



UMA BIBLIOTECA DIDÁTICA PRELIMINAR EM PYTHONSCAD PARA DESENHO E CÁLCULO DE TRELIÇAS BIDIMENSIONAIS

A PRELIMINARY EDUCATIONAL LIBRARY IN PYTHONSCAD FOR DRAWING AND ANALYSIS OF TWO-DIMENSIONAL TRUSSES

Vívia Velloso Fontes¹, Gustavo Koury Costa²

DOI: 10.37702/REE2236-0158.v45p708-738.2026

RESUMO: Este artigo descreve o desenvolvimento de uma biblioteca computacional didática preliminar em PythonSCAD para automatizar o desenho, o cálculo, a análise e a visualização de treliças planas. O estudo enfatiza a importância da combinação entre os recursos gráficos do OpenSCAD e a flexibilidade da programação em Python, superando as limitações numéricas do primeiro. A metodologia inclui algoritmos para a geração automática da geometria da treliça, a aplicação de carregamentos e restrições e a resolução do sistema de equações de equilíbrio nodal pelo método de eliminação de Gauss. Uma biblioteca computacional voltada às treliças do tipo Warren foi desenvolvida e disponibilizada ao público, possibilitando o desenho e o cálculo dos esforços internos por meio de comandos simples em código. Ao final do artigo, são apresentados três exemplos destinados à validação numérica da biblioteca.

PALAVRAS-CHAVE: treliças; PythonSCAD; OpenSCAD.

ABSTRACT: This article describes the creation of a preliminary educational computational library in PythonSCAD to automate the drawing, analysis, and visualization of planar trusses. The study emphasizes the importance of combining OpenSCAD's graphical capabilities with the flexibility of Python programming, thereby overcoming the former's numerical limitations. The methodology includes algorithms for the automatic generation of truss geometry, application of loads and boundary conditions, and solution of the nodal equilibrium equations system using the Gaussian elimination method. A computational library specifically designed for Warren-type trusses is then developed and made available to the public, allowing the drawing and calculation of internal forces through simple lines of code. At the end of the article, three examples are presented for the numerical validation of the library.

KEYWORDS: trusses; PythonSCAD; OpenSCAD.

¹ Estudante do Curso Técnico em Edificações, Instituto Federal de Pernambuco – Recife, viviavf@outlook.com

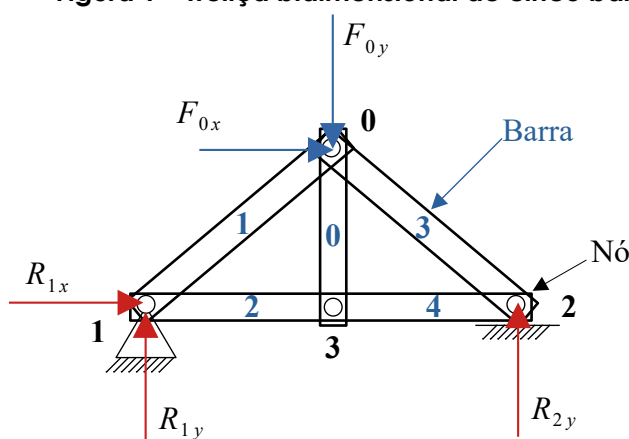
² Professor, Coordenação de Mecânica, Instituto Federal de Pernambuco – Recife, gustavokoury@recife.ifpe.edu.br

INTRODUÇÃO

Treliças são estruturas estáticas, geralmente metálicas, construídas para suportar cargas por meio de um conjunto de barras interligadas, atuando sob esforços de tração ou compressão. O cálculo dos esforços individuais nas barras se reduz à solução de equações de equilíbrio de forças, que pode ser efetuado em cada junção entre barras (nós). O esforço computacional cresce à medida que o número de equações aumenta, com o aumento de nós da treliça. O caso bidimensional, no qual as barras da treliça estão dispostas em um plano, constitui uma simplificação do problema. Entretanto, aplicações práticas são comumente encontradas em situações reais, o que é um incentivo ao estudante do Ensino Técnico, Tecnológico e de Engenharia (Farias, Jesus e Oliveira, 2020).

A Figura 1, a seguir, ilustra uma treliça plana, em que os nós e as barras são numerados aleatoriamente, de 0 a 3 e de 0 a 4. A treliça é estaticamente carregada pelas forças F_{0x} e F_{0y} (representadas por setas azuis), o que, por sua vez, produz três reações desconhecidas, R_{1x} , R_{1y} e R_{2y} (em vermelho) nos apoios dos nós 1 e 2. Esses tipos de apoio (rótula no nó 1 e deslizante no nó 2) resultarão em um sistema estaticamente determinado, que é o objetivo deste artigo.

Figura 1 – Treliça bidimensional de cinco barras



Fonte: elaborada pelos autores.

Vários trabalhos científicos têm explorado o dimensionamento de treliças bidimensionais, nas quais a relação tensão-deformação (Lei de Hooke) permite a formulação da matriz de rigidez da estrutura. A partir de uma análise estática, as forças e as deformações nas barras são, geralmente, obtidas por meio de métodos numéricos. Nesse contexto, o Método dos Elementos Finitos é, certamente, um dos métodos mais empregados (ver, por exemplo, Barrozo *et al.*, 2019; Delgado Jr. *et*



al., 2022). Entretanto, o desenho e o cálculo dos esforços nem sempre são realizados em um mesmo ambiente computacional e, quando isso ocorre, os códigos-fonte podem não estar disponíveis ao usuário final, como, por exemplo, no trabalho de Yang, Li e Sun (2023). Isso se deve, em parte, ao fato de que a criação de um programa de computador que, ao mesmo tempo, permita a visualização da treliça e o cálculo dos esforços não constitui uma tarefa trivial. Linguagens de programação tradicionais, como C e Pascal, não oferecem, nativamente, um ambiente voltado à elaboração de desenhos. Por outro lado, aplicativos gráficos, como o AutoCAD³, não são facilmente programáveis, além de não serem gratuitos.

Um aplicativo de desenho recente e totalmente programável, chamado OpenSCAD⁴, tem se consolidado como uma linguagem de programação voltada à criação de desenhos complexos e dotada de uma boa biblioteca de comandos e funções, bastante semelhantes às linguagens tradicionais. A linguagem de programação do OpenSCAD possui estrutura avançada e permite a criação de *loops*, sub-rotinas e funções, tal como ocorre nas linguagens tradicionais, além de estar disponível gratuitamente para ambientes Windows, Mac e Linux, sem necessitar de grandes requisitos de máquina (menos de 50Mb ocupados em disco após instalação no Windows 11, por exemplo). O impacto desse *software* de CAD foi tamanho que uma sequência de livros foi publicada, explorando o seu uso no ensino de Matemática e Física (Horvath e Cameron, 2016; 2017; 2021; 2022; e 2023).

Apesar da grande versatilidade introduzida pelo OpenSCAD, toda a literatura disponível, incluindo os livros mencionados e alguns trabalhos científicos (ver, por exemplo, Braga, 2023), menciona apenas a criação de formas geométricas para impressão em 3D, com fins pedagógicos. Os autores deste trabalho desconhecem qualquer publicação, até o presente momento, que utilize o OpenSCAD como instrumento de visualização e auxílio em projetos de Engenharia, e, em particular, utilizando treliças. Isso, muito provavelmente, deve-se ao fato de o OpenSCAD não seguir o mesmo paradigma das linguagens de programação usuais, não permitindo que valores de variáveis sejam alterados, tornando impossível a realização, por exemplo, de cálculos iterativos.

Uma tentativa de solucionar o problema surgiu recentemente com um novo ambiente de programação que permite a criação de códigos em linguagem Python, com funcionalidades semelhantes às do OpenSCAD. O PythonSCAD⁵ é um aplicativo gratuito com interface muito parecida com a do OpenSCAD, que une o

³ Autocad é um aplicativo comercial produzido pela Autodesk (<https://www.autodesk.com> – mai, 2026).

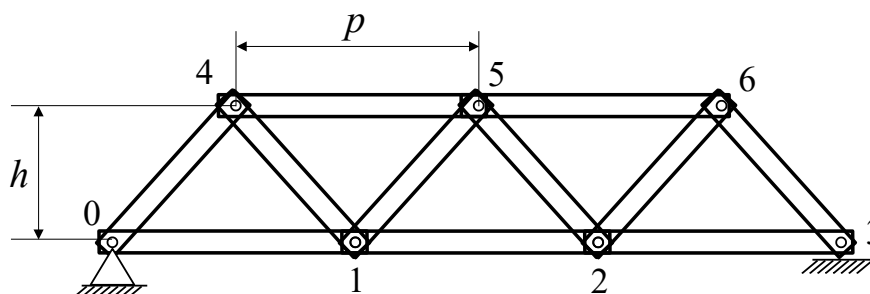
⁴ <https://openscad.org> (mai, 2026)

⁵ <https://pythonscad.org> (mai, 2026)

melhor dos dois mundos: Python e OpenSCAD, e será o ambiente no qual os algoritmos propostos neste artigo serão desenvolvidos.

