**Macheps:**

```
function find_eps(Type(Float16,32,63))
   mach_eps = Type(1.0)
   while Type(1.0) + mach_eps / 2 > Type(1.0)
      mach_eps /= 2
   end
   return mach_eps
end
```

**Eta:**

```
function find_eta(Type)
   eta = Type(1.0)  // Inicjalizacja eta wartością 1.0 w danym formacie zmiennopozycyjnym.
   while Type(eta / 2) > Type(0.0)
      // Pętla wykonuje się, dopóki warunek spełniony:
      // Czy połowa obecnej wartości eta nadal większa od 0.0 w danym formacie zmiennopozycyjnym?
      eta /= 2  // Jeśli warunek jest spełniony, dzielimy aktualną wartość eta przez 2.
   end
   return eta  // Po zakończeniu pętli, funkcja zwraca obliczoną wartość ety.
End
```

**Forward x:**

```
function forward_pseudocode(x, y, T)
   sum = T(0.0)
   for i from 1 to length(x) do
      sum = sum + x[i] * y[i]
   end
   return sum
end
```

**Backward x:**

```
function backward_pseudocode(x, y, T)
   sum = T(0.0)
   for i from length(x) to 1 step -1 do
      sum = sum + x[i] * y[i]
   end
   return sum
end
```

**Metoda bisekcji:**

```
function bisection_method_pseudocode(f, a, b, delta, epsilon)
   u = f(a)     // Wartość funkcji f w punkcie a (początek przedziału)
   v = f(b)     // Wartość funkcji f w punkcie b (koniec przedziału)
   e = b - a    // Długość przedziału [a, b]
   it = 0       // Liczba iteracji

   if (sign(u) == sign(v)) // Funkcja nie zmienia znaku w przedziale [a, b]
      return 0, 0, 0, 1
   end
```

```
  while e > epsilon
    it += 1
    e = e / 2        // Połowa długości przedziału
    c = a + e        // Środek przedziału
    w = f(c)         // Wartość funkcji f w środku przedziału

    if (abs(e) < delta || abs(w) < epsilon) // Warunek końca
      return c, w, it, 0
    end

    if (sign(w) != sign(u))
      b = c
      v = w
    else
      a = c
      u = w
    end
  end

  // Zwróć wyniki końcowe
  return c, w, it, 1
end
```

c – środek przedziału
w – wartość funkcji w przedziale,
sign – zwraca znak,
it – liczba wykonanych iteracji,
e – połowa długości przedziału
err – sygnalizacja błędu:

- 0 - sukces
- 1 - funkcja nie zmienia znaku w przedziale [a,b]

`delta` reguluje zakończenie algorytmu na podstawie długości przedziału
`epsilon` reguluje zakończenie algorytmu na podstawie dokładności wyniku.

### Metoda stycznych (Newtona):

```
function newton_method_pseudocode(f, pf, x0, delta, epsilon, maxit)
  v = f(x0)

  // Sprawdzenie, czy punkt początkowy jest już dostatecznie blisko zera
  if abs(v) < epsilon
    return x0, v, 0, 0  // Zwróć aktualne przybliżenie, wartość funkcji, liczba iteracji (0), kod sukcesu (0)
  end

  // Iteracyjne kroki metody Newtona
  for it from 1 to maxit
    // Sprawdzenie, czy pochodna funkcji w punkcie początkowym jest dostatecznie blisko zera
    if abs(pf(x0)) < epsilon
```

```
        return x0, v, it, 2  // Zwróć aktualne przybliżenie, wartość funkcji, liczba iteracji (it), kod
niepowodzenia (2)
    end

    // Krok metody Newtona
    x1 = x0 - v / pf(x0)
    v = f(x1)

    // Sprawdzenie warunku zakończenia
    if abs(x1 - x0) < delta || abs(v) < epsilon
      return x1, v, it, 0  // Zwróć aktualne przybliżenie, wartość funkcji, liczba iteracji (it), kod sukcesu (0)
    end

    // Aktualizacja przybliżenia początkowego
    x0 = x1
  end

  return x0, v, maxit, 1  // Zwróć ostatnie znane przybliżenie, wartość funkcji, maksymalną liczbę iteracji,
kod niepowodzenia (1)
end
```

x0 – punkt(przybliżenie) początkowe

v – wartość funkcji

err – sygnalizacja błędu

- 0 – sukces (metoda zbieżna)
- 1 - nie osiągnięto wymaganej dokładności w maxit iteracji,
- 2 - pochodna bliska zeru

