

Chem 277B Spring 2024 Tutorial 1

Outline

- Visualizing univariate and multivariate functions
 - Time your function
 - Golden section: 1D function
 - Minimization using CG/BFGS: 2D function
-

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
```

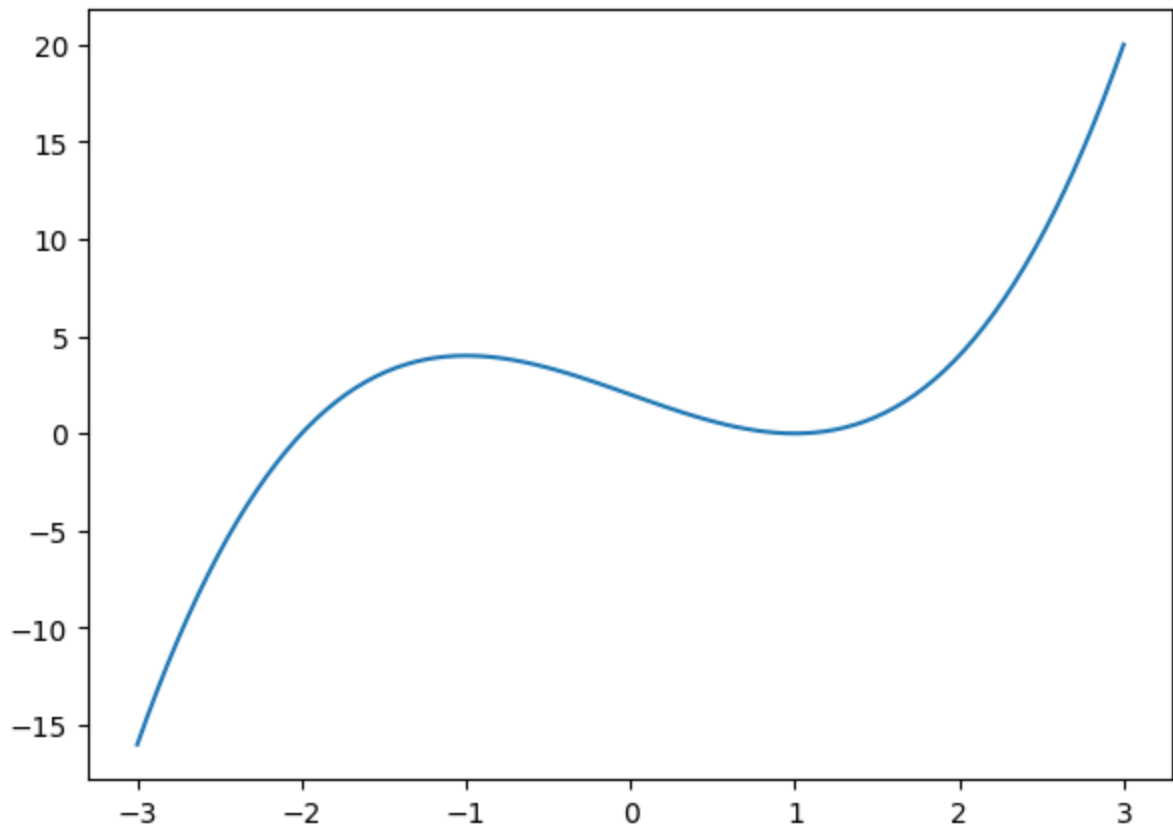
1. Visualizing univariate and multivariate functions

1.1 Univariate

$$f(x) = x^3 - 3x + 2$$

```
In [ ]: def func1d(x):
        f = x**3 - 3*x + 2
        return f

x = np.linspace(-3,3,100)
plt.figure(figsize = (7,5))
plt.plot(x, func1d(x))
plt.show()
```



1.2 Multivariate

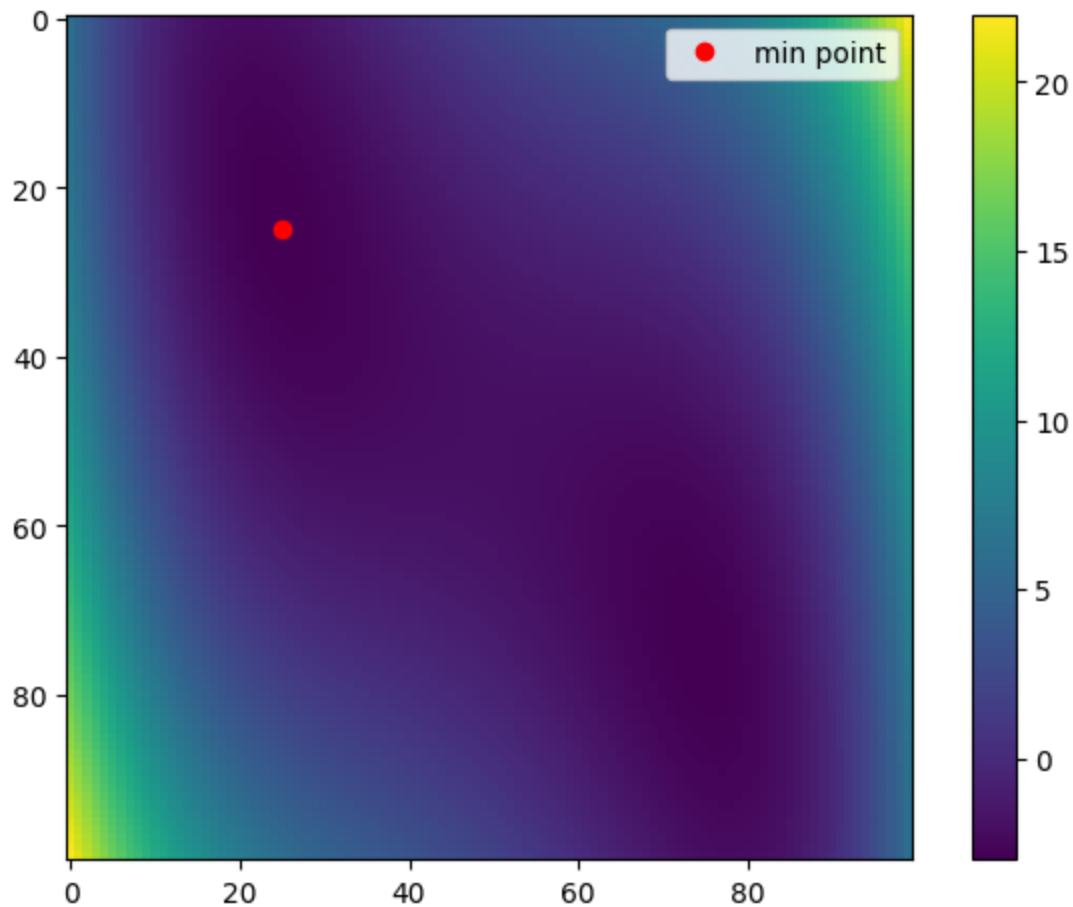
```
In [ ]: def func2d(X):
        """
        A 2D function
        """
        x,y = X
        return x**4-x**2+y**2-2*x*y-2