O presente trabalho visa à criação de uma "biblioteca" computacional em PythonSCAD para ser utilizada no desenho e no cálculo dos esforços nas barras de treliças; em particular, nas treliças do tipo Warren (Ernst, 2023), semelhantes à ilustrada na Figura 2, visando a sua utilização como ferramenta didática. Três estudos de caso serão apresentados ao final do artigo, nos quais os resultados serão comparados, para fins de validação do código computacional. A biblioteca, cujo código-fonte é disponibilizado no Apêndice, é preliminar e embrionária, podendo ser modificada livremente pelo usuário. Ressaltamos que o presente trabalho trata apenas do cálculo dos esforços nas barras de treliças bidimensionais, assumindo que estas são elementos rígidos (sem deformação), e desprezando-se qualquer momento resistente nas interligações. Assim, o código computacional destina-se apenas a fins didáticos, podendo ser futuramente modificado para a utilização em estruturas reais, por se tratar de um código computacional aberto (ver Apêndice).

Figura 2 – Treliça Warren (detalhe do passo p e altura, h)



Fonte: elaborada pelos autores.

Antes de dar prosseguimento com o desenvolvimento dos algoritmos, é importante traçar algumas considerações iniciais sobre as linguagens de programação OpenSCAD e PythonSCAD.

OPENSCAD E PYTHONSCAD

Inicia-se esta seção ilustrando como a criação codificada de desenhos em OpenSCAD é extremamente simples. Por exemplo, a linha de código a seguir produz um retângulo de largura igual a 2 unidades e altura igual a 6 unidades no plano XY, de tal forma que o ponto (0,0) coincida com o canto inferior esquerdo

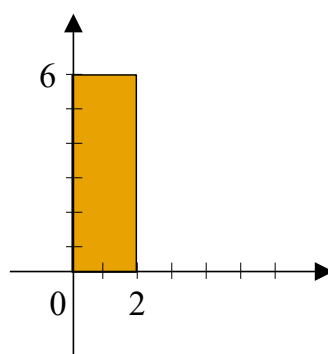


do retângulo (as palavras em **negrito** indicam comandos e funções pertencentes à linguagem).

```
square([2, 6], center = false);
```

O resultado da execução desse código é ilustrado na Figura 3.

Figura 3 – Barra retangular criada no OpenSCAD



Fonte: elaborada pelos autores.

Por meio da programação em OpenSCAD, é possível rotacionar o retângulo da Figura 3 e até produzir animações em tela. Por exemplo, o *script* a seguir ilustra a rotação da barra em torno do ponto (0,0), utilizando parâmetros definidos nas variáveis iniciadas "L", "H" e "Angz" (largura, altura e ângulo de rotação em torno do eixo perpendicular ao plano do papel, respectivamente).

```
L = 2;
```

```
H = 6;
```

```
Az = 45;
```

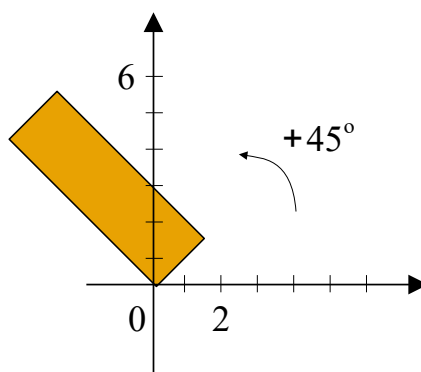
```
rotate([0,0,Az])
```

```
square([L, H], center = false);
```



A Figura 4 ilustra a execução do código anterior.

Figura 4 – Barra retangular rotacionada em 45°



Fonte: elaborada pelos autores.

Apesar da enorme facilidade com que o OpenSCAD permite codificar e exibir desenhos, a linguagem não permite que valores de variáveis sejam alterados. Assim, o seguinte código computacional não seria válido em OpenSCAD:

```
X = 3;
X = X+1; // Operação não permitida!!!
```

A limitação exemplificada no código anterior torna impossível a realização de procedimentos iterativos, necessários para a resolução numérica de equações. Porém, com pequenas modificações no *script* em OpenSCAD, é possível produzir os mesmos resultados em PythonSCAD. Por exemplo, o retângulo rotacionado da Figura 4 poderia ter sido criado com as seguintes linhas de programação em PythonSCAD:

```
from openscad import *
L = 2
H = 6
Az = 45
ret = square([L, H])
rot_ret = ret.rotate([0,0,Az])
rot_ret.show()
```

ou, de maneira simplificada:



```
from openscad import *  
L= 2  
H = 6  
Az = 45  
ret = square([L, H])  
show(ret.rotate([0,0,Az]))
```

Nos dois exemplos anteriores, a principal diferença entre os códigos escritos em OpenSCAD e em PythonSCAD é a necessidade de criação de objetos. Por exemplo, no primeiro dos dois códigos apresentados em PythonSCAD, inicia-se com a criação do objeto “ret” (retângulo). A partir da criação desse objeto, o método “rotate” é chamado e um novo objeto, “rot_ret” (retângulo rotacionado), é criado. Finalmente, o método “show” é invocado pelo objeto “rot_ret”.

A criação de objetos e o consecutivo chamamento de métodos por esses objetos criados são características das linguagens orientadas a objetos, das quais Python constitui um exemplo clássico. Uma grande vantagem da linguagem PythonSCAD sobre o OpenSCAD é que os processos de construção são divididos em pequenas etapas, tornando o código mais claro, ainda que possa, eventualmente, apresentar mais linhas de programação.

Nas próximas seções, serão apresentados os algoritmos necessários para a resolução da treliça Warren (Figura 2). Os seguintes passos serão seguidos, na ordem:

1. Geração dos nós e dos elementos.
2. Desenho da treliça.
3. Carregamento externo e determinação dos tipos de apoio.
4. Cálculo dos esforços individuais nas barras.

Para cada uma das etapas descritas, será proposto um algoritmo, seguido da explicação do *script* em PythonSCAD.

GERAÇÃO DOS NÓS E DOS ELEMENTOS DA TRELIÇA WARREN

No algoritmo proposto neste artigo, as treliças Warren são criadas a partir do nó (0,0), localizado no canto inferior esquerdo, utilizando-se um algoritmo bastante



simples, em que as posições vetoriais dos nós nas barras inferiores e superiores, $\vec{r}_{\text{inf}}$ e $\vec{r}_{\text{sup}}$ são obtidas por meio das seguintes equações:

$$\begin{cases} \vec{r}_{\text{inf}} = (i \cdot p, 0) \\ \vec{r}_{\text{sup}} = (j \cdot p + \frac{p}{2}, h) \end{cases} \quad \text{Eq. [1]}$$

em que $i = 0, 1, 2 \dots N$ e $j = 0, 1, 2 \dots N - 1$, sendo N o número de vãos horizontais inferiores ($N = 3$, na Figura 2, na qual os vãos são definidos pelas barras 0-1, 1-2 e 2-3, respectivamente); p e h , são o passo e a altura da treliça (Figura 2).

A Eq. [1] pode ser implementada em PythonSCAD por meio do *script* a seguir:

```
def genos_wr(p,h,N):
no_in = [[i*p, 0] for i in range(N+1)]
no_sup = [[(j + 1/2)*p, h] for j in range(N)]
return no_in + no_sup
```

em que o comando "**for i in range(x)**" repete o *loop* " x " vezes, a partir de " $i=0$ ". Por exemplo, para $x = 3$, " i " armazenará, consecutivamente, os valores 0, 1 e 2, dentro do *loop*. O mesmo raciocínio se aplica para o índice " j ".

O código computacional descrito anteriormente produz um *array*⁶ ("nodes"), contendo as coordenadas dos nós da treliça Warren. Observa-se que o preenchimento de "nodes" inicia-se no canto inferior esquerdo (nó "0"), e segue até o canto superior direito. No caso da Figura 2, por exemplo, o último elemento ocupará a posição 6 (nó "6"). Particularmente, se $p = 2$ e $h = 1$, o *array* "nos", gerado por "`nos = genos_wr(2,1,3)`" será:

```
nos = [[0, 0], [2, 0], [4, 0], [6, 0], [1, 1], [3, 1], [5, 1]]
```

A fim de complementar o arcabouço básico da treliça, faz-se necessária a criação de um outro *array* contendo informações sobre as barras da treliça. Cada elemento (barra) da treliça é unicamente determinado pelos nós em cada um dos seus pontos extremos. Assim, um possível *array* de elementos para a treliça da Figura 1, por exemplo, pode ser descrito em linguagem PythonSCAD como:

```
elm = [[0,3], [0,1], [3,1], [2,0], [2,3]]
```

⁶ Array é o termo computacional (em inglês) para uma variável contendo lista de elementos.



Os elementos dos *arrays* em PythonSCAD são numerados a partir do índice "0". Assim, o elemento 2 de "elm" possui conectividades 3 e 1, ou seja, $\text{elm}[2] = [3,1]$. O termo "conectividades" se refere aos nós nas extremidades dos elementos. Perceba-se que a ordem das conectividades em cada elemento de "elm" pode ser alterada, bem como a ordem dos elementos.

