MNUM

Telmo Baptista

January 10, 2020

1 Zeros of a real function

1.1 Isolating the roots

In order to ensure that there is atleast one root in an interval $[a, b]$ we must ensure that the sign of $f(a)$ is different than the sign of $f(b)$, this is $f(a) \times f(b) < 0$.
With this we can prove that there is an odd number of solutions in that interval, otherwise if $f(a) \times f(b) > 0$ there is an even number of solutions in that interval, and so it can contain zero solutions.
If we want to guarantee there is exactly one solution in the interval, more information is needed, for example, knowing the derivative of the function then we can study if there's no change of sign of the derivative in that interval, and if so, there is one and only one solution.

1.2 Bisection method

Assuming we have an interval with atleast one root, then we can apply a binary search on the interval and reduce it in order to find the root.
The algorithm, using the initial interval $[a, b]$

1. Calculate the value of the function in the midpoint of the interval, $f(\frac{a+b}{2})$.
   - If the value is null, then the solution is 'exact' and its value is the midpoint $\frac{a+b}{2}$.
   - If $f(a) \times f(\frac{a+b}{2}) < 0$ then the solution is between a and the midpoint, therefore update the right extreme of the interval b with the value of the midpoint.
   - Otherwise, update the left extreme of the interval a with the value of the midpoint.
2. Iterate this process until the stopping criteria is achieved.

1.2.1 Stopping Criteria

One of the following or all or a combination on multiple criteria can be used as the stopping flag for the method.

Absolute precision criteria (interval length)

$$|a - b| \leqslant \varepsilon \tag{1}$$

The criteria to stop the method is the distance between the two extremes being lesser than a given value that determines the absolute precision, ε.

Relative precision criteria

$$|\frac{a-b}{a}| \leqslant \varepsilon \quad or \quad |\frac{a-b}{b}| \leqslant \varepsilon \tag{2}$$

The criteria to stop the method is the ratio of the interval being lesser than a given value that determines the relative precision, ε. This criteria has an advantage regarding the first criteria, being general regardless of the magnitude of the values we are working with.

Function cancellation criteria

$$|f(a) - f(b)| \leqslant \varepsilon \tag{3}$$

The criteria to stop the method is difference between the function values of the interval extremes being lesser than a given value, ε.

Maximum precision criteria

Consists on reducing the interval until the limit precision of the machine is reached. So the maximum absolute error is the limit precision itself.

Number of iterations criteria

The number of iterations to achieve a certain precision can be calculated a priori.

$$\frac{|b-a|}{2^n} <= \varepsilon$$
$$2^n >= \frac{|b-a|}{\varepsilon}$$
$$n >= log_2(\frac{|b-a|}{\varepsilon})$$
$$n \leftarrow log_2(b-a) - log(\varepsilon)$$

in which n is the number of iterations, and ε is the precision wanted.

1.2.2 Choosing the solution

In the end of the method we'll have an interval (small) $[a, b]$. The solution can be chosen in different ways:

- Return the interval where the solution will be.
- Return one of the extremes (can be chosen by comparing residues)
- Return the midpoint (minimizes the maximum absolute error committed)

Bisection Method Implementation

```
def sign(x):
    return 1 if x < 0 else 0

def bissec(f, a, b, precision):
    mid = (a+b)/2
    while(abs(a-b) > precision): # using absolute precision criteria
        if sign(f(a)) != sign(f(mid)): # can be switched to multiplication of
                                       # f(a).f(mid) < 0
            b = mid
        else:
            a = mid
        mid = (a+b)/2
    return mid # return mid for minimization of maximum absolute error
```

1.3 Regula Falsi Method (False Position Method)

This method consists on approximating the function by a straight line that connects the points $(a, f(a))$ and $(b, f(b))$. The equation of the straight line has the following equation:

$$y(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a) \tag{4}$$

And the zero of the function is obtained by:

$$x = \frac{a.f(b) - b.f(a)}{f(b) - f(a)} \tag{5}$$

That will be used to reduce the interval by applying the same strategy of the bisection method, this is, the extreme that has the same sign of $f(x)$ will be updated.
The disadvantage of this technique in relation to that of bisection is that it is not possible to predict a priori the moment to stop, which requires each iteration to test whether the termination condition is already met. This may be a significant waste of time, especially if function values is relatively expeditious.
To study the convergence of the method, assuming the root is well isolated and the sign of the second derivative is constant in the whole interval $[a, b]$ and that the curve is concave, this is $f''(x) > 0$ and if $f''(x) < 0$ we can apply symmetry $f(x) = -f(x)$, that doesn't affect the solution.

- With this the we can observe the point that will get fixed will be the one that has the same sign of the second derivative.
- And the successive approximations of the root, relative to the real root, are located in the side that the sign of the function is opposite of the second derivative.

One disadvantage of this implementation is:
After one of the extremes getting fixed the line that approximates the functions gets closer to a vertical line, and with this the convergence process gets slower.

Regula Falsi Method Implementation

```
def sign(x):
    return 1 if x < 0 else 0

def false_position(a, b, precision, f):
    while abs(a - b) > precision:
        if f(b) - f(a) == 0:
            break

        rr = (a*f(b) - b *f(a)) / (f(b)-f(a))

        if sign(f(a)) != sign(f(rr)):
            b = rr
        else:
            a = rr
    return rr
```

1.4 Newton's Method (Tangent Method)

The previous methods were interval methods, requiring to isolate the root.
With the following methods, including Newton's Method, there's no need to isolate the root, and it's only needed a plausible initial guess of the root.
Newton's method consists in approximating the function by its tangent, and using the zero of the tangent to get a better approximation of the real root.
Being the equation of the tangent defined by:

$$y(x) = f(x_n) + f'(x_n).(x - x_n) \tag{6}$$

So the zero of the function, $y(x) = 0$:

$$0 = f(x_n) + f'(x_n).(x - x_n)$$
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton's Method Implementation
Being f the function and df its derivative, x the first guess.

```
def newton(f, df, x, precision, maxIter):
    last = None
    while (last is None or abs(x - last) > precision) and maxIter:

        last = x

        x = x - f(x)/df(x)

        maxIter -= 1

    return x
```

1.5 Picard Peano Method

The equation we want to solve is $f(x) = 0$. So isolating the variable x, we get:

$$x = g(x) \tag{7}$$

For example:

$$2x^2 - 5x - 3 = 0$$
$$x = \frac{-2x^2 + 3}{-5} \vee x = \pm\sqrt{\frac{5x+3}{2}} \vee x = \frac{5x+3}{2x} = \frac{5}{2} + \frac{3}{2x}$$

Leading to four different recurrent expressions for the method.

Picard Peano Method Implementation

Being g the recurrent Picardo Peano function, x the first guess.

```
def picardo_peano(g, x, precision, maxIter):
    last = None
    while (last is None or abs(x - last) > precision) and maxIter:

        last = x

        x = g(x)

        maxIter -= 1

    return x
```

This method doesn't necessarily require to know the expression for the function $f(x)$ we want to find the root of, as the only necessary expression is the function $g(x)$.

Both Newton's and Picard Peano methods converge if the derivative of the recurrent formula is less than 1. This is:

$$x = g(x) \qquad |g'(x)| < 1 \quad (Picardo\ Peano)$$

As Newton is also a variation of Picardo Peano

$$x = g(x) = x - \frac{f(x)}{f'(x)} \qquad |g'(x)| < 1 \quad (Newton's)$$

1.6 System of Equations

Given a system of equations:

$$\begin{cases} f_1(x,y) = 0 \\ f_2(x,y) = 0 \end{cases} \tag{8}$$

that we want to find the roots of.
Given:

$$\begin{cases} x = x_0 \\ y = y_0 \end{cases}$$

approximate values (guesses) of the solution.
Then we can apply Picard Peano to obtain a system of recurrent expressions we can iterate over to find the roots.

$$\begin{cases} x = g_1(x,y) \\ y = g_2(x,y) \end{cases} \tag{9}$$

This process doens't always converge.

Theorem 1 *If in a neighbourhood ($a \leqslant x \leqslant A, b \leqslant y \leqslant B$) there is one and only one root (ξ, η) and $g_1(x,y)$, $g_2(x,y)$ are differentiable and the values of the function for every pair (x,y) that might occur in the iterative process exists, then the process converges if:*

$$|\frac{\partial g_1}{\partial x}| + |\frac{\partial g_2}{\partial x}| < 1 \qquad\qquad |\frac{\partial g_1}{\partial y}| + |\frac{\partial g_2}{\partial y}|$$

Otherwise, the process might converge or diverge.