1. Sprawdzenie, czy punkt początkowy `x0` jest już dostatecznie blisko zera (wartość funkcji mniejsza niż `epsilon`). Jeśli tak, zwróć aktualne przybliżenie, wartość funkcji, liczbę iteracji (0), oraz kod sukcesu (0).
2. Iteracyjne kroki metody Newtona:
    - Sprawdzenie, czy pochodna funkcji w punkcie początkowym (`x0`) jest dostatecznie blisko zera (mniejsza niż `epsilon`). Jeśli tak, zwróć aktualne przybliżenie, wartość funkcji, liczbę iteracji (`it`), oraz kod niepowodzenia (2).
    - Krok metody Newtona: obliczenie nowego przybliżenia `x1` i wartości funkcji `v` w tym punkcie.
    - Sprawdzenie warunku zakończenia: czy różnica między nowym a poprzednim przybliżeniem jest mniejsza niż `delta`, lub czy wartość funkcji w nowym punkcie jest już dostatecznie bliska zeru (`epsilon`). Jeśli tak, zwróć aktualne przybliżenie, wartość funkcji, liczbę iteracji (`it`), oraz kod sukcesu (0).
    - Aktualizacja przybliżenia początkowego `x0` na nowe przybliżenie `x1`.
3. Jeśli liczba iteracji osiągnie maksymalną wartość `maxit`, zwróć ostatnie znane wartości przybliżenia, wartości funkcji, maksymalną liczbę iteracji, oraz kod niepowodzenia (1).

**Metoda siecznych:**

```
function secant_method_pseudocode(f, x0, x1, delta, epsilon, maxit)
  fa = f(x0)
  fb = f(x1)
```

```
  // Iteracyjne kroki metody siecznych
  for it from 1 to maxit
    if abs(fa) < abs(fb)
      swap(x0, x1)
      swap(fa, fb)
    end

    s = (x1 - x0) / (fb - fa)
    x0 = x1
    fa = fb
    x1 = x1 - fa * s
    fb = f(x1)

    // Sprawdzenie warunku zakończenia
    if abs(x1 - x0) < delta || abs(fb) < epsilon
      return x1, fb, it, 0  // Zwróć aktualne przybliżenie, wartość funkcji, liczbę iteracji (it), kod sukcesu
(0)
    end
  end

  return x1, fb, maxit, 1  // Zwróć ostatnie znane wartości przybliżenia, wartości funkcji, maksymalną
liczbę iteracji, oraz kod niepowodzenia (1)
end
```

fa,fb – wartości funkcji w punkcie (przybliżeniu) początkowym x0,x1

`s` jest współczynnikiem nachylenia stycznej do funkcji w punkcie `(x0, f(x0)`, który wyznacza się na podstawie dwóch punktów `(x0, f(x0))` i `(x1, f(x1))`

Metoda siecznych polega na przybliżaniu miejsca zerowego funkcji poprzez przecięcie siecznej z osią x.

`swap (x0, x1)-> x0, x1 = x1, x0`

`if x0` jest mniejsza niż wartość funkcji w `x1`. Zamień wartości `x0` i `x1`, oraz `fa` i `fb`.