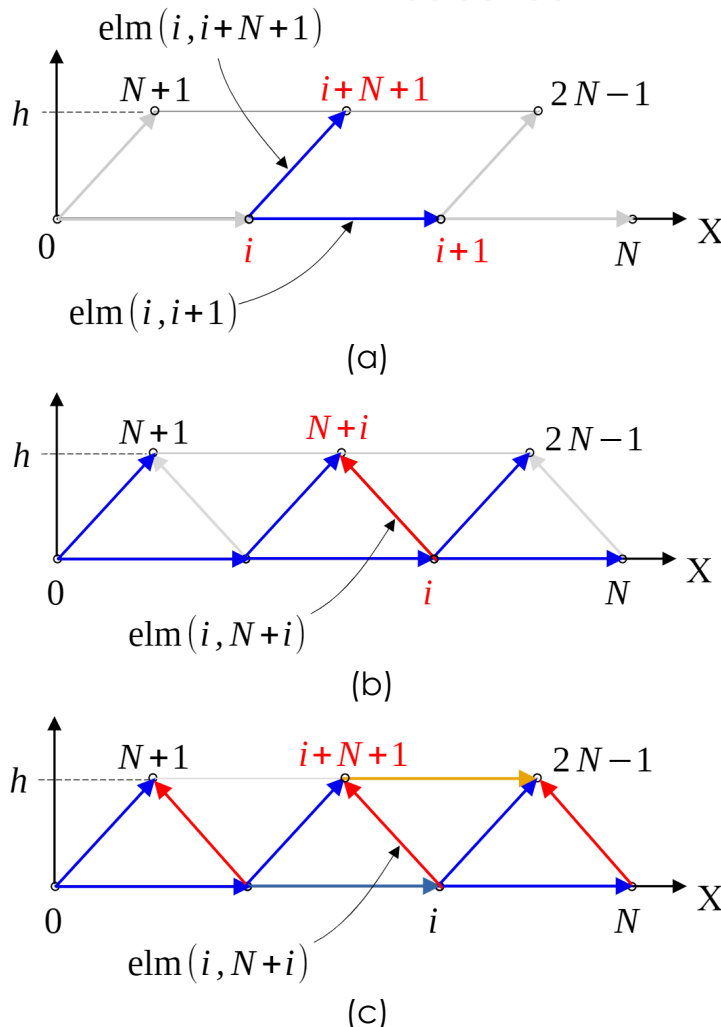
Para se criar um *array* contendo os elementos de uma treliça plana, propõe-se um algoritmo elementar, restrito às treliças Warren. Outras configurações de treliças podem apresentar dificuldades para criação de uma "malha triangular" automática. O algoritmo proposto neste trabalho explora o fato de que, nas treliças Warren, os nós são criados em duas fileiras, sendo a primeira composta por nós inferiores e a segunda composta por nós superiores.

A Figura 5 ajuda a compreender o algoritmo proposto. Os primeiros N nós do *array* "nodes", criados pela função "genos_wr", estão nas barras inferiores, situadas sobre o eixo X . Os nós de índice $N+1$ até $2N-1$ pertencem às barras superiores, situados sobre a reta $y = h$. O algoritmo de criação dos elementos é como se segue.

A partir do nó "0" até o nó " $N-1$ ", criam-se dois elementos orientados: um deles parte do nó " i " e segue até o nó " $i+1$ " (nós sucessivos sobre o eixo X), enquanto o outro parte do nó " i " e segue até o nó " $i+N+1$ " (nó imediatamente à direita do nó " i ", situado sobre a reta na reta $y = h$). Criam-se, então, os primeiros elementos, ilustrados pelas setas azuis na Figura 5 (a). Em seguida, executa-se um segundo *loop*, iniciado a partir do nó "1" (não mostrado na figura) até o N -ésimo nó, para a criação dos elementos entre o i -ésimo nó, no eixo X , e o nó superior, $N+i$, imediatamente à esquerda do nó " i ", sobre a reta $y = h$ (segmento de reta de cor vermelha na Figura 5 (b)). Finalmente, um terceiro *loop*, entre os nós $i+N+1$ e $i+N+2$, cria os elementos horizontais ao longo da reta $y = h$ (seta laranja na Figura 5 (c)). Em particular, o nó $i+N+2$ não é mostrado na Figura 5 (c), uma vez que coincide com o último nó, $2N-1$, da treliça.

Figura 5 – Representação gráfica do algoritmo de criação do array “elm” em uma treliça Warren.

Passos sucessivos (a), (b) e (c)



Fonte: elaborada pelos autores.

A seguinte função em PythonSCAD implementa o algoritmo descrito anteriormente:

```
def gelm_wr(N):
    elA = [[i, i+1] for i in range(N)]
    elB = [[i, i+N+1] for i in range(N)]
    elC = [[i, N+i] for i in range(1, N+1)]
    elD = [[i, i+1] for i in range(N+1, 2*N)]
    return elA + elB + elC + elD
```

Como exemplo, o array “el”, obtido por meio da chamada à função “el=gelm_wr(3)”, será (Figura 2):



$el = [[0, 1], [1, 2], [2, 3], [0, 4], [1, 5], [2, 6], [1, 4], [2, 5], [3, 6], [4, 5], [5, 6]]$

Observação: os usuários da biblioteca proposta neste trabalho podem optar por criar os *arrays* de nós e de elementos manualmente, sem a utilização dos geradores "genos_wr" e "gelm_wr", desde que sigam o mesmo padrão de preenchimento. Ao final deste artigo, será apresentado um estudo de caso que explora esse assunto.

DESENHO DA TRELIÇA

Observa-se, primeiramente, que as linguagens PythonSCAD e OpenSCAD não permitem desenhos planos, de tal forma que as barras da treliça são desenhadas como paralelepípedos. Dessa forma, a função interna "**square**", utilizada na criação das Figuras 3 e 4, cria paralelepípedos de profundidade padrão "1", ao longo do eixo perpendicular ao plano da figura (eixo *Z*).

Em segundo lugar, observa-se que não é possível inserir unidades diretamente no PythonSCAD. Entretanto, para fins de impressão 3D, o PythonSCAD assume, por *default*, o milímetro (mm) como unidade de medida. Assim, sem perda de generalidade, neste trabalho, será utilizado o milímetro (mm) como unidade de medida para o desenho das barras da treliça, o que facilitará a impressão 3D em uma eventual criação de um modelo didático físico.

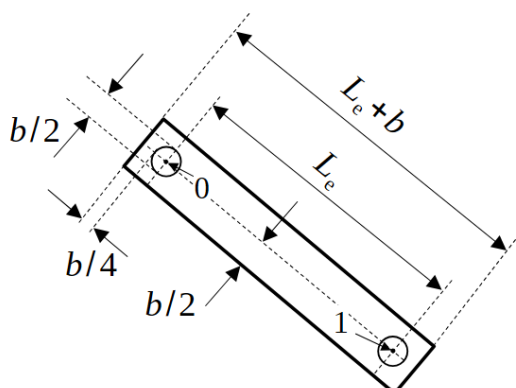
O primeiro procedimento necessário para o desenho da treliça é padronizar a largura das barras e o diâmetro dos pinos conectores dos elementos (como se pretende apenas um modelo 2D, a profundidade padrão unitária será aceita). Uma proposta razoável é adotar a largura das barras, b (ver Figura 6), como sendo proporcional ao comprimento da maior barra da treliça, L_{max} , ou seja

$$b = kL_{max} \quad \text{Eq. [2]}$$

em que:

$0 < k < 1$ (neste trabalho, foi escolhido $k = 1/10$).

Figura 6 – Detalhes de um elemento da treliça



Fonte: elaborada pelos autores.

Os pontos “0” e “1”, na Figura 6, são as conectividades do elemento. Assim, dados os *arrays* de nós e de elementos, “nod” e “elm”, respectivamente, o código inicial em PythonSCAD para a obtenção de $L_{\max}$ pode ser descrito como:

```
Lmax = 0
Ne = len(elm)
for i in range(Ne):
    Dx = nod[elm[i][0]][0]-nod[elm[i][1]][0]
    Dy = nod[elm[i][0]][1]-nod[elm[i][1]][1]
    L = (Dx**2+ Dy**2)**0.5
    if L>Lmax: Lmax = L
```

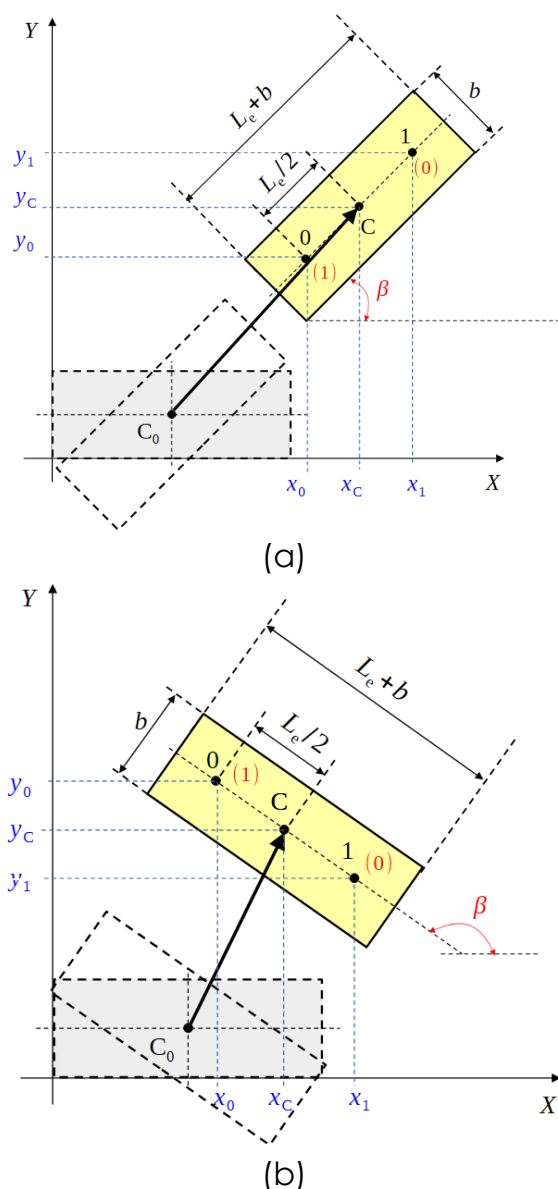
Retângulos, representando os elementos, são criados a partir do canto inferior esquerdo, que é utilizado como eixo de rotação para a função “**rotate**”, como ilustrado na Figura 4, a não ser que se especifique que o centro de rotação é o baricentro do elemento, por meio do argumento “center=True”. Sendo este o caso, é possível criar a barra retangular de altura b , e comprimento $L_e + b$, rotacioná-la, e, depois, transladá-la, como ilustrado nas Figuras 7 (a) e 7 (b). Na ordem: cria-se elemento retangular (retângulo cinza tracejado), efetua-se uma rotação, β , em torno do baricentro, C_0 , seguida de uma translação, ao longo do vetor $\overrightarrow{C_0C}$.

