial,1) = 0;
caminhos = inf(trellis.numStates, 1);
caminhos(estadoInicial,1) = 1;
caminhos_old = caminhos;
infos = reshape(codedin,  tamanhoSaidaInd, [])';
estadoAnteriores = trellisPreviousStates(trellis.nextStates);
for simbolo = 1:quantidadeSimbolosSaida
  possibilidades = [];
  diferenca = [];
  for entrada = 1:trellis.numStates
    caminhosPoss = [];
    for previousStateInput = 1:trellis.numInputSymbols
      caminhoPredecessor = caminhos(estadoAnteriores(entrada,previousStateInput),1:
    simbolo);
      caminhosPoss = [caminhosPoss; caminhoPredecessor entrada];
    endfor
    segmentoFinalCaminho = caminhosPoss(:,end-1:end);
    diferencaPasso = [];
    possivelOutput = [];
    for indiceCandidatos = 1:size(segmentoFinalCaminho,1)
      if(segmentoFinalCaminho(indiceCandidatos,1) == Inf)
```

```
30          diferencaPasso = [diferencaPasso ; Inf];
31        else
32          indiceInputAssociado = find(trellis.nextStates(segmentoFinalCaminho(
      indiceCandidatos,1),:) == segmentoFinalCaminho(indiceCandidatos,2)-1);
33          Output = de2bi(trellis.outputs(segmentoFinalCaminho(indiceCandidatos,1),
      indiceInputAssociado),'left-msb', tamanhoSaidaInd);
34          diferencaPasso = [diferencaPasso ; sum(Output ~= infos(simbolo,:)) +
      metricas(segmentoFinalCaminho(indiceCandidatos, 1),simbolo)];
35        endif
36      endfor
37      [~,linhaDifMin] = min(diferencaPasso);
38      possibilidades = [possibilidades; caminhosPoss(linhaDifMin,:)];
39      diferenca = [diferenca; diferencaPasso(linhaDifMin)];
40    endfor
41    caminhos_old = caminhos;
42    caminhos = possibilidades;
43    metricas(:,simbolo+1) = diferenca;
44  endfor
45  caminho = caminhos(estadoFinal,:);
46  decodedout = [];
47
48  for x = 1:size(caminho,2)-1
49    indiceInputRecuperado = find(trellis.nextStates(caminho(x),:) == caminho(x+1)-1);
50    bitsDecodificados = de2bi(indiceInputRecuperado - 1, 'left-msb',log2(trellis.
        numInputSymbols));
51    decodedout = [decodedout bitsDecodificados];
52  endfor
```