        # Create a meshgrid
        x = np.linspace(-2,2,100)
        y = np.linspace(-2,2,100)
        X, Y = np.meshgrid(x, y)
        Z = func2d([X, Y])

        # Mark the minimum point on this grid (not the true minimum)
        i, j = np.unravel_index(Z.argmin(),Z.shape)
        min_point = x[i], y[j], Z.min()
        print(min_point)
```

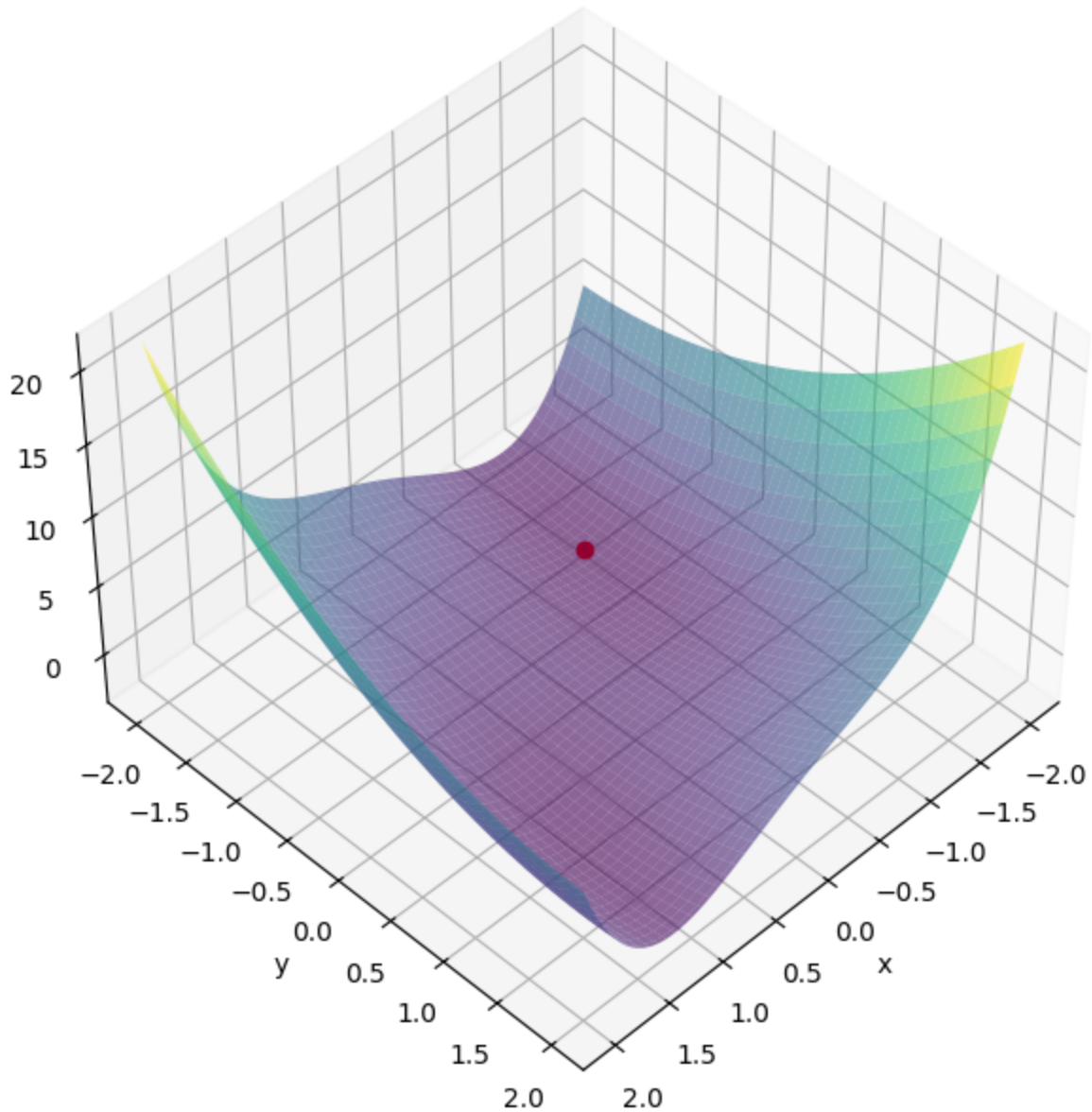
(-0.9898989898989898, -0.9898989898989898, -2.999595990410162)