This system can also be solved by Newton's Method.

$$\begin{cases} f_1(x,y) = 0 \\ f_2(x,y) = 0 \end{cases} \tag{10}$$

Then the iterative expression is defined by:

$$\begin{cases} x = x_n + h_n \\ y = y_n + k_n \end{cases} \tag{11}$$

Being the Jacobian matrix of the system:

$$J(x_n, y_n) = \begin{bmatrix} f'_{1,x}(x_n, y_n) & f'_{1,y}(x_n, y_n) \\ f'_{2,x}(x_n, y_n) & f'_{2,y}(x_n, y_n) \end{bmatrix} \tag{12}$$

If its determinant isn't null, then replacing the column that corresponds to the variable we are calculating the expression for with $[f_1(x_n, y_n) f_2(x_n, y_n)$. Like so, we obtain:

$$h_n = -\frac{\begin{vmatrix} f_1(x_n, y_n) & f'_{1,y}(x_n, y_n) \\ f_2(x_n, y_n) & f'_{2,y}(x_n, y_n) \end{vmatrix}}{|J(x_n, y_n)|} \qquad k_n = -\frac{\begin{vmatrix} f'_{1,x}(x_n, y_n) & f_1(x_n, y_n) \\ f'_{2,x}(x_n, y_n) & f_2(x_n, y_n) \end{vmatrix}}{|J(x_n, y_n)|}$$

Example:
Given the functions:

$$\begin{cases} f(x) = 2x^2 - y - 5x + 1 \\ g(x) = x - y^2 + 5 \end{cases} \tag{13}$$

And we want to find the solution for:

$$\begin{cases} f(x) = 0 \\ g(x) = 0 \end{cases} \tag{14}$$

Maxima

```
(%i29)f:(-y)+2*x^2-5*x+1;
```

(%o29) $$-y + 2\,x^2 - 5\,x + 1$$

```
(%i30)g:(-y^2)+x+5;
```

(%o30) $$-y^2 + x + 5$$

```
(%i31)J:jacobian([f,g],[x,y]);
```

(%o31) $$\begin{pmatrix} 4\,x - 5 & -1 \\ 1 & -2\,y \end{pmatrix}$$

```
(%i32)DJ:determinant(J);
```

(%o32) $$1 - 2\,\left(4\,x - 5\right)\,y$$

```
(%i33)h:matrix([f],[g]);
```

(%o33) $$\begin{pmatrix} -y + 2\,x^2 - 5\,x + 1 \\ -y^2 + x + 5 \end{pmatrix}$$

```
(%i34)h:addcol(h,col(J,2));
```

(%o34) $$\begin{pmatrix} -y + 2\,x^2 - 5\,x + 1 & -1 \\ -y^2 + x + 5 & -2\,y \end{pmatrix}$$

```
(%i35)k:col(J,1);
```

(%o35) $$\begin{pmatrix} 4\,x - 5 \\ 1 \end{pmatrix}$$

```
(%i36)k:addcol(k,[f,g]);
```

(%o36) $$\begin{pmatrix} 4\,x - 5 & -y + 2\,x^2 - 5\,x + 1 \\ 1 & -y^2 + x + 5 \end{pmatrix}$$

```
(%i37)h:-determinant(h)/DJ;
```

(%o37) $$\frac{y^2 + 2\,\left(-y + 2\,x^2 - 5\,x + 1\right)\,y - x - 5}{1 - 2\,\left(4\,x - 5\right)\,y}$$

```
(%i38)k:-determinant(k)/DJ;
```

(%o38) $$\frac{-\left(4\,x - 5\right)\,\left(-y^2 + x + 5\right) - y + 2\,x^2 - 5\,x + 1}{1 - 2\,\left(4\,x - 5\right)\,y}$$

```
(%i39)fx:ratsimp(x+h);
```

(%o39) $$\frac{y^2 + \left(4\,x^2 - 2\right)\,y + 5}{\left(8\,x - 10\right)\,y - 1}$$

```
(%i40)fy:ratsimp(y+k);
```

(%o40) $$\frac{\left(4\,x - 5\right)\,y^2 + 2\,x^2 + 20\,x - 26}{\left(8\,x - 10\right)\,y - 1}$$

We obtained the expressions to iterate and apply the method. In the program now we only have to do:

$$x = fx(x, y)$$
$$y = fy(x, y)$$

2 System of Linear Equations

2.1 Gaussian Elimination

Given a system of linear equations

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ ..... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{cases} \tag{15}$$

Denoting A as the coefficient matrix, b as the column matrix of the independent terms and x the column matrix of the unknowns, then:

$$A = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{bmatrix} \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{bmatrix}$$

The system can be written in the form $A.x = b$.

Algorithm for Gaussian Elimination:

- Triangularization
  1. Divide the whole row i, including the independent term of the column matrix b by the element on the diagonal in the A matrix, a_{ii}.
  2. For each of one of the next rows i_2, multiply the row i by the coefficient $a_{i_2 i}$ and subtract to the row i_2, including the independent term.
  3. Repeat the whole process till the last row.
- Calculating the solution
  1. Starting at the last row, i, the solution will be $S_i = b_i - \sum_{j=n,j--=1}^{i} S_j \times a_{ij}$.
- Testing the solution (calculating residues)
  1. For each row, multiply the solution of index j by the coefficient in the column j and subtract it from the independent term of the row i.

Gauss Elimination Implementation:

```
def gauss(A, b):
    rows, cols = len(A), len(A[0])
    # triangularization
    for i in range(rows):
        pivot = A[i][i]
        b[i] /= pivot
        for j in range(i, cols):
            A[i][j] /= pivot

        for i2 in range(i+1, rows):
            coef = A[i2][i]
            for j in range(i, cols):
                A[i2][j] -= coef * A[i][j]
            b[i2] -= coef*b[i]

    # obtaining solutions
    solutions = []
    for i in range(rows-1, -1, -1):
        sol = b[i]
        for j in range(cols-1, i, -1):
            sol -= solutions[cols-1 - j]*A[i][j]
        solutions.append(sol)
    solutions.reverse()

    return solutions

def get_residues(A, b, X):
    residues = []
    for i, row in enumerate(A):
        residue = b[i]
        for j, el in enumerate(row):
            residue -= el*X[j]
        residues.append(residue)

    return residues
```

2.2 Error in Gauss Method

- $A.x_0 = b + \varepsilon$, this error is designated as the residue. The residue measures the quality of the resolution of the system.

$$\varepsilon_0 = b - A.x_0 \tag{16}$$

- $A.(x_0 + \delta) = b$, this error is designated as the internal stability of the system, and measures what's left to the solution for it to be exact.

$$\begin{gathered} x = x_0 + \delta \\ A.x = b \\ A.(x_0 + \delta) = b \\ A.\delta = b - A.x_0 \\ A.\delta = \varepsilon_0 \end{gathered}$$

- To study the external stability of the system, this is, the effect errors on the coefficients and independent terms have on the solution we must disturb the system with small arbitrary errors on the coefficients and independent terms (the errors must be small).

$$\begin{gathered} A.x = b \\ \implies (A + \delta A).(x + \delta x) = (b + \delta b) \\ A.x + \delta A.x + A.\delta x + \delta A.\delta x = b + \delta b \end{gathered}$$

if the errors are small enough we can despise the term $\delta A.\delta x$, so we have:

$$\begin{gathered} b + \delta A.x + A.\delta x + 0 = b + \delta b \\ \delta A.x + A.\delta x = \delta b \\ A.\delta x = \delta b - \delta A.x \end{gathered}$$

Calculating the errors:

- Residues (ε)

```
def get_residues(A, b, solution):
        # ... code here
        return residues

residues = get_residues(A, b, solution)
```

- Internal Stability ($A.\delta = \varepsilon$)

```
def gauss(A, b):
        # ... code here
        return solutions

internal_stab = gauss(A, residues)
```

- External Stability ($A.\delta x = \delta b - \delta A.x_0$)
  Maxima

```
(%i1)size:3;
```

(%o1) 3

```
(%i2)DA:zeromatrix(size,size)+da;
```

(%o2) $$\begin{pmatrix} da & da & da \\ da & da & da \\ da & da & da \end{pmatrix}$$

```
(%i3)DB:zeromatrix(size,1)+db;
```

(%o3) $$\begin{pmatrix} db \\ db \\ db \end{pmatrix}$$

```
(%i4)X0:[x0,y0,z0];
```

(%o4) $[x_0, y_0, z_0]$

```
(%i5)B:DB-DA . X0;
```

(%o5) $$\begin{pmatrix} -da\,z_0 - da\,y_0 - da\,x_0 + db \\ -da\,z_0 - da\,y_0 - da\,x_0 + db \\ -da\,z_0 - da\,y_0 - da\,x_0 + db \end{pmatrix}$$

Python

```
def gauss(A, b):
        # ... code here
        return solutions

da = arb_error_da
db = arb_error_db
x0, y0, z0 = solution[0], solution[1], solution[2]
B = [
                -da*z0-da*y0-da*x0+db,
                -da*z0-da*y0-da*x0+db,
                -da*z0-da*y0-da*x0+db
        ]

external_stab = gauss(A, B)
```

2.3 Iterative Methods

Given the system

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ ..... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{cases}$$

Which can be written in the form:

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \ (1 \leqslant i \leqslant n)$$

Isolating a variable in the first member:

$$x_i = \frac{\sum_{j=1, i \neq j}^{n} -a_{ij}x_j + b_i}{a_{ii}} \ (1 \leqslant i \leqslant n)$$

