Funções Reais

Bissection

```
function bissection(a, b, f, N) {
  for i in range(1, N+1)
    m = (b + a) / 2.0
    if f(a)*f(b) <= 0.0
      b = m
    else
      a = m
  endfor
}
```

False Position

```
function false_position(a, b, f, N) {
  for i in range(1, N+1)
    rr = (a*f(b) - b*f(a)) / (f(b) - f(a))
    if f(a)*f(rr) <= 0
      b = rr
    else
      a = rr
  endfor
}
```

Newton

```
function newton(guess, f, df, N) {
  for i in range(1, N+1)
    guess -= f(guess) / df(guess)
  endfor
}
```

Picard-Peano

```
function picard_peano(guess, rec, N) {
  for i in range(1, N+1)
    guess = rec(guess)
  endfor
}
```

Sistemas Não-Lineares

hn

```
function hn(f1, f2, f1x, f1y, f2x, f2y, x, y)
  return (f1*f2y - f2*f1y) / (f1x*f2y - f2x*f1y)
```

kn

```
Function kn(f1, f2, f1x, f1y, f2x, f2y, x, y)
  return (f2*f1x - f1*f2x) / (f1x*f2y - f2x*f1y)
```

Newton

```
function sys_newton(f1, f2, f1x, f1y, f2x, f2y, x, y, N) {
  for i in range(1, N+1)
    x_ant = x  // preservar o valor de x
    x -= hn()
    y -= kn()
  endfor
}
```

Picard-Peano

```
function sys_picard_peano(x, y, g1, g2, N) {
  for i in range(1, N+1)
    x_ant = x
    x = g1(x, y)
    y = g2(x_ant, y)
  endfor
}
```

Eliminação Gaussiana

Triangularização Superior

```
function upper_triang(amatrix) {
  dimV = len(amatrix)
  for diag in range(dimV)
    aux = amatrix[diag][diag]
    for col in range(dimV + 1)
      amatrix[diag][col] /= aux    // dividir pelo pivot
    endfor
    for lin in range(diag + 1, dimV):
      aux2 = amatrix[lin][diag]
      for col in range(diag, dimV + 1):
        amatrix[lin][col] -= amatrix[diag][col] * aux2
      endfor
    endfor
  endfor
}
```

Triangularização Inferior

```
function lower_triang(amatrix) {
  dimV = len(amatrix)
  for diag in range(dimV – 1, -1, -1)
    for lin in range(diag - 1, -1 , -1)
      aux = amatrix[lin][col]
      for col in range(diag, dimV + 1)
        amatrix[lin][col] -= amatrix[diag][col] * aux
      endfor
    endfor
  endfor
}
```

Sistemas Lineares

Gauss-Jacobi

```
function gauss_jacobi(A, b, x, N) {
  for i in range(1, N+1)
    aux = []
    for j in range(len(A[0]))
      x1 = (b[j]-(A[j][0]*x[0]+A[j][1]*x[1]+A[j][2]*x[2]))/
      A[j][j]
      aux<-x1
    endfor
    x = [x[i] + aux[i] for i in range(len(x))]
  endfor
}
```

Gauss-Seidel

```
function gauss_seidel(A, b, x, N) {
  for i in range(1, N+1)
    for j in range(len(A[0]))
        x1 = (b[j]-(A[j][0]*x[0]+A[j][1]*x[1]+A[j][2]*x[2]))/
        A[j][j]
        x[j] += x1
    endfor
  endfor
}
```

Métodos de Integração

Método dos trapézios

```
function trapezoidal_method(x0, x1, f, h) {
  N = (x1 – x0) / h
  x = x0 + h
  res = 0
  for i in range(1, N)
    res += f(x)
    x += h
  endfor
  return h * (f(x0) + f(x1) + 2*res) / 2
}
```

Método de Simpson

```
function simpson_method(x0, x1, f, h) {
  N = (x1-x0)/h
  res = 0
  x = x0 + h
  for i in range(1, N)
    if i par
      res += 2*f(x)
    else
      res += 4*f(x)
    x += h
  endfor
  return h * (f(x0) + f(x1) + res) / 3
}
```

Quociente de Convergência

```
function convergence_quotient(x0, x1, f, h, method) {
  s = method(x0, x1, f, h)
  s1 = method(x0, x1, f, h/2)
  s2 = method(x0, x1, f, h/4)
  quotient = (s1 – s) / (s2 – s1)
  error = abs(s2 – s1) / (round(quotient,0) – 1)
}
```