```
In [ ]: # Heatmap
plt.imshow(func2d([X,Y]))
plt.plot(i,j,'ro',label='min point')
plt.colorbar()
plt.legend()
plt.tight_layout()
```



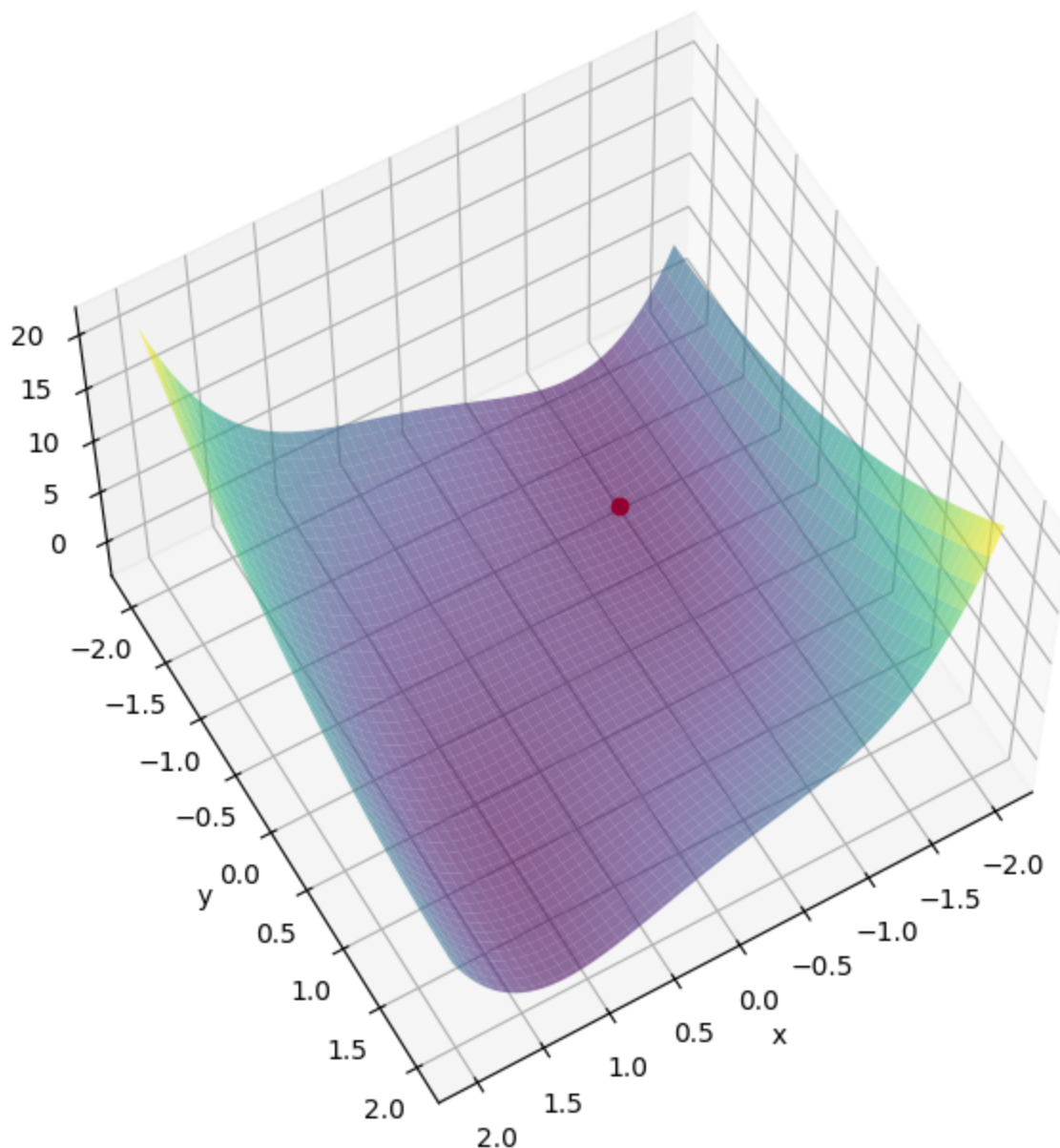
```
In [ ]: # Plot surface + heatmap

# Plotting
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X,Y,Z, cmap='viridis', alpha=0.6)
ax.plot(*min_point,color='r',marker='o')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.view_init(45,45) # set angles for viewing
plt.tight_layout()
plt.show()
```



TODO: Plot the same surface but without the heat map, at a different viewing angle

```
In [ ]: # Your code here
# Plotting
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X,Y,Z, cmap='viridis', alpha=0.6)
ax.plot(*min_point,color='r',marker='o')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.view_init(elev=60,azim=60) # set angles for viewing
plt.tight_layout()
plt.show()
```



2. Time your function

2.1 With a magic method

```
In [ ]: def add(a, b):  
        return a + b  
  
        %timeit add(1, 2)
```

54.1 ns \pm 0.152 ns per loop (mean \pm std. dev. of 7 runs, 10,000,000 loops each)

2.2 With decorators

Decorators wrap a function, and modify its behavior.

```
In [ ]: import time

def timeit(f):

    def timed(*args, **kw):

        ts = time.time()
        result = f(*args, **kw)
        te = time.time()

        print(f'func:{f.__name__} took: {te-ts:.4f} sec')
        return result

    return timed
```

```
In [ ]: @timeit # equivalent to mysleep = timeit(mysleep)
def mysleep(sec):
    print("Sleeping...")
    time.sleep(sec)
    print("Wake up!")

mysleep(2)
```

Sleeping...

Wake up!

func:mysleep took: 2.0029 sec

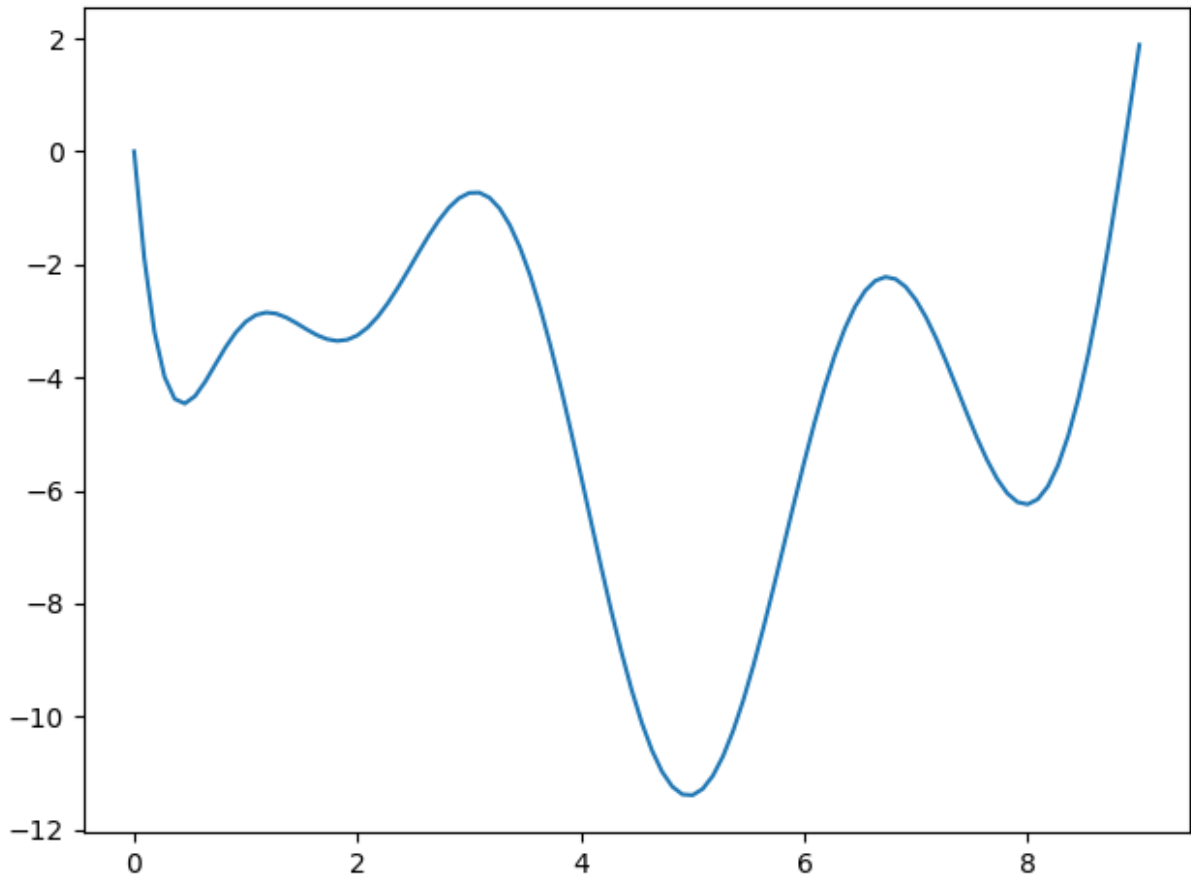
3. Golden section: 1D function

Given the following function, can you find its minimum value in range $[0, 9]$ using golden section?