Obtaining a Picard-Peano recurrent formula to calculate each variable x_i, so to test the convergence of the iterative process, the derivative of the function must be less than 1, in absolute value

Derivating the expression above:

$$|g'| = \frac{\sum_{j=1, i \neq j}^{n} |a_{ij}|}{|a_{ii}|} \ (1 \leqslant i \leqslant n)$$

If the conditions of convergence verify $|g'| < 1$, then the matrix is diagonally dominant, this is:

$$|a_{ii}| > \sum_{j=1, i \neq j}^{n} |a_{ij}| \ (1 \leqslant i \leqslant n)$$

2.3.1 Gauss-Jacobi Iterative Method

Apply the recurrent formula obtained using the original values of the iteration k, this is:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left[- \sum_{j=1, j \neq i}^{n} a_{ij}x_j^{(k-1)} + b_i \right], (1 \leqslant i \leqslant n) \tag{17}$$

Python Implementation

```
import copy
def gauss_jacobi(A, b, x0, iterations):
    rows, cols = len(A), len(A[0])
    for _ in range(iterations):
        x = copy.deepcopy(x0)
        for i in range(rows):
            x[i] = (b[i] - sum(A[i][j]*x0[j] for j in range(cols) if i != j)) / A[i][i] # x is the arra
        x0 = copy.deepcopy(x)

    return x0
```

2.3.2 Gauss-Seidel Iterative Method

Apply the recurrent formula obtained using the most recent values possible, this is:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left[- \sum_{j=1, j<i}^{n} a_{ij} x_j^{(k)} - \sum_{j=1, j>i}^{n} a_{ij} x_j^{(k-1)} + b_i \right], (1 \leqslant i \leqslant n) \tag{18}$$

Python Implementation

```
def gauss_seidel(A, b, x, iterations):
    rows, cols = len(A), len(A[0])
    for _ in range(iterations):
        for i in range(rows):
            x[i] = (b[i] - sum(A[i][j]*x[j] for j in range(cols) if i != j)) / A[i][i] # x is the array

    return x
```

It's possible to speed the convergence process of the Gauss-Seidel method using a relaxation factor w (SOR variant - Successive Overrelaxation):

$$x_i^{(k)} = w \left(\frac{1}{a_{ii}} \left[- \sum_{j=1, j<i}^{n} a_{ij} x_j^{(k)} - \sum_{j=1, j>i}^{n} a_{ij} x_j^{(k-1)} + b_i \right] \right) + (1 - w) x_i^{(k-1)}, (1 \leqslant i \leqslant n) \tag{19}$$

Python Implementation

```
def gauss_seidel_sor(A, b, x, w, iterations):
    rows, cols = len(A), len(A[0])
    for _ in range(iterations):
        for i in range(rows):
            x[i] = w * (b[i] - sum(A[i][j]*x[j] for j in range(cols) if i != j)) / A[i][i] + (1-w)*x[i]

    return x
```

3 Quadrature and Cubature

Indefinite Integral:

$$F_a(x) = \int_a^x f(t).dt \tag{20}$$

results in a table of values that represents the primitive function F_a.
Definite integral:

$$F_a(x) = \int_a^b f(t).dt \tag{21}$$

results in an unique numeric value.

3.1 Trapeze Rule

The area of a trapeze is defined as $h \times \frac{b+B}{2}$. Dividing the area underneath the function $f(x)$ in n trapezes with $h = dx$, then:

$$\begin{aligned}\int_{x_0}^{x_n} f(x)dx &= \sum_{i=0}^{n} \frac{f(x_i) \times f(x_{i+1})}{2} dx \\ &= \frac{y_0 + y_1}{2} \times dx + \frac{y_1 + y_2}{2} \times dx + ... + \frac{y_{n-1} + y_n}{2} \times dx \\ &= \frac{dx}{2} \times \left[y_0 + y_n + 2 \times \sum_{i=1}^{n-1} y_i \right]\end{aligned} \tag{22}$$

Python Implementation

```
#singular integral (quadrature)
def int_trapz(f, t0, t1, n):
    h = (t1-t0)/n
    return h/2 * (f(t0) + f(t1) + 2*sum(f(t0 + i*h) for i in range(1, n)))
```

3.1.1 Cubature

To calculate double integrals:

$$\int_a^A dx \int_b^B f(x,y)dy \tag{23}$$

Apply Trapeze Rule to the integral we can obtain an expression to calculate double integrals:

$$\begin{aligned}\int_a^A dx \int_b^B f(x,y)dy = \\ &= \int_a^A \frac{h_y}{2} \left[f(x,y_0) + f(x,y_n) + 2 \times \sum_{i=1}^{n-1} f(x,y_i) \right] \\ &= \frac{h_y}{2} \left[\int_a^A f(x,y_0) + \int_a^A f(x,y_n) + 2 \times \sum_{i=1}^{n-1} \int_a^A f(x,y_i) \right]\end{aligned} \tag{24}$$

Trapeze Rule Cubature Implementation

```
#double integral (cubature)
    #one point fixed
def dint_trapz_fixedy(func, x0, x1, y, n):
    hx = (x1-x0)/n
    return hx/2 * (func(x0, y) + func(x1, y) + 2*sum(func(x0 + i*hx, y)
                    for i in range(1, n)))

    #outter function
def dint_trapz(func, x0, x1, y0, y1, n):
    hy = (y1-y0)/n
    return hy/2 * (dint_trapz_fixedy(func, x0, x1, y0, n) +
            dint_trapz_fixedy(func, x0, x1, y1, n) +
            2*sum(dint_trapz_fixedy(func, x0, x1, y0 + i*hy, n)
            for i in range(1, n)))
```

3.1.2 Error Control

To estimate the error, calculate the process using three different steps (h).
Using h as the integration step to which will result in S.
Using $h' = \frac{h}{2}$ as the integration step to which will result in S'.
Using $h'' = \frac{h'}{2} = \frac{h}{4}$ as the integration step to which will result in S''.
The convergence quotient (QC) should be approximately 4 (2th order method -¿ $2^o rder$).

$$QC = \frac{S' - S}{S'' - S'} \approx 4 \tag{25}$$

If this condition is met, then the error ε'' can be estimated by:

$$S'' - S' \approx 3\varepsilon'' \longrightarrow \varepsilon'' \approx \frac{S'' - S'}{3} \tag{26}$$

3.2 Simpson Rule

The number of intervals must be even for the Simpson Rule to be applied.

$$\int_{x_0}^{x_{2n}} y.dx = \frac{h}{3} \times \left[y_0 + y_{2n} + 4 \times \sum_{i=1,i+=2}^{2n-1} y_i + 2 \times \sum_{i=2,i+=2}^{2n-2} y_i \right] \tag{27}$$

Python Implementation

```
#singular integral (quadrature)
def int_simpson(func, t0, t1, n):
    h = (t1-t0)/n
    return h/3 * (func(t0) + func(t1) + sum(4*func(t0 + i*h) if i%2
                                    else 2*func(t0 + i*h) for i in range(1, n)))
```

3.2.1 Cubature

To calculate double integrals:

$$\int_a^A dx \int_b^B f(x,y)dy \tag{28}$$

Apply Simpson Rule to the integral we can obtain an expression to calculate double integrals:

$$\begin{aligned}
&\int_a^A dx \int_b^B f(x,y)dy = \\
&= \int_a^A \frac{h_y}{3} \left[f(x,y_0) + f(x,y_n) + 4 \times \sum_{i=1,i+=2}^{2n-1} f(x,y_i) + 2 \times \sum_{i=2,i+=2}^{2n-2} f(x,y_i) \right] \\
&= \frac{h_y}{3} \left[\int_a^A f(x,y_0) + \int_a^A f(x,y_n) + 4 \times \sum_{i=1,i+=2}^{2n-1} \int_a^A f(x,y_i) + 2 \times \sum_{i=2,i+=2}^{2n-2} \int_a^A f(x,y_i) \right]
\end{aligned} \tag{29}$$

Simpson Rule Cubature Implementation

```
#double integral (cubature)
    #one point fixed
def dint_simpson_fixedy(func, x0, x1, y, n):
    hx = (x1-x0)/n
    return hx/3 * (func(x0, y) + func(x1, y) + sum(4*func(x0 + i*hx, y) if i%2 else 2*func(x0 + i*hx, y

    #outter function
def dint_simpson(func, x0, x1, y0, y1, n):
    hy = (y1-y0)/n
    return hy/3 * (dint_simpson_fixedy(func, x0, x1, y0, n) +
            dint_simpson_fixedy(func, x0, x1, y1, n) +
            sum(4*dint_simpson_fixedy(func, x0, x1, y0 + i*hy, n)
            if i%2 else
            2*dint_simpson_fixedy(func, x0, x1, y0 + i*hy, n)
            for i in range(1, n)))
```

3.2.2 Error Control

To estimate the error, calculate the process using three different steps (h).
Using h as the integration step to which will result in S.
Using $h' = \frac{h}{2}$ as the integration step to which will result in S'.
Using $h'' = \frac{h'}{2} = \frac{h}{4}$ as the integration step to which will result in S''.
The convergence quotient (QC) should be approximately 16 (4th order method $\rightarrow 2^{order}$).

$$QC = \frac{S' - S}{S'' - S'} \approx 16 \tag{30}$$

If this condition is met, then the error ε'' can be estimated by:

$$S'' - S' \approx 15\varepsilon'' \longrightarrow \varepsilon'' \approx \frac{S'' - S'}{15} \tag{31}$$

General formula for error:

For a method of order k.