EDO's 1ª Ordem

```
*Euler*
function Euler(h, x, y, f, xf) {
  while x < xf
    x += h
    y += h * f(x,y)
  endwhile
}
*Runga-Kutta-2*
function runga_kutta_2(h, x, y, f, xf) {
  h2 = (x + xf) / 2
  while x < xf
    yln = f(x, y)
    delta_y = f(x + h/2, y+ h2 * yln / 2) * h
    x += h
    y += delta_y
  endwhile
}
*Runga-Kutta-4*
function runga_kutta_4(h, x, y, f, xf) 7
  h2 = (x + xf) / 2
  while x < xf
    delta1 = h*f(x,y)
    delta2 = h*f(x + h/2, y + delta1/2)
    delta3 = h*f(x + h*h2/2, y + delta2/2)
    delta4 = h*f(x + h, y + delta3)
    x += h
    y += (delta1 + 2*delta2 + 2*delta3 + delta4) / 6
  endwhile
}
*Quociente de Convergência*
function convergence_quotient(x0, x1, f, h, method) {
  s = method(x0, x1, f, h)
  s1 = method(x0, x1, f, h/2)
  s2 = method(x0, x1, f, h/4)
  quotient = (s1 - s) / (s2 - s1)
  error = abs(s2 - s1) / (round(quotient,0) - 1)
}
```

Sistemas de EDO's

Euler

```
function Euler(deltaX, x, y, z, xf, dy, dz) {

  while x < xf
    x += deltaX
    deltaY = dy(x, y, z)
    deltaZ = dz(x, y, z)
    y += deltaY * deltaX
    z += deltaZ * deltaX

  endwhile

}
```

Runga-Kutta-4

```
function runga_kutta_4(deltaX, x, y, z, xf, dy, dz) {

  while x < xf:
    x += deltaX
    dY1 = dy(x, y, z) * deltaX
    dZ1 = dz(x, y, z) * deltaX
    dY2 = deltaX * dy(x + deltaX / 2, y + dY1 / 2, z + dZ1 / 2)
    dZ2 = deltaX * dz(x + deltaX / 2, y + dY1 / 2, z + dZ1 / 2)
    dY3 = deltaX * dy(x + deltaX / 2, y + dY2 / 2, z + dZ2 / 2)
    dZ3 = deltaX * dz(x + deltaX / 2, y + dY2 / 2, z + dZ2 / 2)
    dY4 = deltaX * dy(x + deltaX, y + dY3, z + dZ3)
    dZ4 = deltaX * dz(x + deltaX, y + dY3, z + dZ3)
    y += (dY1 / 6 + dY2 / 3 + dY3 / 3 + dY4 / 6)
    z += (dZ1 / 6 + dZ2 / 3 + dZ3 / 3 + dZ4 / 6)

  endwhile

}
```

O quociente de convergência faz-se da mesma forma, apenas com mais uma variável.

Minimização Unidimensional

```
*Sequential Search*
function sequential_search(x0, f, h) {
  a = x0
  b = a + h
  while f(a) > f(b)
    a += h
    b += h
  endwhile
}


*Terços*
function tercos(interval, f, precision) {
  a = interval[0]
  b = interval[1]
  while abs(b - a) > precision
      c = a + (b - a) / 3.0
      d = b - (b - a) / 3.0
      if f(c) < f(d)
          b = d
      else
          a = c
  endwhile
}


Método:
  - Pesquisa Sequencial;
  - Terços ou Secção Áurea;
  - Ajuste da Quádrica.
```

Secção Aurea

```
function aurea_sec(interval, f, precision) {
  B = (sqrt(5) - 1) / 2
  a = interval[0]
  b = interval[1]
  c = a + (b - a) * B
  d = b - (b - a) * B
  fc = f(c)
  fd = f(d)

  while abs(b - a) > prec
      if fc < fd:
          b = d
          d = c
          fd = fc
          c = a + B * (b - a)
          fc = f(c)
      else:
          a = c
          c = d
          fc = fd
          d = b - B * (b - a)
          fd = f(d)
  endwhile
}
```

Ajuste da Quádrica

```
function quad_adjust(interval, f) {
  x1 = points[0]
  x3 = points[1]
  x2 = (x1 + x3) / 2.0
  h = x2 – x1
  return x2-(h*(f(x1)-f(x3)))/(2*(f(x1)-2*f(x2)+f(x3)))
}
```

Minimização Multidimensional

Gradiente

```
function gradient(x, y, h, f, dfx, dfy, N) {

  for i in range(1, N + 1):
    xn = x - dfx(x, y) * h
    yn = y - dfy(x, y) * h

    if f(xn, yn) < f(x, y):
        h *= 2
        x = xn
        y = yn
    else:
        h /= 2

  endfor

}
```