```
In [ ]: def func(x):
    isarray = type(x) is np.ndarray
    coefs = np.array([
        8.0013714770, -24.06731415, 37.07604400, 0.0000000000, -43.86909846,
        44.427011010, -22.01262040, 6.536434989, -1.248082478, 0.157159012,
        -0.012990941, 0.000678657, -2.03269E-05, 2.66065E-07
    ])
    base = np.zeros((len(x) if isarray else 1, 14))
    for i in range(1, 14):
        base[:,i] += x ** i
    result = base.dot(coefs)
    return result if isarray else np.sum(result)
```

```
In [ ]: # Let's plot and see what this function looks like
x = np.linspace(0, 9, 100)
```

```
plt.plot(x, func(x))
plt.tight_layout()
```



TODO: Implement golden section assuming we only need to run it once

```
In [ ]: # Fill out this code

# Define the golden ratio
tau = (1 + np.sqrt(5)) / 2

def golden_section_once(f, a, b):
    """
    Find the next search region in the interval [a, d] using the golden section search.

    Params:
        f: function to optimize
        a: left limit of the region to optimize
        b: right limit of the region to optimize

    Return: next region to search

    """
    # define the two points to search within the region based on golden section search
    # HINT: the points are ordered as a-c-d-b, find points c & d
    c = b - (b - a) / tau
    d = a + (b - a) / tau
```

```

# compare their values and identify the next search space
# HINT: return f'next search space is from {:.3f} to {:.3f}, reduced by
if f(c) < f(d):
    return f'next search space is from {a:.3f} to {d}, reduced by {(d-a)/
else:
    return f'next search space is from {c:.3f} to {b:.3f}, reduced by {(b-c)/

```

```

In [ ]: print(f'1/tau = {1/tau}')
        print(golden_section_once(func, 0, 9))

```

1/tau = 0.6180339887498948

next search space is from 3.438 to 9.000, reduced by 0.618

TODO: Use recursion to feed the new search space into the same function

```

In [ ]: # Actually the return statement can be a function call, ie "return golden_section"
        # However, this recursion can go on forever, and we will be stuck in an infinite loop
        # This means that we need to define a condition for the recursion to stop!

```

```

# Let's now revisit our function
def golden_section(f, a, b, tol=1e-5, i=1):
    """
    Find the minimum of f in the interval [a, b] using the golden section search method.

    Params
    f: function to optimize
    a: left limit of the region to optimize
    b: right limit of the region to optimize
    tol: tolerance for convergence
    i: # of searching rounds

    Return: the minimum point, and its functional value
    """

    c = b - (b - a) / tau
    d = a + (b - a) / tau

    # define the stopping condition here
    if abs(b-a) <= tol:
        return (b+a) / 2, func((b+a)/2) # return the midpoint as solution

    # copy the function from above and change the output to a functional call
    if f(c) < f(d):
        print(f'next search space is from {a:.3f} to {d}, reduced by {(d-a)/tau}')
        return golden_section(f, a, d, tol, i+1)
    else:
        print(f'next search space is from {c:.3f} to {b:.3f}, reduced by {(b-c)/tau}')
        return golden_section(f, c, b, tol, i+1)

```

```

In [ ]: golden_section(func, 0, 9)

```



```
next search space is from 3.438 to 9.000, reduced by 0.618
next search space is from 3.438 to 6.875388202501893, reduced by 0.618
next search space is from 3.438 to 5.562305898749054, reduced by 0.618
next search space is from 4.249 to 5.562, reduced by 0.618
next search space is from 4.751 to 5.562, reduced by 0.618
next search space is from 4.751 to 5.252329215011357, reduced by 0.618
next search space is from 4.751 to 5.0607530887414836, reduced by 0.618
next search space is from 4.869 to 5.061, reduced by 0.618
next search space is from 4.869 to 4.987577519939434, reduced by 0.618
next search space is from 4.914 to 4.988, reduced by 0.618
next search space is from 4.942 to 4.988, reduced by 0.618
next search space is from 4.942 to 4.970303111409937, reduced by 0.618
next search space is from 4.953 to 4.970, reduced by 0.618
next search space is from 4.960 to 4.970, reduced by 0.618
next search space is from 4.960 to 4.966225176725874, reduced by 0.618
next search space is from 4.962 to 4.966, reduced by 0.618
next search space is from 4.964 to 4.966, reduced by 0.618
next search space is from 4.964 to 4.96526250693263, reduced by 0.618
next search space is from 4.964 to 4.964667544280463, reduced by 0.618
next search space is from 4.964 to 4.965, reduced by 0.618
next search space is from 4.964 to 4.964440288769372, reduced by 0.618
next search space is from 4.964 to 4.964299837139387, reduced by 0.618
next search space is from 4.964 to 4.964, reduced by 0.618
next search space is from 4.964 to 4.964, reduced by 0.618
next search space is from 4.964 to 4.96426668100716, reduced by 0.618
next search space is from 4.964 to 4.964, reduced by 0.618
next search space is from 4.964 to 4.964, reduced by 0.618
next search space is from 4.964 to 4.964258853906084, reduced by 0.618
next search space is from 4.964 to 4.964, reduced by 0.618
```