Considerando-se as Figuras 7 (a) e 7 (b), as seguintes relações são observadas

$$\begin{cases} x_C = x_0 + \text{sign}(x_1 - x_0) \left(\frac{L_e}{2}\right) |\cos \beta| \\ y_C = y_0 + \text{sign}(y_1 - y_0) \left(\frac{L_e}{2}\right) |\sin \beta| \end{cases} \quad \text{Eq. [3]}$$

A função *sign* na Eq. [3] retorna os valores “+1” ou “-1”, de acordo com o sinal resultante da expressão entre parêntesis. O objetivo é corrigir os sinais de $\cos \beta$ e $\sin \beta$ para o caso em que os nós “0” e “1” estejam dispostos de maneira invertida, como indicado pelas numerações em vermelho nas Figuras 7 (a) e (b).

Figura 7 – Rotação e translação do elemento retangular para (a) $0^\circ < \beta < 90^\circ$ e (b) $90^\circ < \beta < 180^\circ$



Fonte: elaborada pelos autores.

O seguinte *script* define a função “*sign*” (Eq. [3]) e cria as barras da treliça de acordo com o algoritmo proposto.



```

import math
def sign(number):
    sg = 0
    if number > 0: sg = 1
    elif number < 0: sg = -1
    return sg
for i in range(Ne):
    x0 = nod[elm[i][0]][0]
    x1 = nod[elm[i][1]][0]
    y0 = nod[elm[i][0]][1]
    y1 = nod[elm[i][1]][1]
    L = ((x1-x0)**2 + (y1-y0)**2)**0.5
    sn = (y1-y0)/L
    cs = (x1-x0)/L
    xc = x0 + sign(x1-x0)*(L/2)*abs(cs)
    yc = y0 + sign(y1-y0)*(L/2)*abs(sn)
    beta = math.atan((y1-y0)/(x1-x0))
    beta = math.degrees(beta)
    bar = square([L + b, b], center = True)
    bar = bar.rotate([0, 0, beta])
    bar = bar.translate([xc, yc, 0]).show()

```

O programa completo, disponibilizado no Apêndice, prevê ainda a possibilidade de se mostrar os nós e os elementos juntamente com seus respectivos números no desenho da treliça. Isso é importante, pois será necessário especificar o carregamento da treliça e os tipos de apoios, antes da determinação numérica das forças nas barras. Como exemplo, a Figura 8 ilustra o desenho gerado da treliça da Figura 2, com $h = 100\text{mm}$, $p = 200\text{mm}$ e três barras horizontais inferiores. O script gerador do desenho é como segue:

```

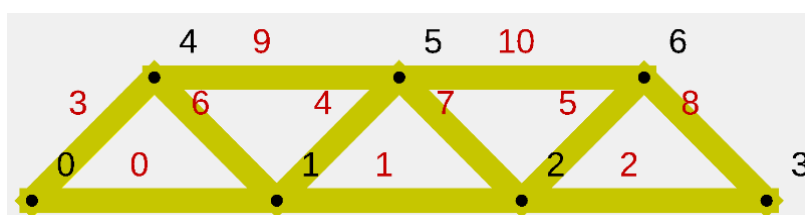
no = genos_wr(200,100,3)
el = gelm_wr(3)
des_wr(no, el, "Yellow", "Black")

```

Com relação ao código anterior, a sub-rotina "des_wr (nod, elm, cor_elm, cor_nod, shownod=True, showelm=True)", desenha a treliça Warren criada por "genos_wr (p,h,N)" e "gelm_wr (N)", que devem ser chamados em sequência

(como observado anteriormente, os *arrays* “no” e “el” também poderiam ter sido criados manualmente). Os parâmetros “cor_elm” e “cor_nod” são as cores desejáveis para os elementos (barras) e os nós. Finalmente, “shownod” e “showelm” definem se os números dos nós e dos elementos gerados serão ou não mostrados no desenho. Por *default*, “shownod=True” e “showelm=True”, de maneira que as numerações são mostradas, como na Figura 8.

Figura 8 – Treliza gerada no PythonSCAD



Fonte: elaborada pelos autores.

CARREGAMENTO DA TRELIÇA

Em cada nó da treliça, podem-se especificar as forças externas atuantes na estrutura. De forma semelhante, os graus de liberdade (indicando as restrições de movimento) devem ser especificados para os nós situados nos apoios da treliça. Essas condições de carregamento devem ser informadas pelo usuário por meio da criação de dois *arrays*, como exemplificado a seguir:

```
no_apoio = [[0,2], [3,1]]
no_carga = [[4, 50,-100], [6, 0, -200]]
```

No *array* “no_apoio”, o primeiro elemento “[0,2]” indica que o nó “0” é um apoio do tipo 2, ou “rótula”. O segundo elemento segue o mesmo padrão e indica que o nó 3 contém um apoio do tipo “1”, ou “deslizante”. As Figuras 2 e 8 ilustram as posições dos nós usados neste exemplo.

No *array* “no_carga”, especifica-se que as magnitudes das componentes, F_x e F_y , das forças aplicadas no nó 4 são $F_{x4} = 50$ e $F_{y4} = -100$, respectivamente. De forma semelhante, o nó 6 é carregado com as forças $F_{x6} = 0$ e $F_{y6} = -200$. As unidades das forças não são necessárias para os cálculos (desde que todas as forças sejam expressas na mesma unidade).



Observa-se que os nomes dos *arrays* não precisam ser “no_apoio” e “no_carga”. Apenas a disposição dos elementos nos *arrays* deve seguir o padrão indicado. Outra observação importante é que não seria possível criar os *arrays* dos tipos de apoio e dos esforços externos sem que, primeiramente, fosse desenhada a treliça (como saber qual a numeração dos nós arbitrada pelo programa?).

A criação dos *arrays* “no_apoio” e “no_carga” constitui um passo necessário para o cálculo dos esforços nas barras da treliça, que será apresentado na sequência.

CÁLCULO DOS ESFORÇOS INTERNOS

Em uma treliça, cada barra está sujeita a uma força puramente axial de tração ou de compressão. Assim, para uma treliça com 11 elementos (enumerados de 0 a 10), como a representada na Figura 8, existem 11 forças axiais desconhecidas, juntamente com as três forças de reação nos nós de apoio: duas reações (em X e em Y) no apoio do tipo 2 e uma reação (em Y) no apoio do tipo 1 (Figura 1). Assim, sendo N_e o número de elementos da treliça, existirão $N_e + 3$ variáveis a serem determinadas. Portanto, para um sistema estaticamente determinado, será necessário um igual número de equações.

Para cada nó da treliça, duas equações garantem o equilíbrio estático no plano XY . Cada uma das equações indica que o somatório das componentes das forças (em X ou em Y) deve ser igual a zero para o equilíbrio ser satisfeito. Assim, para uma treliça com N_d nós, existirão $2N_d$ equações disponíveis. No caso da treliça da Figura 8, por exemplo, existirão 14 equações, uma para cada um dos 7 nós da treliça (enumerados de 0 a 6). Em geral, em uma treliça estaticamente determinada, é válida a seguinte relação (Beer, Johnston e Elwood, 1980)

$$2N_d = N_e + 3$$

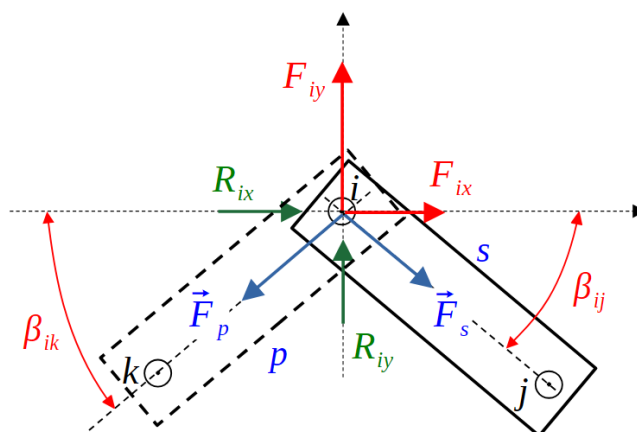
Eq. [4]

Considere-se, então, o equilíbrio estático do i -ésimo nó de uma treliça qualquer, como ilustrado na Figura 9. Nessa figura, todas as possibilidades de esforços estão representadas, muito embora nem todos os nós sejam submetidos a esforços externos ou de reação.