**Ilorazy różnicowe:**

```
function ilorazyRoznicowe(x, f)
  n = length(f)
  fx = Array(n)  //wektor

  // Przypisanie wartości f[i] do fx[i] dla wszystkich i
  for i from 1 to n
    fx[i] = f[i]
  end

  // Obliczanie ilorazów różnicowych
  for j from 2 to n
    for i from n to j step -1
      fx[i] = (fx[i] - fx[i-1]) / (x[i] - x[i-j+1])
```

```
    end
  end

  return fx
end
```

x – wektor długości n + 1 zawierający węzły x0, . . . , xn
x[1]=x0,..., x[n+1]=xn
f – wektor zawierający wartości interpolowanej funkcji w węzłach f (x0), . . . , f (xn)
fx – wektor długości n + 1 zawierający ilorazy różnicowe
fx[1]=f [x0],
fx[2]=f [x0, x1],..., fx[n]=f [x0, . . . , xn−1], fx[n+1]=f [x0, . . . , xn]
Iteracja zaczyna się od drugiego stopnia ilorazów różnicowych, ponieważ ilorazy różnicowe pierwszego stopnia są po prostu wartościami funkcji `f`.
W zagnieżdżonej pętli, rozpoczynając od końca (najwyższego stopnia), obliczany jest kolejny iloraz różnicowy na podstawie poprzednich ilorazów różnicowych oraz odpowiednich wartości węzłów.

### Postać Newtona (algorytm Hornera):

```
function horner_interpolation(x, fx, t)
  n = length(x)
  nt = fx[n]

  // Uogólniony algorytm Hornera
  for i from n-1 to 1 step -1
    nt = nt * (t - x[i]) + fx[i]
  end

  return nt
end
```

t – punkt, w którym należy obliczyć wartość wielomianu
nt – wartość wielomianu w punkcie t(stopnia n w punkcie xn).
Iteracja odbywa się w kierunku malejącym, zaczynając od przedostatniego współczynnika.

### Współczynniki w postaci naturalnej:

```
function natural_interpolation_coefficients(x, fx)
  n = length(fx)
  a = Array(n)   //wektor
  a[n] = fx[n]

  // Obliczanie współczynników w postaci naturalnej
  for k from n-1 to 1 step -1
    a[k] = fx[k] - a[k+1] * x[k]

    // Aktualizacja współczynników
    for i from k+1 to n-1
      a[i] += -x[k] * a[i+1]
    end
  end
```

```
    return a
end
```

`a` – wektor, w którym będą przechowywane współczynniki wielomianu w postaci naturalnej. Początkowo wartość `a[n]` ustawiana jest na wartość funkcji w ostatnim węźle, ponieważ jest to współczynnik przy najwyższej potędze.

W pętli iteracyjnej obliczane są pozostałe współczynniki wielomianu interpolacyjnego w postaci naturalnej. Iteracja zaczyna się od przedostatniego współczynnika.

Wewnętrzna pętla aktualizuje współczynniki dla kolejnych potęg wielomianu w postaci naturalnej.

### Metoda eliminacji Gaussa bez częściowego wyboru:

```
function gaussian_elimination(A, n, b)
    x = new Float64[n]

    // Eliminacja współczynników
    for k from 1 to n-1
        for i from k+1 to n
            factor = A[i, k] / A[k, k]
            A[i, k] = 0.0
            for j from k+1 to n
                A[i, j] -= factor * A[k, j]
            end
            b[i] -= factor * b[k]
        end
    end

    // Wyliczanie wartości zmiennych wstecz
    for i from n to 1 step -1
        sum = 0.0
        for j from i+1 to n
            sum += A[i, j] * x[j]
        end
        x[i] = (b[i] - sum) / A[i, i]
    end

    return x
end
```

factor – współczynnik dla wyzerowania.

### Metoda eliminacji Gaussa z częściowym wyborem:

```
function gaussian_elimination_partial_pivot(A, n, b, l)
    x = Array(n) //wektor
    p = [i for i from 1 to n]

    // Eliminacja współczynników z częściowym wyborem
    for k from 1 to n-1
        max_val = 0.0
        max_index = 0
```

```
      // Znalezienie elementu głównego w kolumnie
      for i from k to min(n, k + 2*l)
         if abs(A[p[i], k]) > max_val
            max_val = abs(A[p[i], k])
            if max_val == 0.0
               error("Macierz jest osobliwa")
            end
            max_index = i
         end
      end

      // Zamiana miejscami wierszy
      p[max_index], p[k] = p[k], p[max_index]      // swap (p[max_index], p[k])

      // Eliminacja współczynników
      for i from k+1 to n
         factor = A[p[i], k] / A[p[k], k]
         A[p[i], k] = 0.0
         for j from k+1 to min(n, k + 2*l)
            A[p[i], j] -= factor * A[p[k], j]
         end
         b[p[i]] -= factor * b[p[k]]
      end
   end

   // Wyliczanie wartości zmiennych wstecz
   for i from n to 1 step -1
      sum = 0.0
      for j from i+1 to min(n, i + 2*l)
         sum += A[p[i], j] * x[j]
      end
      x[i] = (b[p[i]] - sum) / A[p[i], i]
   end

   return x
end
```