```
Out[ ]: (4.964254940355545, -11.400276695047907)
```

4. Minimization using CG/BFGS: 2D function

```
In [ ]: from scipy.optimize import minimize
        ?minimize
```

Signature:

```

minimize(
    fun,
    x0,
    args=(),
    method=None,
    jac=None,
    hess=None,
    hessp=None,
    bounds=None,
    constraints=(),
    tol=None,
    callback=None,
    options=None,
)

```

Docstring:

Minimization of scalar function of one or more variables.

Parameters

fun : callable

The objective function to be minimized.

```fun(x, *args) -> float```

where ```x``` is a 1-D array with shape `(n,)` and ```args``` is a tuple of the fixed parameters needed to completely specify the function.

**x0** : ndarray, shape `(n,)`

Initial guess. Array of real elements of size `(n,)`, where ```n``` is the number of independent variables.

**args** : tuple, optional

Extra arguments passed to the objective function and its derivatives (```fun```, ```jac``` and ```hess``` functions).

**method** : str or callable, optional

Type of solver. Should be one of

- 'Nelder-Mead' :ref:`(see here) <optimize.minimize-neldermead>`
- 'Powell' :ref:`(see here) <optimize.minimize-powell>`
- 'CG' :ref:`(see here) <optimize.minimize-cg>`
- 'BFGS' :ref:`(see here) <optimize.minimize-bfgs>`
- 'Newton-CG' :ref:`(see here) <optimize.minimize-newtoncg>`
- 'L-BFGS-B' :ref:`(see here) <optimize.minimize-lbfgsb>`
- 'TNC' :ref:`(see here) <optimize.minimize-tnc>`
- 'COBYLA' :ref:`(see here) <optimize.minimize-cobyla>`
- 'SLSQP' :ref:`(see here) <optimize.minimize-slsqp>`
- 'trust-constr' :ref:`(see here) <optimize.minimize-trustconstr>`
- 'dogleg' :ref:`(see here) <optimize.minimize-dogleg>`
- 'trust-ncg' :ref:`(see here) <optimize.minimize-trustncg>`
- 'trust-exact' :ref:`(see here) <optimize.minimize-trustexact>`
- 'trust-krylov' :ref:`(see here) <optimize.minimize-trustkrylov>`
- custom - a callable object, see below for description.

If not given, chosen to be one of ```BFGS```, ```L-BFGS-B```, ```SLSQP```, depending on whether or not the problem has constraints or bounds.

**jac** : {callable, '2-point', '3-point', 'cs', bool}, optional

Method for computing the gradient vector. Only for CG, BFGS, Newton-CG, L-BFGS-B, TNC, SLSQP, dogleg, trust-ncg, trust-krylov, trust-exact and trust-constr.  
If it is a callable, it should be a function that returns the gradient vector:

```
``jac(x, *args) -> array_like, shape (n,)``
```

where ``x`` is an array with shape (n,) and ``args`` is a tuple with the fixed parameters. If ``jac`` is a Boolean and is True, ``fun`` is assumed to return a tuple ``(f, g)`` containing the objective function and the gradient.

Methods 'Newton-CG', 'trust-ncg', 'dogleg', 'trust-exact', and 'trust-krylov' require that either a callable be supplied, or that ``fun`` return the objective and gradient.

If None or False, the gradient will be estimated using 2-point finite difference estimation with an absolute step size.

Alternatively, the keywords {'2-point', '3-point', 'cs'} can be used to select a finite difference scheme for numerical estimation of the gradient with a relative step size. These finite difference schemes obey any specified ``bounds``.

hess : {callable, '2-point', '3-point', 'cs', HessianUpdateStrategy}, optional

Method for computing the Hessian matrix. Only for Newton-CG, dogleg, trust-ncg, trust-krylov, trust-exact and trust-constr.  
If it is callable, it should return the Hessian matrix:

```
``hess(x, *args) -> {LinearOperator, spmatrix, array}, (n, n)``
```

where ``x`` is a (n,) ndarray and ``args`` is a tuple with the fixed parameters.

The keywords {'2-point', '3-point', 'cs'} can also be used to select a finite difference scheme for numerical estimation of the hessian.

Alternatively, objects implementing the ``HessianUpdateStrategy`` interface can be used to approximate the Hessian. Available quasi-Newton methods implementing this interface are:

- ``BFGS``;
- ``SR1``.

Not all of the options are available for each of the methods; for availability refer to the notes.

hessp : callable, optional

Hessian of objective function times an arbitrary vector p. Only for Newton-CG, trust-ncg, trust-krylov, trust-constr.

Only one of ``hessp`` or ``hess`` needs to be given. If ``hess`` is provided, then ``hessp`` will be ignored. ``hessp`` must compute the Hessian times an arbitrary vector:

```
``hessp(x, p, *args) -> ndarray shape (n,)``
```

where ``x`` is a (n,) ndarray, ``p`` is an arbitrary vector with dimension (n,) and ``args`` is a tuple with the fixed parameters.

bounds : sequence or ``Bounds``, optional

Bounds on variables for Nelder-Mead, L-BFGS-B, TNC, SLSQP, Powell,

trust-constr, and COBYLA methods. There are two ways to specify the bounds:

1. Instance of ``Bounds`` class.
2. Sequence of ```(min, max)``` pairs for each element in ``x``. None is used to specify no bound.

constraints : {Constraint, dict} or List of {Constraint, dict}, optional  
Constraints definition. Only for COBYLA, SLSQP and trust-constr.