Calculate S with step h.

Calculate S' with step $h' = \frac{h}{2}$.

Calculate S'' with step $h'' = \frac{h'}{2}$.

$$QC \approx \frac{S' - S}{S'' - S'} \approx 2^k$$

If this condition is met, then the absolute error ε'' can be estimated:

$$\varepsilon'' = \frac{S'' - S'}{2^k - 1}$$

And the relative error can also be calculated:

$$\xi'' = \frac{\varepsilon''}{S''}$$

4 Ordinary Differential Equations (ODE)

Given the ordinary differential equation:

$$y' = \frac{dy}{dx} = f(x, y) \tag{32}$$

The solution of this equation is any curve $g(x, y)$ so that:

$$g'_x(x, y) = \frac{dg(x, y)}{dx} = f(x, y) \tag{33}$$

4.1 Euler's Method

This method consists on using the finite increments formula.
Given an infinitesimal distance dx and dy, we know that the derivative of a function g in order to x is:

$$g'_x(x, y) = \frac{dy}{dx} = f(x, y) \tag{34}$$

We can then transform this equation on using small finite distances Δx and Δy instead of infinitesimal distance.

$$\frac{\Delta y}{\Delta x} = f(x, y) \Rightarrow \Delta y = \Delta x f(x, y) \tag{35}$$

Being Δx arbitrary, we obtain the value of the increments to iterate the process

$$\begin{cases} y_{n+1} \leftarrow y_n + \Delta y \\ x_{n+1} \leftarrow x_n + \Delta x \end{cases} \tag{36}$$

until we've covered all of the interval we wanted to find.
Euler's method is a first order method, therefore the error is proportional to h.
As seen before we can calculate the convergence quotient, QC:

$$\frac{S' - S}{S'' - S'} \approx 2^{order} = 2^1 = 2 \tag{37}$$

If the quotient is approximately 2, than the absolute error can be estimated:

$$\varepsilon'' \approx \frac{S'' - S'}{2^{order} - 1} = S'' - S' \tag{38}$$

Euler Method Implementation

```
def euler(f, x, y, dx): # step function
    return dx * f(x, y)
```

Example of the method:
Given the equation $\frac{dy}{dx} = f(x, y) = sin(x)$, and the initial conditions $x_0 = 0$, $y_0 = 0$ and a step of $\Delta x = 0.1$ calculate the values of the function in the interval $[0, 5]$.
Resolution:
This function is simple enough to determine the analytical solution, as:

$$\int dy = \int f(x, y)dx \Rightarrow$$
$$y = \int sin(x)dx \Rightarrow$$
$$y = -cos(x) + C$$

For the initial conditions of the problem:

$$0 = -cos(0) + C \Rightarrow C = 1$$

The answer for this question then is the function:

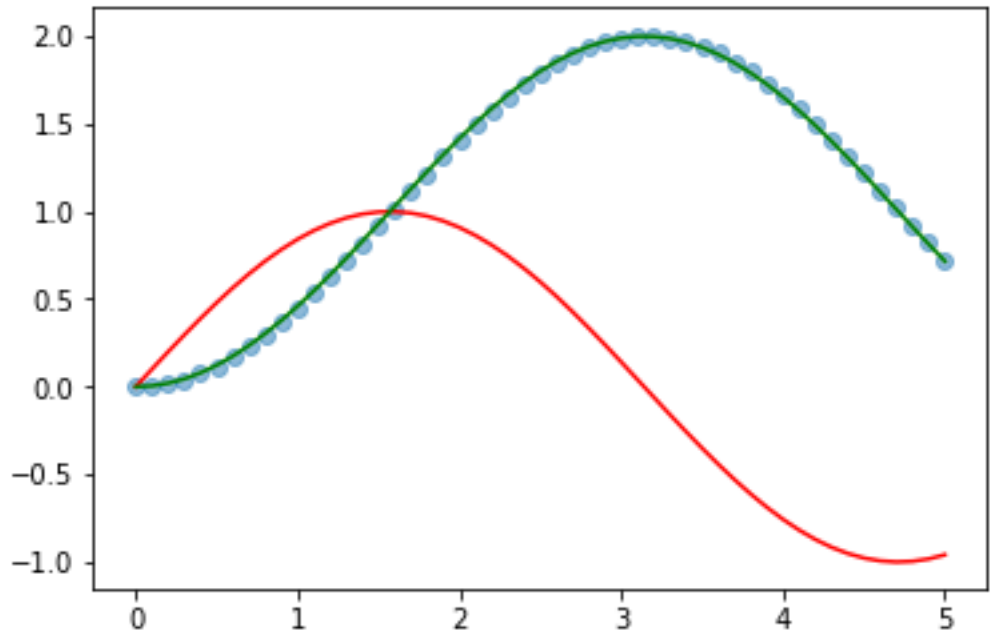
$$y = -cos(x) + 1$$

However this isn't a numerical resolution, and for complex equations this wouldn't be as easy, so let's build a table of points that will approximate the function that we obtained analytically.

```
import math
import matplotlib.pyplot as plt
import numpy as np
function = lambda x, y=0: math.sin(x)

def float_equal(f1, f2, tolerance):
    return abs(f1-f2) <= tolerance

def euler(f, x, y, dx):
    return dx * f(x, y)

def solve_ode(f, x0, y0, deltax, n, tolerance):

    x = [x0] # table of points for X
    y = [y0] # table of points for Y
    x1 = x0
    y1 = y0
    iteration = 1
    xf = x0 + n*deltax
    while(abs(x1-xf) >= tolerance): #number of iterations
        print(f"Iteration: {iteration}")
        iteration+=1
        print(f"dx: {deltax}")

        deltay = euler(f, x1, y1, deltax) # h = deltax

        print("dy: ", deltay)

        deltay1 = 0
        for i in range(2): # h' = h/2
            deltay1 += euler(f, x1+deltax/2*i, y1+deltay1, deltax/2)

        print("dy': ", deltay1)

        deltay2 = 0
        for i in range(4): #h'' = h'/2
            deltay2 += euler(f, x1+deltax/4*i, y1+deltay2, deltax/4)
        print("dy'': ", deltay2)

        QC = (deltay1 - deltay) / (deltay2 - deltay1) # (S'-S)/(S''-S')
        error = 1
```

```
            print(f"QC: {QC}")
            if float_equal(QC, 2, 0.1): #2^order = 2^1 = 2
                absolute_error = abs((deltay2-deltay1))
                print(f"Absolute error: {absolute_error}")
                error = abs(absolute_error/deltay2)
                print(f"Relative Error: {error}")
            if (absolute_error <= tolerance):
                x1+=deltax
                y1+=deltay2

                x.append(x1)
                y.append(y1)
            else:
                deltax *= 0.95 #reduce h

    return x, y

x0 = float(input("X0: "))
y0 = float(input("Y0: "))
deltax = float(input("dx: "))
points = int(input("Number of iterations: "))
tol = float(input("Tolerance: "))

x_points, y_points = solve_ode(function, x0, y0, deltax, points, tol)

print("\tX\t\t\t\t\tY")
for x, y in zip(x_points, y_points):
    print(f"\t{x:.8f}\t\t\t\t{y:.8f}")

np_x = np.arange(0,5.1,0.1)   # start,stop,step
np_y = np.sin(np_x)
np_y2 = -np.cos(np_x) + 1
plt.scatter(x_points, y_points, alpha=0.5)
plt.plot(np_x, np_y, color='red')
plt.plot(np_x, np_y2, color='green')
plt.show()
```

In which we obtain the following result:



In red it's the function $f(x, y) = sin(x)$, in green the analytical solution and in blue the result of the tabular function we obtained.

X	Y
0.00000000	0.00000000
0.10000000	0.00374766
0.20000000	0.01744902
0.30000000	0.04096718
0.40000000	0.07406717
0.50000000	0.11641824
0.60000000	0.16759726
0.70000000	0.22709284
0.80000000	0.29431054
0.90000000	0.36857874
1.00000000	0.44915536
1.10000000	0.53523533
1.20000000	0.62595855
1.30000000	0.72041854
1.40000000	0.81767150
1.50000000	0.91674571
1.60000000	1.01665125
1.70000000	1.11638989
1.80000000	1.21496508
1.90000000	1.31139189
2.00000000	1.40470686
2.10000000	1.49397761
2.20000000	1.57831218
2.30000000	1.65686792
2.40000000	1.72885994
2.50000000	1.79356890
2.60000000	1.85034827
2.70000000	1.89863072
2.80000000	1.93793383
2.90000000	1.96786489
3.00000000	1.98812485
3.10000000	1.99851127
3.20000000	1.99892037
3.30000000	1.98934808
3.40000000	1.96989002
3.50000000	1.94074062
3.60000000	1.90219113
3.70000000	1.85462673
3.80000000	1.79852266
3.90000000	1.73443949
4.00000000	1.66301752
4.10000000	1.58497039
4.20000000	1.50107790
4.30000000	1.41217829
4.40000000	1.31915980
4.50000000	1.22295186
4.60000000	1.12451574
4.70000000	1.02483498
4.80000000	0.92490555
4.90000000	0.82572592
5.00000000	0.72828706

4.2 Improved Euler's Method

In Euler's method it is used the value of the derivative in only one point of the extremes, which can to errors. The solution to this problem is suggested by Lagrange's Theorem:

$$\Delta y = f'_{mean} \Delta x \tag{39}$$

Considering we have the pairs (x_{n-1}, y_{n-1}) and (x_n, y_n) in order to calculate the pair (x_{n+1}, y_{n+1}) we can:

- Calculate the slope on the point (x_n, y_n) and use its value to calculate from the point (x_{n-1}, y_{n-1}) an estimate for the point in (x_{n+1}, y_{n+1}), resulting in (x_{n+1}, p_{n+1}), begin p_{n+1} the estimate for the point.
- Using the point estimated p_{n+1} calculate the slope in (x_{n+1}, p_{n+1}), and then calculate the average slope between the slope of the point (x_n, y_n) and the point (x_{n+1}, p_{n+1}), and use it to calculate the definitive step.

Calculate the slope in (x_n, y_n)

$$y'_n = f(x_n, y_n)$$

Calculate the estimate for the point in $n+1$

$$p_{n+1} = y_{n-1} + 2y'_n \times \Delta x$$

Calculate the derivative on the estimated point (slope)

$$p'_{n+1} = f(x_{n+1}, p_{n+1})$$

Calculate the definitive step using the average slope

$$\Delta y_n = \frac{p'_{n+1} + y'_n}{2} \times \Delta x$$

Apply the reccurent expressions to obtain the next points

$$\begin{cases} y_{n+1} \leftarrow y_n + \Delta y_n \\ x_{n+1} \leftarrow x_n + \Delta x_n \end{cases}$$

Improved Euler's Method Implementation

```
#first order
def euler_improved(f, x, y, dx, prev_y):
    if prev_y is None: #first iteration there's no y_{n-1}
        prev_y = y - euler(f, x, y, dx) # y - dx * f(x, y)

    k1 = func(x, y)
    k2 = func(x + dx, prev_y + 2 * dx * k1)
    k = (k1+k2)/2
    return dx*k
```

For higher order differential equations:

$$\begin{cases} y_1' = f_1(x, y_1, y_2, ..., y_n) \\ y_2' = f_2(x, y_1, y_2, ..., y_n) \\ \quad ... \\ y_n' = f_n(x, y_1, y_2, ..., y_n) \end{cases} \tag{40}$$

The algorithm is applied to every single variable sequentially:

Calculate the slopes of the point $(x^{(k)}, y_1^{(k)}, y_2^{(k)}, ..., y_n^{(k)})$

$$\begin{cases} y_1^{'(k)} = f_1(x^{(k)}, y_1^{(k)}, y_2^{(k)}, ..., y_n^{(k)}) \\ y_2^{'(k)} = f_2(x^{(k)}, y_1^{(k)}, y_2^{(k)}, ..., y_n^{(k)}) \\ \quad ... \\ y_n^{'(k)} = f_n(x^{(k)}, y_1^{(k)}, y_2^{(k)}, ..., y_n^{(k)}) \end{cases}$$

Calculate the estimate for the point in $k+1$

$$\begin{cases} p_1^{(k+1)} = y_1^{k-1} + 2y_1^{'(k)} \times \Delta x \\ p_2^{(k+1)} = y_2^{k-1} + 2y_2^{'(k)} \times \Delta x \\ \quad ... \\ p_n^{(k+1)} = y_n^{k-1} + 2y_n^{'(k)} \times \Delta x \end{cases}$$

Calculate the derivative on the estimated points (slope)

$$\begin{cases} p_1^{'(k+1)} = f_1(x^{(k+1)}, p_1^{(k+1)}, p_2^{(k+1)}, ..., p_n^{(k+1)}) \\ p_2^{'(k+1)} = f_2(x^{(k+1)}, p_1^{(k+1)}, p_2^{(k+1)}, ..., p_n^{(k+1)}) \\ \quad ... \\ p_n^{'(k+1)} = f_n(x^{(k+1)}, p_1^{(k+1)}, p_2^{(k+1)}, ..., p_n^{(k+1)}) \end{cases}$$

Calculate the definitive steps using the average slopes

$$\begin{cases} \Delta y_1^{(k)} = \frac{p_1^{'(k+1)} + y_1^{'(k)}}{2} \times \Delta x \\ \Delta y_2^{(k)} = \frac{p_2^{'(k+1)} + y_2^{'(k)}}{2} \times \Delta x \\ \quad ... \\ \Delta y_n^{(k)} = \frac{p_n^{'(k+1)} + y_n^{'(k)}}{2} \times \Delta x \end{cases}$$

Apply the reccurent expressions to obtain the next points

$$\begin{cases} y_1^{(k+1)} \leftarrow y_1^{(k)} + \Delta y_1^{(k)} \\ y_2^{(k+1)} \leftarrow y_2^{(k)} + \Delta y_2^{(k)} \\ \quad ... \\ y_n^{(k+1)} \leftarrow y_n^{(k)} + \Delta y_n^{(k)} \\ x^{(k+1)} \leftarrow x^{(k)} + \Delta x^{(k)} \end{cases}$$

Second Order Euler's Improved Method Implementation

```
def euler_improved(fy, fz, x, y, z, dx, prev_y, prev_z):
    if prev_y is None: # first iteration theres no y_{n-1}
        prev_y = y - euler(fy, x, y, z, dx)
    if prev_z is None: # first iteration theres no z_{n-1}
        prev_z = z - euler(fz, x, y, z, dx)

    y1 = fy(x, y, z)
    z1 = fz(x, y, z)

    y2 = fy(x + dx, prev_y + 2 * dx * y1, prev_z + 2 * dx * z1)
    z2 = fz(x + dx, prev_y + 2 * dx * y1, prev_z + 2 * dx * z1)

    y = (y1+y2)/2
    z = (z1+z2)/2
    return dx*y, dx*z
```

Improved Euler's method is a third order method, therefore the error is proportional to h^3.
As seen before we can calculate the convergence quotient, QC:

$$\frac{S' - S}{S'' - S'} \approx 2^{order} = 2^3 = 8 \tag{41}$$

If the quotient is approximately eight, than the absolute error can be estimated:

$$\varepsilon'' \approx \frac{S'' - S'}{2^{order} - 1} = \frac{S'' - S'}{7} \tag{42}$$

4.3 Runge-Kutta's Method

4.3.1 Second Order Runge-Kutta Method

1. Calculate $y'_n = f(x_n, y_n)$
2. Create an estimation for y' in the middle of the interval

$$y'_a = f(x_n + \frac{\Delta x_n}{2}, y_n + \frac{\Delta x_n}{2}.y'_n)$$

3. Calculate the increment of y using y'_a:

$$\Delta y_{n+1} = y'_a \Delta x_n$$

4. Calculate the next point:

$$\begin{cases} y_{n+1} \leftarrow y_n + \Delta y_n \\ x_{n+1} \leftarrow x_n + \Delta x_n \end{cases}$$

RK2 is a second order method, therefore the error is proportional to h^2.
As seen before we can calculate the convergence quotient, QC:

$$\frac{S' - S}{S'' - S'} \approx 2^{order} = 2^2 = 4 \tag{43}$$

If the quotient is approximately 4, than the absolute error can be estimated:

$$\varepsilon'' \approx \frac{S'' - S'}{2^{order} - 1} = \frac{S'' - S'}{3} \tag{44}$$

RK2 Method Implementation

```
#first order
def rk2(f, x, y, dx):
    y1 = dx * f(x, y)
    y2 = f(x + dx/2, y + y1/2)
    return y2*dx
```

Similiar to Euler's Improved, if we have a system of ODE's (high order) then we need to apply the method to each variable sequentially:

RK2 Second Order ODE Method Implementation

```
#second order
def rk2(fy, fz, x, y, z, dx):
    y1 = dx * fy(x, y, z)
    z1 = dx * fz(x, y, z)

    y2 = fy(x + dx/2, y + y1/2, z + z1/2)
    z2 = fz(x + dx/2, y + y1/2, z + z1/2)
    return y2*dx, z2*dx
```

4.3.2 Fourth Order Runge-Kutta Method

1. Calculate the first estimation δ_1 for the increment Δy_n using y'_n:

$$\delta_1 = \Delta x_n . f(x_n, y_n)$$

2. Using the previous estimation, calculate an estimation for y' in the middle step, and use this value to calculate the second estimation δ_2 for the increment Δy_n:

$$\delta_2 = \Delta x_n . f(x_n + \frac{\Delta x_n}{2}, y_n + \frac{\delta_1}{2})$$

3. Calculate a second estimation for y'_n in the middle step, using the previous estimation, and using its value calculate the third estimation δ_3 for the increment Δy_n:

$$\delta_3 = \Delta x_n . f(x_n + \frac{\Delta x_n}{2}, y_n + \frac{\delta_2}{2})$$

4. Calculate the estimation for y'_n in the final step, using the previous one, and obtaining the fourth estimation δ_4 for the increment Δy_n:

$$\delta_4 = \Delta x_n . f(x_n + \Delta x_n, y_n + \delta_3)$$

5. Calculate the weighted average of the estimations:

$$\Delta y_n = \frac{\delta_1}{6} + \frac{\delta_2}{3} + \frac{\delta_3}{3} + \frac{\delta_4}{6}$$

6. Calculate the next point:

$$\begin{cases} y_{n+1} \leftarrow y_n + \Delta y_n \\ x_{n+1} \leftarrow x_n + \Delta x_n \end{cases}$$

RK4 is a fourth order method, therefore the error is proportional to h^4.
As seen before we can calculate the convergence quotient, QC:

$$\frac{S' - S}{S'' - S'} \approx 2^{order} = 2^4 = 16 \tag{45}$$

If the quotient is approximately 16, than the absolute error can be estimated:

$$\varepsilon'' \approx \frac{S'' - S'}{2^{order} - 1} = \frac{S'' - S'}{15} \tag{46}$$

RK4 Method Implementation

```
#first order
def rk4(f, x, y, dx):
    y1 = dx * f(x, y)

    y2 = dx * f(x + dx/2, y + y1/2)

    y3 = dx * f(x + dx/2, y + y2/2)

    y4 = dx * f(x + dx, y + y3)

    return y1/3 + y2/6 + y3/6 + y4/3
```

Similiar to RK2, if we have a system of ODE's (high order) then we need to apply the method to each variable sequentially:

RK4 Second Order ODE Method Implementation

```
#second order
def rk4(fy, fz, x, y, z, dx):
    y1 = dx * fy(x, y, z)
    z1 = dx * fz(x, y, z)

    y2 = dx * fy(x + dx/2, y + y1/2, z + z1/2)
    z2 = dx * fz(x + dx/2, y + y1/2, z + z1/2)

    y3 = dx * fy(x + dx/2, y + y2/2, z + z2/2)
    z3 = dx * fz(x + dx/2, y + y2/2, z + z2/2)

    y4 = dx * fy(x + dx, y + y3, z + z3)
    z4 = dx * fz(x + dx, y + y3, z + z3)

    return y1/3 + y2/6 + y3/6 + y4/3, z1/3 + z2/6 + z3/6 + z4/3
```

5 Optimization

It will be presented techniques to find the extremes of a function, such as the minimum of the a function.

$$min[f(x)] \tag{47}$$

5.1 Unidimensional Optimization

Consisting of interval techniques, unidimensional optimization is essentially divided into three parts:

1. Searching an interval that contains the minimum of the function.
2. Reducing the interval
3. Adjusting the solution

5.1.1 Unidimensional Search

A simple method of searching the interval that contains one minimum of the function is to follow the direction that the function decreases until the function increases again.

Unidimensional Search Implementation - Constant Step

Let *guess* be a plausible number near the minimum, and h the step that will used to increase the interval until the interval contains the minimum.

```
""" CONSTANT STEP """
def search(f, guess, h):
    x0 = guess
    x1 = guess + h
    f0 = f(x0)
    f1 = f(x1)
    invert = False # go the right assuming h > 0
    if (f0 < f1):
        h = -h
        x1 = guess + h
        f1 = f(x1)
        invert = True # the function increases in this direction so goes in opposite direction

    x2 = x1 + h
    f2 = f(x2)

    while f1 > f2:
        x0 = x1
        f0 = f1
        x1 = x2
        f1 = f2
        x2 = x1 + h
        f2 = f(x2)

    return (x2, x0) if invert else (x0, x2)
```

A variable step can also be used in the search, this is, at each iteration the step increases by a factor k.

Unidimensional Search Implementation - Variable Step

```
""" VARIABLE STEP """
def search_var(f, guess, h, k):
    x0 = guess
    x1 = guess + h
    f0 = f(x0)
    f1 = f(x1)
    invert = False
    if (f0 < f1):
        h = -h
        x1 = guess + h
        f1 = f(x1)
        invert = True
    h *= k
    x2 = x1 + h
    f2 = f(x2)

    while f1 > f2:
        x0 = x1
        f0 = f1
        x1 = x2
        f1 = f2
        h *= k
        x2 = x1 + h
        f2 = f(x2)

    return (x2, x0) if invert else (x0, x2)
```

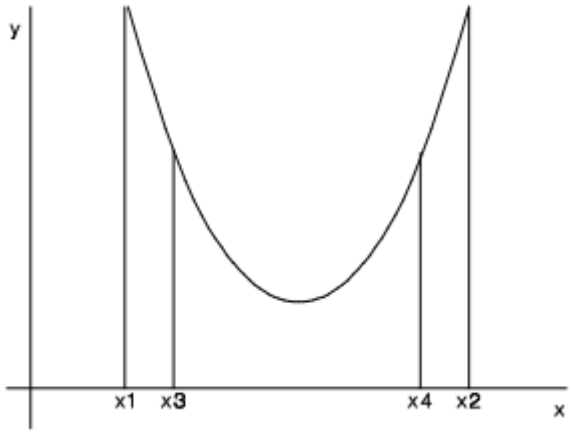
5.1.2 Reducing Interval

Dividing the interval into 4 parts and following the Golden Ratio so that the distance between each part is similar enough in order to increase the performance when discarding sections of the interval. This is, when discarding the leftmost or rightmost it won't have different impacts on the performance of the method.

$$\boldsymbol{Golden\ Ratio} = \frac{\sqrt{5}-1}{2} \approx 0.6180$$

Considering now 4 points: (x_1, x_3, x_4, x_2), in which x_1 and x_2 are known and are the extremes of the interval, the other points x_3 and x_4 will be calculated following the golden ratio.
The point x_3 is at a distance proportional to the golden ratio from x_2, the same applies for the point x_4 and x_1.
Let the interval that contains the minimum be $[a, b]$ and the Golden Ratio g_ratio, then we obtain the following division:

$$\begin{cases} x_1 = a \\ x_2 = b \\ x_3 = x_2 - g\_ratio \times |b - a| \\ x_4 = x_1 + g\_ratio \times |b - a| \end{cases} \tag{48}$$



We can now apply a method to find the interval by comparing the value of the function on these points and then discard the section that doesn't contain the minimum.

Trisection Method (Thirds Method)

This method consists on comparing the function value of x_3 and x_4 to then discard on the sections, $[x_1, x_3[$ or $]x_4, x_2]$.
If the value of the function on x_3 is lesser than in x_4 then the minimum is certainly contained on the interval $[x_1, x_4]$ and we can discard $]x_4, x_2]$. Otherwise the minimum is contained on the interval $[x_3, x_2]$ and we discard $[x_1, x_3[$.

Trisection Method Implementation

```
def trisection(f, lower, upper, error):
    a = lower
    b = upper
    c = b - g_ratio * (b-a)
    d = a + g_ratio * (b-a)

    while(abs(b-a) > error): # use abs(b-a)/b or abs(b-a)/a for relative error

        if f(c) < f(d):
            b = d
        else:
            a = c

        c = b - g_ratio * (b-a)
        d = a + g_ratio * (b-a)

    return [a, b]
```

Golden Rule Method

In comparison to the trisection method that doesn't necessarily need to follow the golden ratio, the Golden Rule method or Golden Section method is an improvement of the previous one, as the number of calculations needed reduces due to saving points using Golden Ratio properties.
Chosen x_3 and x_4 so that $x_3 - x_1 = (g\_ratio)^2(x_2 - x_1)$ and $x_4 - x_1 = g\_ratio(x_2 - x_1)$.
Note that $(g\_ratio)^2 = 1 - g\_ratio$, with this we obtain:

- If $f(x_3) < f(x_4)$ then minimum is in $[x_1, x_4]$ and the next step implies $x_2 \leftarrow x_4$, $x_4 \leftarrow x_3$ and calculate the new value of x_3 using the rule $x_3 - x_1 = (g\_ratio)^2(x_2 - x_1)$
- If $f(x_3) > f(x_4)$ then minimum is in $[x_3, x_2]$ and the next step implies $x_1 \leftarrow x_3$, $x_3 \leftarrow x_4$ and calculate the new value of x_4 using the rule $x_4 - x_1 = g\_ratio(x_2 - x_1)$

Golden Rule Implementation

```
def aurea(f, lower, upper, error):
    a = lower
    b = upper
    c = b - g_ratio * (b-a)
    d = a + g_ratio * (b-a)
    fc = f(c)
    fd = f(d)
    while(abs(b-a) > error): # use abs(b-a)/b or abs(b-a)/a for relative error
        if fc < fd: # calculate new c
            b = d
            d = c
            fd = fc
            c = b - g_ratio * (b-a)
            fc = f(c)
        else:                   #calculate new d
            a = c
            c = d
            fc = fd
            d = a + g_ratio * (b-a)
            fd = f(d)

    return [a, b]
```

5.1.3 Adjusting

Both methods to reduce interval only gives us information about an interval where the solution is, we don't know what's the better minimum of the function, it could be x_1 or x_2 or other point in the interval.
To solve this, we have adjusting methods.

Quadratic Interpolation Method

Consists on approximating the curve by a parable and using the minimum of the parable as a better estimation for the minimum of the function.
Dividing the interval into four equidistant sections, we now have four points: x_1, x_3, x_4, x_2.
If the value of the function in x_3 is lesser than in x_4 then discard the point x_2, otherwise discard point x_1.
With the remaining three points, we will call now x_1, x_2 and x_3, and its function values, f_1, f_2 and f_3 we can build a parable to approximate the function, the parable has the following equation:

$$y = y_1 + \frac{x - x_1}{h}.(f_2 - f_1) + \frac{(x - x_1).(x - x_1 - h)}{2h^2}.(f_3 - 2f_2 + f1) \tag{49}$$

That has a minimum at:

$$x = x_2 + \frac{h.(f_1 - f_3)}{2.(f_3 - 2f_2 + f_1)} \tag{50}$$

Quadratic Interpolation Implementation

```
def interpolation(f, lower, upper):
    a = lower
    b = upper
    h = (1 - g_ratio) * (b-a) # value of h can change, doesn't require to be golden ratio
    c = a + h
    d = b - h
    fc = f(c)
    fd = f(d)
    if fc < fd: # discard b
        # x1 = a, x2 = c, x3 = d
        fa = f(a)

        return c + (h * (fa - fd))/(2 * (fd - 2*fc + fa))

    else: # discard a
        # x1 = c, x2 = d, x3 = b
        fb = f(b)

        return d + (h * (fc - fb))/(2 * (fb - 2*fd + fc))
```

5.2 Multidimensional Optimization

5.2.1 Gradient Method

Let $f(x_1, x_2, ..., x_n) = constant$, then the gradient of the function will be perpendicular to the surface and its direction is the direction of the maximum point of the surface, if we then follow the opposite direction the gradient points to then we can reach the minimum of the function.
The gradient of a function is defined as:

$$\nabla f(\overline{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n}\right) \tag{51}$$

And as the function is a level surface, the sum of all the components of the gradient will equal 0 and the vector gradient points to the maximum of the function.
Using this information we can develop a method that consists on stepping on the direction of the gradient:

$$x_i^{(k+1)} = x_i^{(k)} - h.\nabla_i^{(k)} \tag{52}$$

- If $f(\overline{x}^{(k+1)}) < f(\overline{x}^{(k)})$ the step was successful, and so we can increase the step to $h^{(k+1)} = 2 * h^{(k)}$, or leave it the same.
- If $f(\overline{x}^{(k+1)}) > f(\overline{x}^{(k)})$ the step was unsuccessful, and we discard the value of $\overline{x}^{(k+1)}$ and repeat now with smaller step $h^{(k+1)} = \frac{h^{(k)}}{2}$.

5.2.2 Stopping Criteria

As every iterative method, there must be a criteria to stop the iteration process. With this we get the most common stopping criteria:

Absolute precision criteria (interval length)

$$|\overline{x}^{(k+1)} - \overline{x}^{(k)}| \leqslant \varepsilon \tag{53}$$

The criteria to stop the method is the difference (norm of the difference vector) between the two consecutive values for $\overline{x}$ is lesser than a given value that determines the absolute precision, ε.

Function cancellation criteria

$$|f(\overline{x}^{(k+1)}) - f(\overline{x}^{(k)})| \leqslant \varepsilon \tag{54}$$

The criteria to stop the method is difference between the function values of the two consecutive approximations for the minimum being lesser than a given value, ε. This will indicate the gradient is near zero and will lead to small steps.

Gradient Cancellation Criteria

$$\nabla f(\overline{x}) \approx 0 \tag{55}$$

The criteria to stop the method is the value of the gradient being approximately 0. This means the step taken in the direction of the minimum is small and the method in this situation isn't effective. The best thing to do in this situation is to exit this method and either apply other method (like the Quadric Method to adjust) or just end the process.

These criteria can be used alone or in combination.
For example, we can exit the process when all three check out, or use the two first ones to check for the end of the method and the last one (gradient cancellation) to decide if the result needs adjustment before the next iteration.

Gradient Method Implementation

Let f be the function to two variables x and y, and dfx and dfy the x and y components of the gradient.
Let also x and y be the first guess for the coordinates of the minimum and h is the step.

```
def diff(x0, y0, x1, y1):
    return sqrt((x0-x1)**2 + (y0-y1)**2)

def residue(f, x0, y0, x1, y1):
    return abs(f(x0, y0) - f(x1, y1))

def vector_norm(v0, v1):
    return sqrt(v0**2 + v1**2)

def float_equal(x0, x1, tolerance):
    return abs(x0-x1) <= tolerance

def stop_criteria(f, x0, y0, x1, y1, tolerance):
    # using difference between sucessive points and sucessive values of the minimum
    return float_equal(diff(x0, y0, x1, y1), 0, tolerance) and float_equal(residue(f, x0, y0, x1, y1),

def gradient(f, dfx, dfy, x, y, h, tolerance):

    xt = None # auxiliar variable
    yt = None # auxiliar variable

    success = False # successful step

    gradx = None # value of dfx(x, y)
    grady = None # value of dfy(x, y)

    while (not success or not stop_criteria(f, x, y, xt, yt, tolerance)):

        if success or gradx is None or grady is None:
                # step with success needs gradient values to be recalculated
                # same applies for first iteration
            gradx = dfx(x, y)
            grady = dfy(x, y)

        xt = x - h * gradx
        yt = y - h * grady

        success = True

        if (f(x, y) <= f(xt, yt)): # step without success, discard values
            success = False
            h /= 2 # reduce step

        else: # step with success
            x, xt = xt, x # switch values as x will store the current value,
                          # and xt the old value for comparison in stopping criteria
            y, yt = yt, y # switch values as y will store the current value,
                          # and yt the old value for comparison in stopping criteria

            h *= 2  # increase step (optional)

    return x, y # coordinates for the minimum obtained
```

A problem of this method occurs when near the minimum the value of the gradient is so small that the step taken is almost insignificant and can lead to a very slow convergence.
To solve this, we have adjusting methods like the Quadric Method, that aren't very useful when used alone, but it's a great tool to improve the gradient method.

Quadric Method (Adjusting Method)

Let $f(\mathbf{x})$ be the function to minimize, $\nabla f(\mathbf{x})$ its gradient and $H(\mathbf{x})$ its Hessian matrix, then:

$$g(\mathbf{x}) = f(\mathbf{x}_n) + (\mathbf{x} - \mathbf{x}_n).\nabla f(\mathbf{x}_n) + (\mathbf{x} - \mathbf{x}_n)^2.H(\mathbf{x}_n) \tag{56}$$

Is a good approximation for $f(\mathbf{x})$ near the minimum.

$$\nabla g(\mathbf{x}) = \nabla f(\mathbf{x}_n) + (\mathbf{x} - \mathbf{x}_n).H(\mathbf{x}_n) = 0 \tag{57}$$

From which we obtain the value for $\Delta\mathbf{x}_n$:

$$\Delta\mathbf{x}_n = x - x_n = -H^{-1}(\mathbf{x}_n).\nabla f(\mathbf{x}_n) \tag{58}$$

Gradient + Quadric Adjust Method Implementation

Problem: Find the minimum for the function $f(x, y) = 4x^2 + 3y^2 - 4xy + x + 10$ with initial guess $x = 5$ and $y = 5$ with step of $h = 0.8$.

Resolution:

First let's start by gathering expressions we'll need, such as the expressions for the gradient and as well the Hessian matrix.

Using Maxima:

```
(%i1)f:3*y^2-4*x*y+4*x^2+x+10;
```

(%o1) $$3\,y^2 - 4\,x\,y + 4\,x^2 + x + 10$$

```
(%i2)dfx:diff(f,x);
```

(%o2) $$-4\,y + 8\,x + 1$$

```
(%i3)dfy:diff(f,y);
```

(%o3) $$6\,y - 4\,x$$

```
(%i4)H:hessian(f,[x,y]);
```

(%o4) $$\begin{pmatrix} 8 & -4 \\ -4 & 6 \end{pmatrix}$$

```
(%i5)h:-invert(H) . [dfx,dfy];
```

(%o5) $$\begin{pmatrix} -\frac{6\,y-4\,x}{8} - \frac{3\,(-4\,y+8\,x+1)}{16} \\ -\frac{6\,y-4\,x}{4} - \frac{-4\,y+8\,x+1}{8} \end{pmatrix}$$

```
(%i6)ratsimp(%);
```

(%o6) $$\begin{pmatrix} -\frac{16\,x+3}{16} \\ -\frac{8\,y+1}{8} \end{pmatrix}$$

With this we obtained the expressions needed to implement the method:

```
from math import sqrt

def diff(x0, y0, x1, y1):
    return sqrt((x0-x1)**2 + (y0-y1)**2)

def residue(f, x0, y0, x1, y1):
    return abs(f(x0, y0) - f(x1, y1))

def vector_norm(v0, v1):
    return sqrt(v0**2 + v1**2)
```

```
def float_equal(x0, x1, tolerance):
    return abs(x0-x1) <= tolerance

def stop_criteria(f, x0, y0, x1, y1, tolerance):
    # using difference between sucessive points and sucessive values of the minimum
    return float_equal(diff(x0, y0, x1, y1), 0, tolerance) and float_equal(residue(f, x0, y0, x1, y1),

def gradient(f, dfx, dfy, x, y, h, tolerance):

    xt = None # auxiliar variable
    yt = None # auxiliar variable

    success = False # successfuç step

    gradx = None # value of dfx(x, y)
    grady = None # value of dfy(x, y)

    while (not success or not stop_criteria(f, x, y, xt, yt, tolerance)):

        if success or gradx is None or grady is None: # step with success needs gradient values to be r
            gradx = dfx(x, y)
            grady = dfy(x, y)

        xt = x - h * gradx
        yt = y - h * grady

        success = True

        if (float_equal(vector_norm(gradx, grady), 0, tolerance)):
            print("Gradient Norm: ", vector_norm(gradx, grady))
            #adjust
            #-H^-1.∇ = [-(16*x+3)/16, -(8*y+1)/8]
            print("Adjusting")
            xt = x - (16*x+3)/16
            yt = y - (8*y+1)/8

        if (f(x, y) <= f(xt, yt)): # step without success, discard values
            success = False
            h /= 2 # reduce step

        else: # step with success
            x, xt = xt, x # switch values as x will store the current value, and xt the old value for c
            y, yt = yt, y # switch values as y will store the current value, and yt the old value for c

            h *= 2  # increase step (optional)

    return x, y # coordinates for the minimum obtained

f = lambda x, y: 4*x**2 + 3*y**2 - 4*x*y + x + 10

dfx = lambda x, y: -4*y+8*x+1
dfy = lambda x, y: 6*y-4*x

x, y, h = 5, 5, 0.8
x, y = gradient(f, dfx, dfy, x, y, h, 0.00001)
print(f"Coordinates of minimum: X = {x} |  Y = {y}")
print("Minimum value: ", f(x, y))
```

Program Output:
Coordinates of minimum: X = -0.18750300043746476 — Y = -0.12500507506737413
Minimum value: 9.90625000005237

5.3 Levenberg-Marquardt Method

This method consists on the combination of the two previous methods (Gradient and Quadric) in the same step.

$$\mathbf{h} = \mathbf{h}_{quad} + \lambda \mathbf{h}_{grad} \tag{59}$$

In which $\mathbf{h}_{grad}$ and $\mathbf{h}_{quad}$ are the steps of the Gradient method and Quadric method.
The value of λ is an arbitrary value (value doesn't matter, but initially should be high compared to $\mathbf{h}_{quad}$, as in most cases we're still far from the minimum).
Now in each iteration we just need to decide on whether to increase λ or decrease.

- If the step is successful, this is if $f(\overline{x}^{(k+1)}) < f(\overline{x}^{(k)})$, then we are closer to the minimum and so the importance of the quadric method is higher, as near the minimum the gradient method isn't good. So in a successful step the value of λ decreases.
- If the step is unsuccessful, this is if $f(\overline{x}^{(k+1)}) > f(\overline{x}^{(k)})$, then we are getting farther away from the minimum, and so the importance factor of the gradient step should increase, this is the value of λ should increase. The values obtained for $\overline{x}^{(k+1)}$ on this step are discarded.

Even if this method combines two methods into a single step, there's still chances of the method diverging. The initial value of λ can determine if the method will diverge or not.
For example, if the initial value for λ is too high, the method can overstep in the direction of the gradient and possibly get new approximation whose function value $f(x_0, x_1, ..., x_n)$ is greater than the previous approximation, and thus causing a failed step. According to the method, when a step is failed we must increase the value of λ, and doing will cause an even greater overstep in the next iteration, and the method diverges.

Levenberg-Marquardt Method Implementation

Solving the same problem presented earlier.
Problem: Find the minimum for the function $f(x, y) = 4x^2 + 3y^2 - 4xy + x + 10$ with initial guess $x = 5$ and $y = 5$ with initial gradient weight of $\lambda = 0.8$.
Resolution:
First let's start by gathering expressions we'll need, such as the expressions for the gradient and as well the Hessian matrix.
Using Maxima:

```
(%i1)f:3*y^2-4*x*y+4*x^2+x+10;
```

(%o1) $$3\,y^2 - 4\,x\,y + 4\,x^2 + x + 10$$

```
(%i2)dfx:diff(f,x);
```

(%o2) $$-4\,y + 8\,x + 1$$

```
(%i3)dfy:diff(f,y);
```

(%o3) $$6\,y - 4\,x$$

```
(%i4)H:hessian(f,[x,y]);
```

(%o4) $$\begin{pmatrix} 8 & -4 \\ -4 & 6 \end{pmatrix}$$

```
(%i5)h:-invert(H) . [dfx,dfy];
```

(%o5) $$\begin{pmatrix} -\frac{6\,y-4\,x}{8} - \frac{3\,(-4\,y+8\,x+1)}{16} \\ -\frac{6\,y-4\,x}{4} - \frac{-4\,y+8\,x+1}{8} \end{pmatrix}$$

```
(%i6)ratsimp(%);
```

(%o6) $$\begin{pmatrix} -\frac{16\,x+3}{16} \\ -\frac{8\,y+1}{8} \end{pmatrix}$$

With this we obtained the expressions needed to implement the method:
Being the increment of the method defined as

$$h_{LM} = h_{quad} + \lambda \times h_{grad} = -H^{-1}.\nabla - \lambda.\nabla \tag{60}$$

```
from math import sqrt

def diff(x0, y0, x1, y1):
    return sqrt((x0-x1)**2 + (y0-y1)**2)

def residue(f, x0, y0, x1, y1):
    return abs(f(x0, y0) - f(x1, y1))

def vector_norm(v0, v1):
    return sqrt(v0**2 + v1**2)

def float_equal(x0, x1, tolerance):
    return abs(x0-x1) <= tolerance

def stop_criteria(f, x0, y0, x1, y1, tolerance):
    # using difference between sucessive points and sucessive values of the minimum
    return float_equal(diff(x0, y0, x1, y1), 0, tolerance) and float_equal(residue(f, x0, y0, x1, y1),

def levenberg_m(f, dfx, dfy, qx, qy, x, y, h, tolerance):

    xt = None # auxiliar variable
    yt = None # auxiliar variable

    success = False # successful step

    gradx = None # value of dfx(x, y)
    grady = None # value of dfy(x, y)
    quadx = None # value of £-H^{-1}.\nabla_x£
    quady = None # value of £-H^{-1}.\nabla_y£

    while (not success or not stop_criteria(f, x, y, xt, yt, tolerance)):

        if success or gradx is None or grady is None or quadx is None or quady is None:
            # step with success, values need to be recalculated or first iteration
            gradx = dfx(x, y)
            grady = dfy(x, y)
            quadx = qx(x, y)
            quady = qy(x, y)

        hx = quadx - h * gradx
        hy = quady - h * grady
        xt = x + hx
        yt = y + hy

        success = True

        if (f(x, y) < f(xt, yt)):
            #further away, gradient importance raises
            print("Failure")
            h *= 2
            success = False

        else:
            #closer to min, gradient importance decreases
            print("Success")
            x, xt = xt, x
            y, yt = yt, y
            h /= 2

    return x, y # coordinates for the minimum obtained
```

```
f = lambda x, y: 4*x**2 + 3*y**2 - 4*x*y + x + 10

dfx = lambda x, y: -4*y+8*x+1
dfy = lambda x, y: 6*y-4*x

qx = lambda x, y: -(16*x+3)/16
qy = lambda x, y: -(8*y+1)/8

x, y, h = 5, 5, 0.2
x, y = levenberg_m(f, dfx, dfy, qx, qy, x, y, h, 0.001)
print(f"Coordinates of minimum: X = {x} |  Y = {y}")
print("Minimum value: ", f(x, y))
```

Program Output:
Coordinates of minimum: X = -0.18749835323242187 — Y = -0.12500128534667967
Minimum value: 9.90625000002427