As equações de equilíbrio estático para o nó i da Figura 9 são:

$$\begin{cases} F_p \cos \beta_{ik} + F_s \cos \beta_{ij} + R_{ix} = -F_{ix} \\ F_p \sin \beta_{ik} + F_s \sin \beta_{ij} + R_{iy} = -F_{iy} \end{cases} \quad \text{Eq. [5]}$$

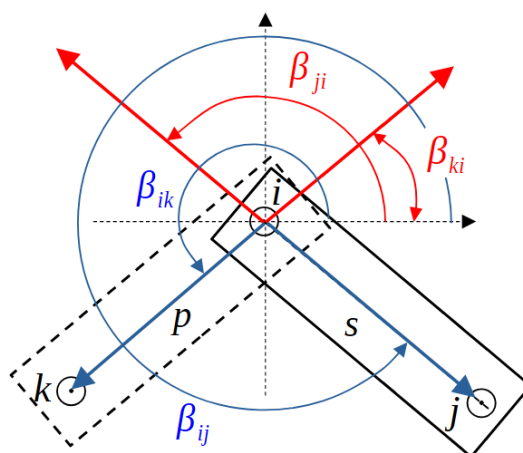
Figura 9 – Forças concorrentes no i -ésimo nó de uma treliça



Fonte: elaborada pelos autores.

A notação para os ângulos dos elementos na Figura 9 não é prática do ponto de vista da criação de um algoritmo computacional, como será visto adiante. A Figura 10 ilustra uma notação mais conveniente, que será adotada neste trabalho, em que, por exemplo, β_{ik} passa a ser o ângulo formado pelo vetor orientado do nó "i" para o nó "k" com relação ao eixo horizontal X . Observa-se que o primeiro nó, "i", corresponde à origem do vetor (os ângulos β_{ik} e β_{ki} , na Figura 10, ilustram esse princípio).

Figura 10 – Ângulos orientados dos elementos p e s



Fonte: elaborada pelos autores.



Ao se considerarem todos os elementos da treliça, as equações de equilíbrio dos nós resultarão em um sistema de equações lineares, $\mathbf{Ax} = \mathbf{b}$, no qual $\mathbf{A}$ é uma matriz quadrada de dimensão $2N_d$, enquanto $\mathbf{x}$ e $\mathbf{b}$ são vetores coluna, também de dimensão $2N_d$. Como exemplo, as condições de equilíbrio dos nós da treliça da Figura 1 resultarão na configuração ilustrada na Tabela 1, na qual são apresentadas as linhas e colunas da matriz $\mathbf{A}$ (em destaque), bem como os vetores $\mathbf{x}$ e $\mathbf{b}$ (últimas colunas à direita). A construção da matriz aumentada, $\mathbf{A}^* = [\mathbf{A}|\mathbf{b}]$, em que a última coluna contém o vetor $\mathbf{b}$, envolve duas etapas bem delineadas que podem ser facilmente automatizadas por um algoritmo computacional, como será descrito a seguir.

Tabela 1 – Construção da matriz $\mathbf{A}$ e dos vetores $\mathbf{x}$ e $\mathbf{b}$, para o sistema de equações $\mathbf{Ax} = \mathbf{b}$

	ELM	0	1	2	3	4	Ne	Ne+1	Ne+2	x	b
i	k/j	0	1	2	3	4	5	6	7		
0	0	$\cos\beta_{03}$	$\cos\beta_{01}$	0	$\cos\beta_{02}$	0	0	0	0	F_0	$-F_{0x}$
	1	$\sin\beta_{03}$	$\sin\beta_{01}$	0	$\sin\beta_{02}$	0	0	0	0	F_1	$+F_{0y}$
1	2	0	$\cos\beta_{10}$	$\cos\beta_{13}$	0	0	1	0	0	F_2	0
	3	0	$\sin\beta_{10}$	$\sin\beta_{13}$	0	0	0	1	0	F_3	0
2	4	0	0	0	$\cos\beta_{20}$	$\cos\beta_{23}$	0	0	0	F_4	0
	5	0	0	0	$\sin\beta_{20}$	$\sin\beta_{23}$	0	0	1	R_{1x}	0
3	6	$\cos\beta_{30}$	0	$\cos\beta_{31}$	0	$\cos\beta_{32}$	0	0	0	R_{1y}	0
	7	$\sin\beta_{30}$	0	$\sin\beta_{31}$	0	$\sin\beta_{32}$	0	0	0	R_{2y}	0

Fonte: elaborada pelos autores.

Dois *loops* aninhados podem ser utilizados para gerar a primeira parte da matriz $\mathbf{A}^*$, entre as colunas 0 e 4 (parte da matriz referente aos ângulos orientados dos elementos). O *loop* externo realiza uma varredura no *array* dos elementos (colunas j). Para cada coluna, realiza-se um *loop* (interno) nos nós, i , verificando-se se o número do nó está contido no elemento, j . Observe-se que para o elemento j , de conectividades c_1 e c_2 , os endereços (k, j) das células correspondentes aos nós $i = c_1$ e $i = c_2$ serão obtidos fazendo-se $k = 2i$ e $k = 2i + 1$, respectivamente. A primeira linha ($k = 2i$) contém o cosseno do ângulo β , obtido a partir do vetor orientado do nó i até o nó correspondente à conectividade seguinte do elemento. Por exemplo, se $i = c_1$, β corresponderá ao vetor orientado de c_1 para c_2 . A segunda linha $k = 2i + 1$ segue o mesmo modelo para o seno do ângulo β . O código em PythonSCAD para a implementação dessa primeira parte do algoritmo é apresentado a seguir.

$N_e = \text{len}(\text{elm})$

$N_d = \text{len}(\text{nod})$



```

A = [[0 for _ in range(2*Nd+1)] for _ in range(2*Nd)]
for j in range(Ne):
    for i in range(Nd):
        if i in elm[j]:
            c_1 = elm[j].index(i)
            c_2 = 1 if c_1==0 else 0
            #-----
            x0 = nod[elm[j][c_1]][0]
            x1 = nod[elm[j][c_2]][0]
            y0 = nod[elm[j][c_1]][1]
            y1 = nod[elm[j][c_2]][1]
            L = ((x1-x0)**2 + (y1-y0)**2)**0.5
            sn = (y1-y0)/L
            cs = (x1-x0)/L
            #-----
            A[2*i][j] = cs
            A[2*i+1][j] = sn

```

A partir da coluna $j = N_e$, procura-se, primeiramente, o nó i que contém o apoio do tipo 2 e procede-se da seguinte forma:

$$\begin{cases} A^*(2i, N_e) = 1 \\ A^*(2i + 1, N_e + 1) = 1 \end{cases} \quad \text{Eq. [6]}$$

Finalmente, procura-se o nó i , que contém o apoio do tipo 1, e escreve-se:

$$A^*(2i + 1, N_e + 2) = 1 \quad \text{Eq. [7]}$$

O seguinte código pode ser utilizado para complementar as três últimas colunas da matriz A^* , assumindo-se que o array "apoio" contém informações sobre os tipos de restrição nos nós (semelhantemente ao array "no_apoio", descrito na seção anterior).

```

for i in range(len(apoio)):
    no = apoio[i][0]
    if apoio[i][1]==2:
        A[2*no][Ne] = 1

```



```

A[2*no+1][Ne+1] = 1
if apoio[i][1]==1:
    A[2*no+1][Ne+2] = 1

```

Finalmente, a última coluna da matriz $\mathbf{A}^*$ deve ser preenchida com os simétricos dos valores dos esforços, prescritos para os nós selecionados (coluna $\mathbf{b}$ na Tabela 1). Para um *array* de entrada “carga”, o seguinte código executará esta tarefa:

```

for i in range(len(carga)):
    no = carga[i][0]
    A[2*no][Ne+3] = - carga[i][1]
    A[2*no+1][Ne+3] = - carga[i][2]

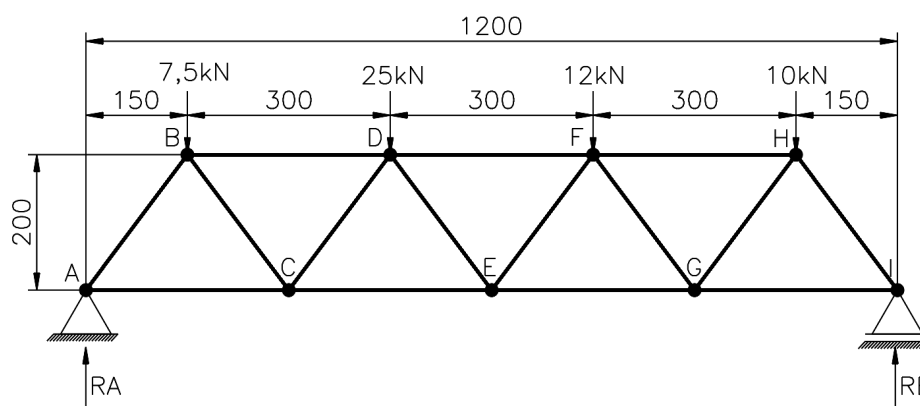
```

Após a criação da matriz $\mathbf{A}^*$, aplica-se um método numérico para a resolução do sistema linear. O método de eliminação de Gauss, cujo algoritmo em Python pode ser encontrado em Geek for Geeks (2025), foi utilizado para esse fim. Com o programa completo, armazenado no arquivo “trelica.py”, procede-se à utilização prática, descrita na próxima seção.

ESTUDOS DE CASO

Nesta seção, serão desenvolvidos três estudos de caso. No primeiro, será analisada uma treliça com carregamento puramente vertical (Figura 11), obtida *on-line* (Jacob, 2026). No segundo, será realizada a simulação da treliça publicada por Farias, Jesus e Oliveira (2020). Em ambos os casos, as treliças contêm quatro vãos em sua base, diferindo apenas quanto às suas dimensões. Finalmente, será ilustrada a resolução de uma treliça Pratt, na qual os *arrays* dos nós e dos elementos são criados diretamente pelo usuário. É importante ressaltar que nenhuma comparação entre dados experimentais e numéricos será realizada neste trabalho, uma vez que esse não é o objetivo. O leitor interessado em comparar resultados teóricos e experimentais pode consultar, por exemplo, Farias, Jesus e Oliveira (2020), trabalho no qual uma treliça bidimensional é construída e os esforços nas barras são medidos experimentalmente.

Considere-se, inicialmente, a treliça de quatro vãos ilustrada na Figura 11, apresentada a seguir.

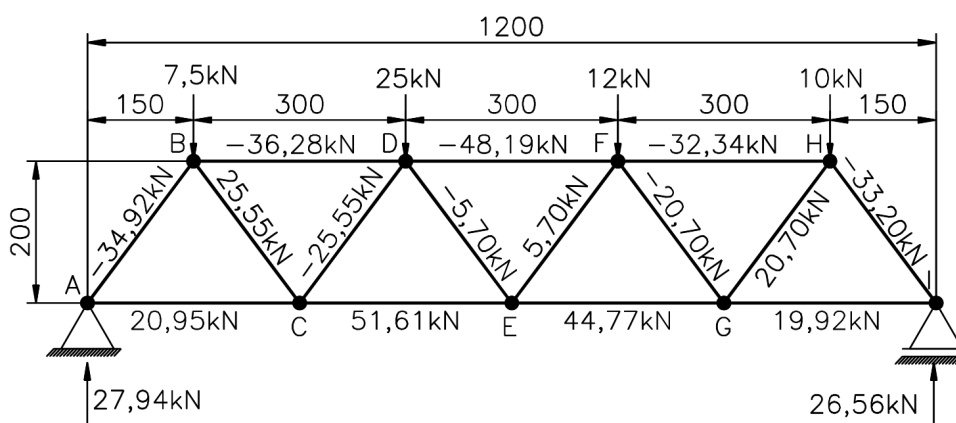
Figura 11 – Treliza Warren de quatro vãos, com carregamento vertical nos nós superiores

Dimensões em cm

www.calculistadeaco.com.br

Fonte: <https://calculistadeaco.com.br/como-calculiar-uma-trelica-metalica-usando-o-metodo-dos-nos>. Acesso em 05 jan, 2026.

A solução da treliza da Figura 11 está disponibilizada *on-line* (Jacob, 2026) e foi validada por meio da comparação com resultados numéricos obtidos em um aplicativo para cálculos estruturais. A Figura 12 mostra os resultados para cada barra, sendo que os esforços positivos indicam tração e os negativos, compressão nas barras.

Figura 12 – Resultados da análise de forças da treliza Warren

Dimensões em cm

www.calculistadeaco.com.br

Fonte: <https://calculistadeaco.com.br/como-calculiar-uma-trelica-metalica-usando-o-metodo-dos-nos>. Acesso em 05 jan, 2026.

Observa-se que a convenção de sinais nas barras da Figura 12 (positivo para tração e negativo para compressão) é coerente com a análise de forças da Figura



9, na qual são representadas forças de tração atuando sobre o nó. Assim, se, ao final da resolução numérica, o sinal da força for invertido, isso indica que a força em questão é compressiva, divergindo da representação inicial. Raciocínio semelhante deve ser aplicado às reações nos apoios, que são inicialmente representadas nos sentidos positivos dos eixos cartesianos.

Para simular a treliça da Figura 12, assumindo-se que o arquivo “treliça.py” esteja presente no mesmo diretório, cria-se um novo programa no ambiente PythonSCAD e adicionam-se as seguintes linhas:

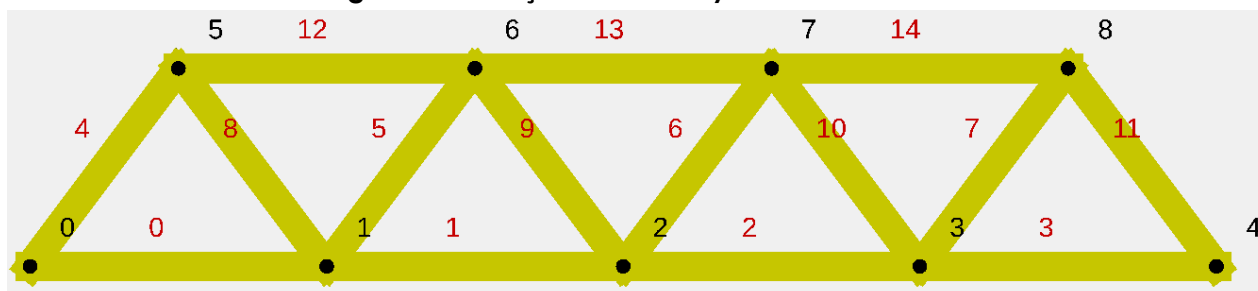
```
from treliça import *  
nos = genos_wr(300,200,4)  
el = gelm_wr(4)  
des_wr(nos,el)
```

O código anterior cria uma treliça semelhante àquela da Figura 11. A Figura 13 ilustra o resultado da execução do programa.

Uma vez desenhada a treliça, aplicam-se as cargas nos nós 5 a 8, conforme inicialmente indicado nas Figuras 11 e 12. Observa-se que a numeração gerada pelo programa é diferente da numeração do problema original. Os tipos de apoio nos nós 0 e 4 também devem ser descritos. Com essas informações, a treliça pode ser calculada pela função “solwr”. O código completo é listado a seguir:

```
from treliça import *  
nos = genos_wr(300,200,4)  
el = gelm_wr(4)  
des_wr(nos,el)  
ap = [[0,2], [4,1]]  
car = [[5, 0,-7.5], [6, 0, -25], [7, 0, -12], [8, 0, -10]]  
solwr(nos, el, ap, car)
```

Figura 13 – Treliza criada no PythonSCAD



Fonte: elaborada pelos autores.

O resultado do programa é exibido no “console” do PythonSCAD como:

Força na barra 0 : 20.953
 Força na barra 1 : 51.609
 Força na barra 2 : 44.766
 Força na barra 3 : 19.922
 Força na barra 4 : -34.922
 Força na barra 5 : -25.547
 Força na barra 6 : 5.703
 Força na barra 7 : 20.703
 Força na barra 8 : 25.547
 Força na barra 9 : -5.703
 Força na barra 10 : -20.703
 Força na barra 11 : -33.203
 Força na barra 12 : -36.281
 Força na barra 13 : -48.187
 Força na barra 14 : -32.344
 Reações na rótula A. $R_{Ax} = -0.0$ e $R_{Ay} = 27.937$
 Reação no apoio deslizando B. $R_{By} = 26.562$

A Tabela 2 compara os resultados obtidos com aqueles exibidos na Figura 12. Pode-se observar que os valores são praticamente idênticos.

Tabela 2 – Comparação entre os valores simulados e os obtidos por Jacob (2026)

BARRA	FORÇA	FORÇA (Jacob, 2026)
0	20,953 kN	20,95 kN
1	51,609kN	51,61kN

2	44,766kN	44,77kN
3	19,922kN	19,92kN
4	-34,922kN	-34,92kN
5	-25,547kN	-25,55kN
6	5,703kN	5,70kN
7	20,703kN	20,70kN
8	25,547kN	25,55kN
9	-5,703kN	-5,70kN
10	-20,703kN	-20,70kN
11	-33,203kN	-33,20kN
12	-36,281kN	-36,28kN
13	-48,187kN	-48,19kN
14	-34,344kN	-32,34kN
Apoios		
Esquerdo-X	0kN	-----
Esquerdo-Y	27,937kN	27,94kN
Direito	26,562kN	26,56kN

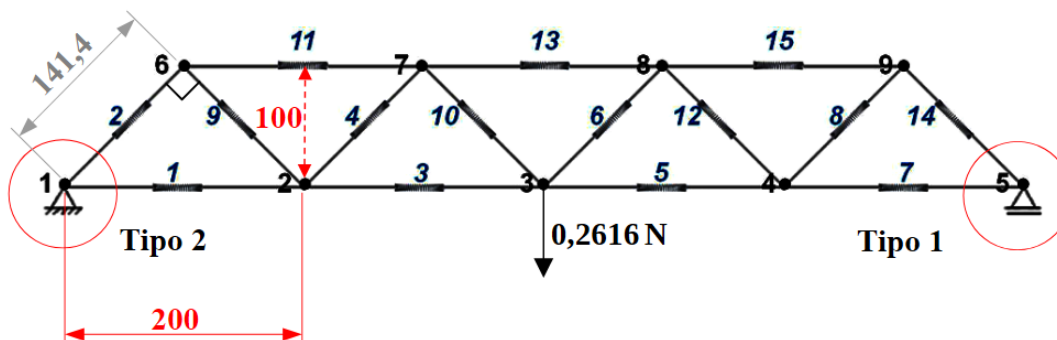
Fonte: elaborada pelos autores.

No segundo estudo de caso, a treliça a ser analisada é a apresentada por Farias, Jesus e Oliveira (2020), reproduzida na Figura 14, com a numeração das barras conforme o artigo original.

O código para o desenho da treliça da Figura 14 é o seguinte:

```
from trelica import *
nos = genos_wr(200,100,4)
el = gelm_wr(4)
des_wr(nos,el)
```

Figura 14 – Treliça Warren de quatro vãos, com carregamento vertical nó central inferior (dimensões em mm)]



Fonte: Farias, Jesus e Oliveira, 2020.



A treliça gerada é idêntica à apresentada na Figura 13 e, por isso, não há necessidade de repeti-la. De fato, como não houve mudança no número de barras da treliça, a numeração permaneceu a mesma, de modo que é possível utilizar a Figura 13 como base e modificar o vetor de carga, que passa a ser composto apenas pelo nó central inferior (nó 2), ou seja:

$$\text{car} = [[2, 0, -0.2616]]$$

A Tabela 3 compara os resultados do programa desenvolvido neste artigo com os resultados teóricos publicados por Farias, Jesus e Oliveira (2020). Novamente, os valores obtidos são praticamente idênticos.

Tabela 3 – Comparação entre os valores simulados e os obtidos por Farias, Jesus e Oliveira (2020)

BARRA	FORÇA	FORÇA (Farias, Jesus e Oliveira, 2020)
0	0,131 N	0,1308 N
1	0,392N	0,3923N
2	0,392N	0,3923N
3	0,131N	0,1308N
4	-0,185N	-0,1849N
5	-0,185N	-0,1849N
6	0,185N	0,1849N
7	0,185N	0,1849N
8	0,185N	0,1849N
9	0,185N	0,1849N
10	-0,185N	-0,1849N
11	-0,185N	-0,1849N
12	-0,262N	-0,2616N
13	-0,523N	-0,5229N
14	-0,262N	-0,2616N
Apoios		
Esquerdo-X	0kN	-----
Esquerdo-Y	0,131N	0,1308N
Direito	0,131N	0,1308N

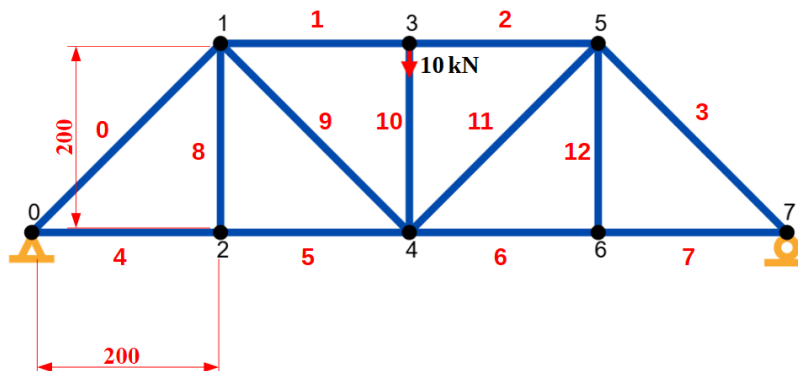
Fonte: elaborada pelos autores.

O último exemplo foi gerado pelo software on-line *Efficalc – Truss design*⁷. A Figura 15 mostra a treliça construída automaticamente pelo aplicativo, capaz de gerar e calcular treliças dos tipos Pratt, Warren e Howe. No caso da Figura 15, a treliça escolhida foi do tipo Pratt. As dimensões da treliça foram alteradas para 200mm de altura e 200mm de vão inferior (a treliça original possui 2m de altura por 2m de vão), a fim de melhorar a visualização. Como no cálculo das forças na

⁷ <https://www.trussanalysis.com>. Acesso em: 05 de jan. de 2026

treliça apenas as proporções entre altura e vão são significativas, essa mudança não altera os resultados.

Figura 15 – Treliza Pratt carregada no nó central superior



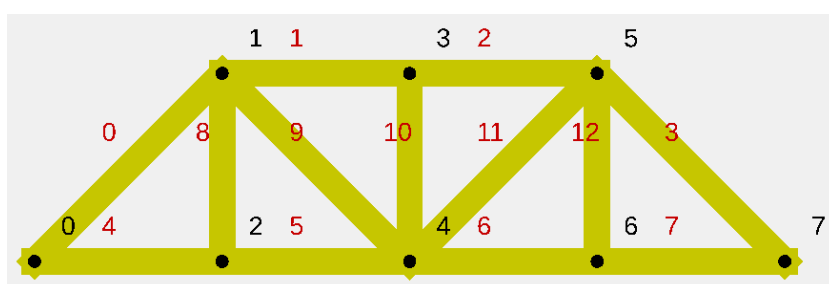
Fonte: <https://www.trussanalysis.com>. Acesso em: 05 de jan. de 2026.

Para resolver a treliça da Figura 15, devem-se preencher manualmente os vetores “nos” e “el”. Assim, observando-se essa figura, obtém-se o seguinte programa:

```
from trelica import *
nos = [[0,0], [200,200], [200,0], [400,200], [400,0], [600, 200], [600,0], [800,0]]
el = [[0,1], [1,3], [3,5], [5,7], [0,2], [2,4], [4,6], [6,7], [2,1], [4,1],[4,3], [4,5], [5,6]]
des_wr(nos,el)
ap = [[0,2], [7,1]]
car = [[3, 0,-10]]
solwr(nos, el, ap, car)
```

A Figura 16 ilustra a treliça gerada pelo programa em PythonSCAD.

Figura 16 – Treliza Pratt gerada pela biblioteca proposta em PythonSCAD



Fonte: elaborada pelos autores.



A Tabela 4 compara os resultados obtidos pelo programa em PythonSCAD com os resultados fornecidos pela simulação *on-line*. Novamente, observa-se uma concordância perfeita entre os valores calculados.

Tabela 4 – Comparação entre os valores simulados e os calculados pelo *Efficalc*

BARRA	FORÇA	FORÇA (<i>Efficalc</i>)
0	-7,071 kN	-7,071 kN
1	-10,0 kN	-10,0 kN
2	-10,0 kN	-10,0 kN
3	-7,071 kN	-7,071 kN
4	5,0 kN	5,0 kN
5	5,0 kN	5,0 kN
6	5,0 kN	5,0 kN
7	5,0 kN	5,0 kN
8	0,0 kN	0,0 kN
9	7,071 kN	7,071 kN
10	-10,0 kN	-10,0 kN
11	7,071 kN	7,071 kN
12	0,0 kN	0,0 kN
Apoios		
Esquerdo-X	0kN	-----
Esquerdo-Y	5,0 kN	-----
Direito	5,0 kN	-----

Fonte: elaborada pelos autores.

Observação: o simulador *Efficalc* não forneceu os valores das reações nos apoios 0 e 7. Os resultados foram comparados com aqueles indicados na Tabela 5, extraída da página do *Efficalc* após a simulação.

Tabela 5 – Resultados da simulação do *Efficalc*

MEMBER ID	START -> END NODE	AXIAL FORCE (kN)
0	0 → 1	-7,071
1	1 → 3	-10
2	3 → 5	-10
3	5 → 7	-7,071
4	0 → 2	5
5	2 → 4	5
6	4 → 6	5
7	6 → 7	5
8	1 → 2	0
9	1 → 4	7,071
10	3 → 4	-10
11	4 → 5	7,071
12	→ 6	0

Fonte: <https://www.trussanalysis.com>. Acesso em: 05 de jan. de 2026.



CONCLUSÕES

Neste trabalho, apresentou-se um algoritmo para o desenho e o cálculo de treliças planas do tipo Warren. Apesar de não ter sido possível realizar a programação em OpenSCAD, o PythonSCAD se mostrou uma ferramenta adequada, tornando possível a obtenção dos resultados.

O código disponibilizado ao final do artigo é útil para o estudo preliminar de treliças por estudantes dos níveis médio e superior, especialmente nas áreas de Engenharia e Arquitetura, o que constituiu o principal objetivo da pesquisa. Além disso, o programa é aberto e pode ser amplamente aperfeiçoado. Por exemplo, podem-se criar estruturas mais elaboradas para impressão 3D e, assim, produzir um modelo experimental a partir da análise teórica.

Os autores desconhecem outro trabalho publicado até o presente momento que una as facilidades de uma linguagem paramétrica de desenho (OpenSCAD) à versatilidade de uma linguagem moderna como o Python. Os desenhos podem ser aprimorados de modo a incluir os apoios, bem como a representação das forças nas barras, quer por meio de uma escala de cores, quer por descrição textual.

Uma grande dificuldade encontrada foi a limitação da linguagem OpenSCAD que, a princípio, havia sido escolhida para o desenvolvimento da biblioteca. Uma leitura preliminar da documentação disponível *on-line* dava a impressão de que a linguagem era “completa”. Foi somente a partir da codificação de estruturas em *loop* que as limitações do OpenSCAD se tornaram evidentes. A descoberta do PythonSCAD foi essencial para a mudança de rumo da pesquisa, possibilitando o desenvolvimento do trabalho apresentado neste artigo.