Constraints for 'trust-constr' are defined as a single object or a list of objects specifying constraints to the optimization problem. Available constraints are:

- ``LinearConstraint``
- ``NonlinearConstraint``

Constraints for COBYLA, SLSQP are defined as a list of dictionaries. Each dictionary with fields:

```
type : str
 Constraint type: 'eq' for equality, 'ineq' for inequality.
fun : callable
 The function defining the constraint.
jac : callable, optional
 The Jacobian of `fun` (only for SLSQP).
args : sequence, optional
 Extra arguments to be passed to the function and Jacobian.
```

Equality constraint means that the constraint function result is to be zero whereas inequality means that it is to be non-negative.

Note that COBYLA only supports inequality constraints.

```
tol : float, optional
 Tolerance for termination. When `tol` is specified, the selected minimization algorithm sets some relevant solver-specific tolerance(s) equal to `tol`. For detailed control, use solver-specific options.
```

```
options : dict, optional
 A dictionary of solver options. All methods except `TNC` accept the following generic options:
```

```
maxiter : int
 Maximum number of iterations to perform. Depending on the method each iteration may use several function evaluations.

 For `TNC` use `maxfun` instead of `maxiter`.
disp : bool
 Set to True to print convergence messages.
```

For method-specific options, see `:func:`show_options()``.

```
callback : callable, optional
 A callable called after each iteration.
```

All methods except TNC, SLSQP, and COBYLA support a callable with the signature:

```
``callback(intermediate_result: OptimizeResult)``
```

where ``intermediate\_result`` is a keyword parameter containing an `OptimizeResult` with attributes ``x`` and ``fun``, the present values of the parameter vector and objective function. Note that the name of the parameter must be ``intermediate\_result`` for the callback to be passed an `OptimizeResult`. These methods will also terminate if the callback raises ``StopIteration``.

All methods except trust-constr (also) support a signature like:

```
``callback(xk)``
```

where ``xk`` is the current parameter vector.

Introspection is used to determine which of the signatures above to invoke.

## Returns

```
res : OptimizeResult
```

The optimization result represented as a `OptimizeResult` object. Important attributes are: ``x`` the solution array, ``success`` a Boolean flag indicating if the optimizer exited successfully and ``message`` which describes the cause of the termination. See `OptimizeResult` for a description of other attributes.

## See also

```
minimize_scalar : Interface to minimization algorithms for scalar
 univariate functions
show_options : Additional options accepted by the solvers
```

## Notes

This section describes the available solvers that can be selected by the 'method' parameter. The default method is \*BFGS\*.

### \*\*Unconstrained minimization\*\*

Method :ref:`CG <optimize.minimize-cg>` uses a nonlinear conjugate gradient algorithm by Polak and Ribiere, a variant of the Fletcher-Reeves method described in [5]\_ pp.120–122. Only the first derivatives are used.

Method :ref:`BFGS <optimize.minimize-bfgs>` uses the quasi-Newton method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) [5]\_ pp. 136. It uses the first derivatives only. BFGS has proven good performance even for non-smooth optimizations. This method also returns an approximation of the Hessian inverse, stored as `hess\_inv` in the OptimizeResult object.

Method :ref:`Newton-CG <optimize.minimize-newtoncg>` uses a Newton-CG algorithm [5]\_ pp. 168 (also known as the truncated Newton method). It uses a CG method to compute the search direction. See also \*TNC\* method for a box-constrained

minimization with a similar algorithm. Suitable for large-scale problems.

Method :ref:`dogleg <optimize.minimize-dogleg>` uses the dog-leg trust-region algorithm [5]\_ for unconstrained minimization. This algorithm requires the gradient and Hessian; furthermore the Hessian is required to be positive definite.

Method :ref:`trust-ncg <optimize.minimize-trustncg>` uses the Newton conjugate gradient trust-region algorithm [5]\_ for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems.

Method :ref:`trust-krylov <optimize.minimize-trustkrylov>` uses the Newton GLTR trust-region algorithm [14]\_, [15]\_ for unconstrained minimization. This algorithm requires the gradient and either the Hessian or a function that computes the product of the Hessian with a given vector. Suitable for large-scale problems. On indefinite problems it requires usually less iterations than the `trust-ncg` method and is recommended for medium and large-scale problems.

Method :ref:`trust-exact <optimize.minimize-trustexact>` is a trust-region method for unconstrained minimization in which quadratic subproblems are solved almost exactly [13]\_. This algorithm requires the gradient and the Hessian (which is *\*not\** required to be positive definite). It is, in many situations, the Newton method to converge in fewer iterations and the most recommended for small and medium-size problems.

#### **\*\*Bound-Constrained minimization\*\***

Method :ref:`Nelder-Mead <optimize.minimize-neldermead>` uses the Simplex algorithm [1]\_, [2]\_. This algorithm is robust in many applications. However, if numerical computation of derivative can be trusted, other algorithms using the first and/or second derivatives information might be preferred for their better performance in general.

Method :ref:`L-BFGS-B <optimize.minimize-lbfgsb>` uses the L-BFGS-B algorithm [6]\_, [7]\_ for bound constrained minimization.

Method :ref:`Powell <optimize.minimize-powell>` is a modification of Powell's method [3]\_, [4]\_ which is a conjugate direction method. It performs sequential one-dimensional minimizations along each vector of the directions set (`direc` field in `options` and `info`), which is updated at each iteration of the main minimization loop. The function need not be differentiable, and no derivatives are taken. If bounds are not provided, then an unbounded line search will be used. If bounds are provided and the initial guess is within the bounds, then every function evaluation throughout the minimization procedure will be within the bounds. If bounds are provided, the initial guess is outside the bounds, and `direc` is full rank (default has full rank), then some function evaluations during the first iteration may be outside the bounds, but every function evaluation after the first

iteration will be within the bounds. If `direc` is not full rank, then some parameters may not be optimized and the solution is not guaranteed to be within the bounds.

Method :ref:`TNC <optimize.minimize-tnc>` uses a truncated Newton algorithm [5]\_, [8]\_ to minimize a function with variables subject to bounds. This algorithm uses gradient information; it is also called Newton Conjugate-Gradient. It differs from the *\*Newton-CG\** method described above as it wraps a C implementation and allows each variable to be given upper and lower bounds.