`p` - wektor permutacji identycznościowej.
`max_val` - maksymalną wartość bezwzględną z elementów w aktualnej **kolumnie.**

`max_index` - indeks **wiersza**, w którym znajduje się element główny o maksymalnej wartości bezwzględnej.

**Rozkład LU bez częściowego wyboru:**

```
function lu_decomposition_without_pivot(U, n, l)
   L = sparse_matrix(n, n)

   // Rozkład LU
   for k from 1 to n-1
```

```
      L[k, k] = 1.0

      // Eliminacja współczynników
      for i from k+1 to min(n, k + l + 1)
         factor = U[i, k] / U[k, k]
         L[i, k] = z
         U[i, k] = 0.0
         for j from k+1 to min(n, k + 2 * l)
            U[i, j] -= factor * U[k, j]
         end
      end
   end
   L[n, n] = 1.0

   return (L, U)
end

function solve_lu(L, U, n, l, b)
   x = Array(n) //wektor

   // Rozwiązanie układu równań z wykorzystaniem LU
   for k from 1 to n-1
      for i from k+1 to min(n, k + l + 1)
         b[i] -= L[i, k] * b[k]
      end
   end

   // Wyliczanie wartości zmiennych wstecz
   for i from n to 1 step -1
      sum = 0.0
      for j from i+1 to min(n, i + l)
         sum += U[i, j] * x[j]
      end

      x[i] = (b[i] - sum) / U[i, i]
   end

   return x
end
```

**Rozkład LU z częściowym wyborem:**

```
function lu_decomposition_with_pivot(U, n, l)
   L = sparse_matrix(n, n)
   p = [i for i from 1 to n]

   // Rozkład LU z częściowym wyborem
   for k from 1 to n-1
      max_val = 0.0
      max_index = 0
```

```
        // Wybór elementu głównego z uwzględnieniem częściowego wyboru
        for i from k to min(n, k + 2*l)
            if abs(U[p[i], k]) > max_val
                max_val = abs(U[p[i], k])
                if max_val == 0.0
                    error("Macierz jest osobliwa")
                end
                max_index = i
            end
        end

        // Zamiana miejscami wierszy z wybranym elementem głównym
        p[max_index], p[k] = p[k], p[max_index]

        // Eliminacja współczynników
        for i from k+1 to n
            factor = U[p[i], k] / U[p[k], k]
            L[p[i], k] = factor
            U[p[i], k] = 0.0
            for j from k+1 to min(n, k + 2*l)
                U[p[i], j] -= factor * U[p[k], j]
            end
        end
    end
    L[n, n] = 1.0

    return (L, U, p)
end

function solve_lu_with_pivot(L, U, n, l, b, p)
    x = create_vector(n)

    // Rozwiązanie układu równań z wykorzystaniem LU i permutacji
    for k from 1 to n-1
        for i from k+1 to min(n, k + l + 1)
            b[p[i]] -= L[p[i], k] * b[p[k]]
        end
    end

    // Wyliczanie wartości zmiennych wstecz
    for i from n to 1 step -1
        sum = 0.0
        for j from i+1 to min(n, i + 2 * l)
            sum += U[p[i], j] * x[j]
        end

        x[i] = (b[p[i]] - sum) / U[p[i], i]
    end
```

```
    return x
end
```

`p` - wektor permutacji identycznościowej.
`max_val` - maksymalną wartość bezwzględną z elementów w aktualnej **kolumnie.**

`max_index` - indeks **wiersza**, w którym znajduje się element główny o maksymalnej wartości bezwzględnej.