Por fim, ressalta-se que o programa pode ser adaptado para diferentes tipos de treliça mediante a modificação da função “*gelm_wr()*”. Dessa forma, as perspectivas de desenvolvimento futuro deste trabalho abrangem tanto melhorias no cálculo, incluindo, por exemplo, aspectos não abordados, como as deformações das barras, quanto o aprimoramento das rotinas de desenho.

Financiamento

Esta pesquisa foi financiada pela Fundação de Amparo à Ciência e Tecnologia do Estado de Pernambuco (FACEPE), por meio do processo de número BICJ-0031-3.05/24.



REFERÊNCIAS

- BARROZO, *et al.* Análise dos deslocamentos e reações de apoio em treliças planas pelo método dos elementos finitos em comparativo ao software FTOOL. **Revista Científica Semana Acadêmica**, v. 1, n. 182, p. 1–20, 2019.
- BEER, F. P.; JOHNSTON JR., E. R. **Mecânica vetorial para engenheiros: estática**. 3. ed. Brasil: McGraw-Hill, 1980.
- BRAGA, C. A. OpenSCAD, gerando modelos tridimensionais para uso no ensino de geometria. **Brazilian Journal of Development**, v. 9, n. 5, p. 17673–17686, 2023.
- DELGADO JR., *et al.* Um modelo para aplicação em análise estrutural de treliças planas: formulação e implantação pelo método dos elementos finitos. **Cadernos UniFOA**, v. 17, n. 50, p. 1–18, 2022.
- ERNST, L. **11 types of trusses [the most used]**. 2023. Disponível em: <https://www.structuralbasics.com/types-of-trusses/>. Acesso em: 30 dez. 2025.
- FARIAS, R. B.; JESUS, V. L. B.; OLIVEIRA, A. L. Uma maquete da estrutura em treliças simples triangulares para o ensino de estática. **Revista Brasileira de Ensino de Física**, v. 42, 2020.
- GEEK FOR GEEKS. **Gaussian elimination to solve linear equations**. Disponível em: <https://www.geeksforgeeks.org/dsa/gaussian-elimination>. Acesso em: 17 dez. 2025.
- HORVATH, J.; CAMERON, R. **3D printed science projects: ideas for your classroom, science fair or home**. 1. ed. Estados Unidos: MatterHackers/Apress, 2016.
- HORVATH, J.; CAMERON, R. **3D printed science projects volume 2: physics, math, engineering and geology models**. 1. ed. Estados Unidos: MatterHackers/Apress, 2017.
- HORVATH, J.; CAMERON, R. **Make geometry: learn by coding, 3D printing and building**. 1. ed. Estados Unidos: Make Community LLC, 2021.
- HORVATH, J.; CAMERON, R. **Make calculus: build models to learn, visualize, and explore**. 1. ed. Estados Unidos: Make Community LLC, 2022.
- HORVATH, J.; CAMERON, R. **Make trigonometry: build your way from triangles to analytic geometry**. 1. ed. Estados Unidos: Make Community LLC, 2023.
- JACOB, F. **Como calcular uma treliça metálica usando o método dos nós**. Disponível em: <https://calculistadeaco.com.br/como-calcular-uma-trelica-metalica-usando-o-metodo-dos-nos>. Acesso em: 5 jan. 2026.
- YANG, X.; LI, X.; SUN, P. Structural analysis and optimization of steel truss bridge based on finite element method. **Applied and Computational Engineering**, v. 25, p. 133–141, 2023.

APÊNDICE – CÓDIGO “treliça.py”

```
# Biblioteca para cálculo de treliças
# Autores: Gustavo Koury Costa
# e Vivia Velloso Fontes
```

```
# Deverá ser salvo como texto em um
arquivo.
# (sugere-se “trelica.py”)
```

```

#
# Para utilizar esta biblioteca, adicione a
# seguinte linha no início do programa no
# ambiente PythonSCAD:
#
# from trelica import *
#
# (exemplos de uso se encontram no
artigo)
#
# OBS: é necessário que esta
# biblioteca esteja no mesmo diretório
# (pasta) do seu programa
#
# A utilização é livre, desde que este
# artigo seja citado como fonte, ainda
que
# o programa sofra alterações

from openscad import *
import math

#-----
def genos_wr(p,h,N):
    no_in = [[i*p, 0] for i in range(N+1)]
    no_sup = [[(j + 1/2)*p, h] for j in range(N)]
    return (no_in + no_sup)

#-----

def gelm_wr(N):
    elA = [[i, i+1] for i in range(N)]
    elB = [[i, i+N+1] for i in range(N)]
    elC = [[i, N+i] for i in range(1, N+1)]
    elD = [[i, i+1] for i in range(N+1, 2*N)]
    return elA + elB + elC + elD
#-----

def sign(number):
    sg = 0
    if number > 0: sg = 1
    elif number < 0: sg = -1
    return sg
#-----

```

```

def des_wr (nod,elm, cor_elm = "yellow",
cor_nod = "black", shownod=True, showelm
=True):
    Lmax = 0
    Ne = len(elm)
    Nd = len(nod)
    # Primeiro Loop: determinação do
comprimento máximo
    for i in range(Ne):
        Dx = nod[elm[i][0]][0]-nod[elm[i][1]][0]
        Dy = nod[elm[i][0]][1]-nod[elm[i][1]][1]
        L = (Dx**2+ Dy**2)**0.5
        if L>Lmax: Lmax = L
    # Determinação da largura da barra
    k = 0.1
    b = k*Lmax
    # Segundo Loop: Desenho das barras
    for i in range(Ne):
        x0 = nod[elm[i][0]][0]
        x1 = nod[elm[i][1]][0]
        y0 = nod[elm[i][0]][1]
        y1 = nod[elm[i][1]][1]
        L = ((x1-x0)**2+ (y1-y0)**2)**0.5
        sn = (y1-y0)/L
        cs = (x1-x0)/L
        xc = x0 + sign(x1-x0)*(L/2)*abs(cs)
        yc = y0 + sign(y1-y0)*(L/2)*abs(sn)
        if ((x1-x0)!= 0):
            beta = math.atan((y1-y0)/(x1-x0))
        if ((x1-x0) == 0):
            beta = (math.pi)/2

        beta = math.degrees(beta)
        bar = square([L + b, b], center =
True).color(cor_elm)
        bar = bar.rotate ([0, 0, beta])
        bar = bar.translate([xc,yc, 0]).show()
        if (showelm==True):
            text(str(i), size = 20).translate([xc-
b,yc+b, 0]).color("red"). show()
    # Terceiro Loop: Desenho dos nós
    for i in range (Nd):
        xn = nod[i][0]
        yn = nod[i][1]
        if (shownod==False):

```

```

        no =
circle(b/4).translate([xn,yn,0]).color(cor_nod).
show()
    else:
        no =
circle(b/4).translate([xn,yn,0]).color(cor_nod).
show()
        lb = text(str(i),
size=20).translate([xn+b,yn+b,
0]).color("black").show()
#-----
# A função a seguir foi obtida do site
#
https://www.geeksforgeeks.org/dsa/gaussian-elimination
# em Novembro de 2025

def gaussianElimination(mat, N):
    for k in range(N):
        i_max = max(range(k, N),
key=lambda i: abs(mat[i][k]))
        if mat[i_max][k] == 0:
            print("Singular Matrix.")
            print("Inconsistent System." if
mat[i_max][N]
            else "May have infinitely many
solutions.")
            return
        mat[k], mat[i_max] = mat[i_max],
mat[k]

        for i in range(k + 1, N):
            f = mat[i][k] / mat[k][k]
            mat[i][k + 1:] = [mat[i][j] - f *
mat[k][j]

                                for j in range(k + 1, N + 1)]
            mat[i][k] = 0

        x = [0]*N
        for i in range(N - 1, -1, -1):

```

```

            x[i] = (mat[i][N] - sum(mat[i][j] * x[j]
for j in range(i + 1, N))) / mat[i][i]

        print("\nSolution for the system:")
        return x
#-----

def solwr (nod, elm, apoio,carga):
    Ne = len(elm)
    Nd = len(nod)
    A = [[0 for _ in range(2*Nd+1)] for _ in
range(2*Nd)]
    for j in range(Ne):
        for i in range(Nd):
            if i in elm[j]:
                c_1 = elm[j].index(i)
                c_2 = 1 if c_1==0 else 0
                #-----
                x0 = nod[elm[j][c_1]][0]
                x1 = nod[elm[j][c_2]][0]
                y0 = nod[elm[j][c_1]][1]
                y1 = nod[elm[j][c_2]][1]
                L = ((x1-x0)**2+ (y1-y0)**2)**0.5
                sn = (y1-y0)/L
                cs = (x1-x0)/L
                #-----
                A[2*i][j] = cs
                A[2*i+1][j] = sn
    for i in range(len(apoi)):
        no = apoio [i][0]
        if apoio[i][1]==2:
            A[2*no][Ne] = 1
            A[2*no+1][Ne+1] = 1
        if apoio[i][1]==1:
            A[2*no+1][Ne+2] = 1
    for i in range(len(carga)):
        no = carga [i][0]
        A[2*no][Ne+3] = - carga[i][1]
        A[2*no+1][Ne+3] = - carga[i][2]

```