#### **\*\*Constrained Minimization\*\***

Method :ref:`COBYLA <optimize.minimize-cobyla>` uses the Constrained Optimization BY Linear Approximation (COBYLA) method [9]\_, [10]\_, [11]\_. The algorithm is based on linear approximations to the objective function and each constraint. The method wraps a FORTRAN implementation of the algorithm. The constraints functions 'fun' may return either a single number or an array or list of numbers.

Method :ref:`SLSQP <optimize.minimize-slsqp>` uses Sequential Least Squares Programming to minimize a function of several variables with any combination of bounds, equality and inequality constraints. The method wraps the SLSQP Optimization subroutine originally implemented by Dieter Kraft [12]\_. Note that the wrapper handles infinite values in bounds by converting them into large floating values.

Method :ref:`trust-constr <optimize.minimize-trustconstr>` is a trust-region algorithm for constrained optimization. It switches between two implementations depending on the problem definition. It is the most versatile constrained minimization algorithm implemented in SciPy and the most appropriate for large-scale problems. For equality constrained problems it is an implementation of Byrd-Omojokun Trust-Region SQP method described in [17]\_ and in [5]\_, p. 549. When inequality constraints are imposed as well, it switches to the trust-region interior point method described in [16]\_. This interior point algorithm, in turn, solves inequality constraints by introducing slack variables and solving a sequence of equality-constrained barrier problems for progressively smaller values of the barrier parameter. The previously described equality constrained SQP method is used to solve the subproblems with increasing levels of accuracy as the iterate gets closer to a solution.

#### **\*\*Finite-Difference Options\*\***

For Method :ref:`trust-constr <optimize.minimize-trustconstr>` the gradient and the Hessian may be approximated using three finite-difference schemes: {'2-point', '3-point', 'cs'}. The scheme 'cs' is, potentially, the most accurate but it requires the function to correctly handle complex inputs and to be differentiable in the complex plane. The scheme '3-point' is more accurate than '2-point' but requires twice as many operations. If the gradient is estimated via finite-differences the Hessian must be estimated using one of the quasi-Newton strategies.

**\*\*Method specific options for the\*\* `hess` **\*\*keyword\*\*****

method/Hess	None	callable	'2-point'/'3-point'/'cs'	HUS
Newton-CG	x	(n, n) L0	x	x
dogleg		(n, n)		
trust-ncg		(n, n)	x	x
trust-krylov		(n, n)	x	x
trust-exact		(n, n)		
trust-constr	x	(n, n) L0 sp	x	x

where L0=LinearOperator, sp=Sparse matrix, HUS=HessianUpdateStrategy

**\*\*Custom minimizers\*\***

It may be useful to pass a custom minimization method, for example when using a frontend to this method such as `scipy.optimize.basinhopping` or a different library. You can simply pass a callable as the `method` parameter.

The callable is called as `method(fun, x0, args, **kwargs, **options)` where `kwargs` corresponds to any other parameters passed to `minimize` (such as `callback`, `hess`, etc.), except the `options` dict, which has its contents also passed as `method` parameters pair by pair. Also, if `jac` has been passed as a bool type, `jac` and `fun` are mangled so that `fun` returns just the function values and `jac` is converted to a function returning the Jacobian. The method shall return an `OptimizeResult` object.

The provided `method` callable must be able to accept (and possibly ignore) arbitrary parameters; the set of parameters accepted by `minimize` may expand in future versions and then these parameters will be passed to the method. You can find an example in the `scipy.optimize` tutorial.

## References

- .. [1] Nelder, J A, and R Mead. 1965. A Simplex Method for Function Minimization. The Computer Journal 7: 308-13.
- .. [2] Wright M H. 1996. Direct search methods: Once scorned, now respectable, in Numerical Analysis 1995: Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis (Eds. D F Griffiths and G A Watson). Addison Wesley Longman, Harlow, UK. 191-208.
- .. [3] Powell, M J D. 1964. An efficient method for finding the minimum of a function of several variables without calculating derivatives. The



- Computer Journal 7: 155–162.
- .. [4] Press W, S A Teukolsky, W T Vetterling and B P Flannery. Numerical Recipes (any edition), Cambridge University Press.
  - .. [5] Nocedal, J, and S J Wright. 2006. Numerical Optimization. Springer New York.
  - .. [6] Byrd, R H and P Lu and J. Nocedal. 1995. A Limited Memory Algorithm for Bound Constrained Optimization. SIAM Journal on Scientific and Statistical Computing 16 (5): 1190–1208.
  - .. [7] Zhu, C and R H Byrd and J Nocedal. 1997. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization. ACM Transactions on Mathematical Software 23 (4): 550–560.
  - .. [8] Nash, S G. Newton-Type Minimization Via the Lanczos Method. 1984. SIAM Journal of Numerical Analysis 21: 770–778.
  - .. [9] Powell, M J D. A direct search optimization method that models the objective and constraint functions by linear interpolation. 1994. Advances in Optimization and Numerical Analysis, eds. S. Gomez and J-P Hennart, Kluwer Academic (Dordrecht), 51–67.
  - .. [10] Powell M J D. Direct search algorithms for optimization calculations. 1998. Acta Numerica 7: 287–336.
  - .. [11] Powell M J D. A view of algorithms for optimization without derivatives. 2007. Cambridge University Technical Report DAMTP 2007/NA03
  - .. [12] Kraft, D. A software package for sequential quadratic programming. 1988. Tech. Rep. DFVLR-FB 88–28, DLR German Aerospace Center -- Institute for Flight Mechanics, Koln, Germany.
  - .. [13] Conn, A. R., Gould, N. I., and Toint, P. L. Trust region methods. 2000. Siam. pp. 169–200.
  - .. [14] F. Lenders, C. Kirches, A. Potschka: "trlib: A vector-free implementation of the GLTR method for iterative solution of the trust region problem", :arxiv:`1611.04718`
  - .. [15] N. Gould, S. Lucidi, M. Roma, P. Toint: "Solving the Trust-Region Subproblem using the Lanczos Method", SIAM J. Optim., 9(2), 504–525, (1999).
  - .. [16] Byrd, Richard H., Mary E. Hribar, and Jorge Nocedal. 1999. An interior point algorithm for large-scale nonlinear programming. SIAM Journal on Optimization 9.4: 877–900.
  - .. [17] Lalee, Marucha, Jorge Nocedal, and Todd Plantega. 1998. On the implementation of an algorithm for large-scale equality constrained optimization. SIAM Journal on Optimization 8.3: 682–706.

## Examples

-----

Let us consider the problem of minimizing the Rosenbrock function. This function (and its respective derivatives) is implemented in `rosen` (resp. `rosen\_der`, `rosen\_hess`) in the `scipy.optimize`.

```
>>> from scipy.optimize import minimize, rosen, rosen_der
```

A simple application of the \*Nelder-Mead\* method is:

```
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> res = minimize(rosen, x0, method='Nelder-Mead', tol=1e-6)
>>> res.x
array([1., 1., 1., 1., 1.])
```

Now using the \*BFGS\* algorithm, using the first derivative and a few options:

```
>>> res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
... options={'gtol': 1e-6, 'disp': True})
Optimization terminated successfully.
 Current function value: 0.000000
 Iterations: 26
 Function evaluations: 31
 Gradient evaluations: 31
>>> res.x
array([1., 1., 1., 1., 1.])
>>> print(res.message)
Optimization terminated successfully.
>>> res.hess_inv
array([
 [0.00749589, 0.01255155, 0.02396251, 0.04750988, 0.09495377], # ma
y vary
 [0.01255155, 0.02510441, 0.04794055, 0.09502834, 0.18996269],
 [0.02396251, 0.04794055, 0.09631614, 0.19092151, 0.38165151],
 [0.04750988, 0.09502834, 0.19092151, 0.38341252, 0.7664427],
 [0.09495377, 0.18996269, 0.38165151, 0.7664427, 1.53713523]
])
```

Next, consider a minimization problem with several constraints (namely Example 16.4 from [5]). The objective function is:

```
>>> fun = lambda x: (x[0] - 1)**2 + (x[1] - 2.5)**2
```

There are three constraints defined as:

```
>>> cons = ({'type': 'ineq', 'fun': lambda x: x[0] - 2 * x[1] + 2},
... {'type': 'ineq', 'fun': lambda x: -x[0] - 2 * x[1] + 6},
... {'type': 'ineq', 'fun': lambda x: -x[0] + 2 * x[1] + 2})
```

And variables must be positive, hence the following bounds:

```
>>> bnds = ((0, None), (0, None))
```

The optimization problem is solved using the SLSQP method as:

```
>>> res = minimize(fun, (2, 0), method='SLSQP', bounds=bnds,
... constraints=cons)
```

It should converge to the theoretical solution (1.4 ,1.7).

```
File: ~/miniconda3/envs/chem277b/lib/python3.10/site-packages/scipy/opt
imize/_minimize.py
Type: function
```

```
In []: def save_step(*args):
 for arg in args:
 if type(arg) is np.ndarray:
 steps.append(arg)
```

## TODO: Writer a wrapper function around `scipy.optimize.minimize`

```
In []: @timeit
def minimize_function(x0, func, method):
 """
 Minimize a function

 Parameters

 x0: np.ndarray
 Starting point
 func: function
 Scalar function to minimize
 method: str
 Method for minimization

 Returns

 res: OptimizeResult
 Result object of scipy optimization
 """
 res = minimize(
 func,
 x0,
 method=method,
 options={"gtol": 1e-5, "disp": True},
 callback=save_step,
)
 return res
```

### 4.1 Previous 2D function

$$f(x, y) = x^4 - x^2 + y^2 - 2xy - 2$$

```
In []: def func2d(X):
 """
 A 2D function
 """
 x, y = X
 return x**4 - x**2 + y**2 - 2*x*y - 2
```

```
In []: x0 = [0, -0.1]
steps = [x0] # for storing the searched points
res = minimize_function(x0, func2d, "CG")
steps1 = np.array(steps)
print('Solution:', res.x)
print()
```

```
x0 = [0.4, -0.5]
steps = [x0] # for storing the searched points
res = minimize_function(x0, func2d, "CG")
steps2 = np.array(steps)
print('Solution:', res.x)
print()
```

```
In []: x = np.linspace(-1.5, 1.5, 100)
y = np.linspace(-1.5, 1.5, 100)
X, Y = np.meshgrid(x, y)
Z = func2d([X, Y])

Your code here
fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, alpha=0.6)

ax.plot(steps1[:, 0], steps1[:, 1], [func2d(p) for p in steps1], color='r')
ax.plot(*steps1[-1], func2d(steps1[-1]), color='r', marker='o')
ax.plot(steps2[:, 0], steps2[:, 1], [func2d(p) for p in steps2], color='g')
ax.plot(*steps2[-1], func2d(steps2[-1]), color='g', marker='o')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.view_init(60, 60) # set angles for viewing
plt.tight_layout()
plt.show()
```

## 4.2 Rosenbrock function

$$f(\vec{x}) = \sum_{i=1}^{N-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i^2)^2$$

Use `x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])` as a starting point to find the minimum

### **TODO:** Implement the Rosenbrock function

```
In []: def rosenbrock(X):
 """
 Rosenbrock function

 Parameters

 X: np.ndarray
 Input values

 Returns

 y: float
 Output of Rosenbrock function with given X
```

```

'''

X2 = X ** 2
y = np.sum(100 * (X[1:] - X2[:-1]) ** 2 + (1 - X2[:-1])**2)
return y

def test_rosenbrock():
 testX = np.array([1.0, 2.0])
 if rosenbrock(testX) == 100.0:
 print("Well done!")
 else:
 raise ValueError("Bad implementation")

test_rosenbrock()

```

Well done!

```

In []: steps = []
x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize_function(x0, rosenbrock, "CG")
print('Solution:', res.fun, res.x)

print()
print()

steps = []
res = minimize_function(x0, rosenbrock, "BFGS")
print('Solution:', res.fun, res.x)

```

Optimization terminated successfully.

Current function value: 0.000000

Iterations: 43

Function evaluations: 492

Gradient evaluations: 82

func:minimize\_function took: 0.0199 sec

Solution: 2.4922358041114047e-11 [0.99999973 0.99999947 0.99999892 0.9999978  
4 0.99999564]

Optimization terminated successfully.

Current function value: 0.000000

Iterations: 21

Function evaluations: 150

Gradient evaluations: 25

func:minimize\_function took: 0.0065 sec

Solution: 1.3311336330291604e-11 [0.9999998 0.9999996 0.99999921 0.9999984  
3 0.99999684]

In [ ]: