Chem/BioE C142/242 Spring 2024 - Tutorial 1

- GSI: Eric Wang (ericwangyz@berkeley.edu)
- Office Hours: **Tuesday 12-1 pm, Lewis 1**

Outline:

- Visualizing univariate and multivariate functions
- Time your function
- Minimization using CG/BFGS
- Golden section example

Visualizing univariate and multivariate functions

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

In [2]:
```
# Prior to Matplotlib 3.2.0, you will have to explicitly import Axes3D
# to enable 3d plot
from mpl_toolkits.mplot3d import Axes3D
```

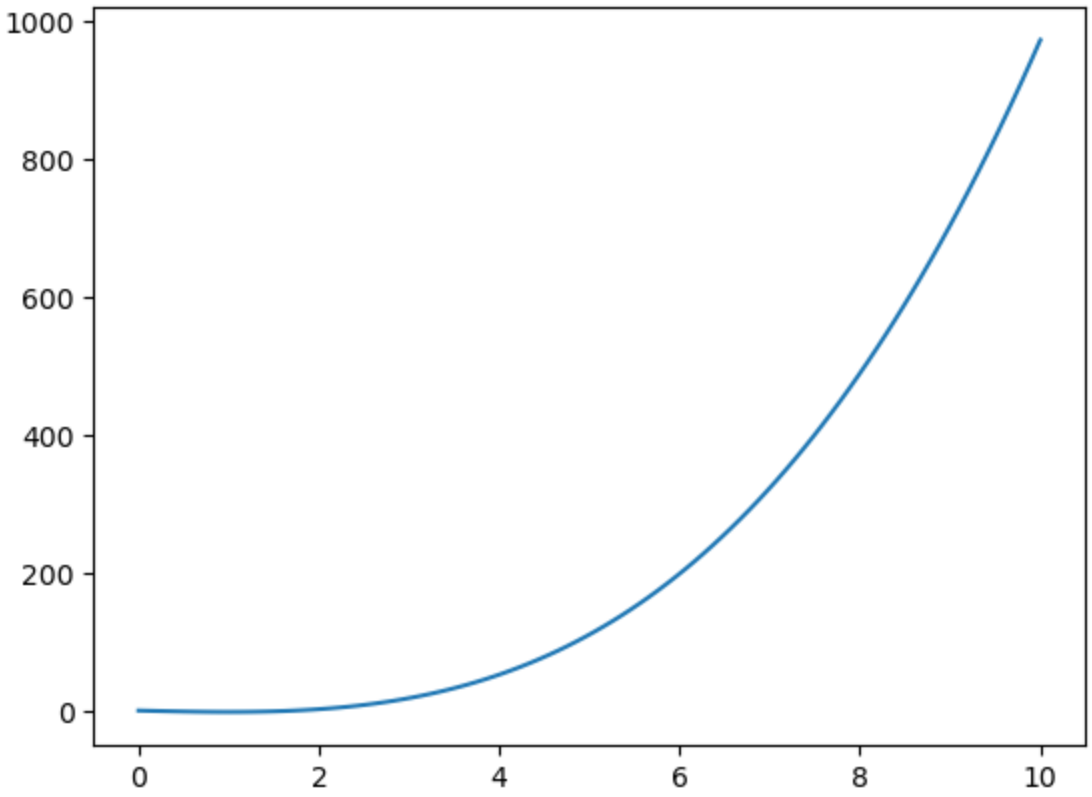
Univariate function

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

In [3]:
```
def func2d(x):
    f = x ** 3 - 3 * x + 2
    return f
```

In [4]:
```
# to plot func
x = np.linspace(0, 10, 100)
y = func2d(x)
plt.plot(x, y)
```

Out[4]:
```
[<matplotlib.lines.Line2D at 0x19dd2e5cec0>]
```

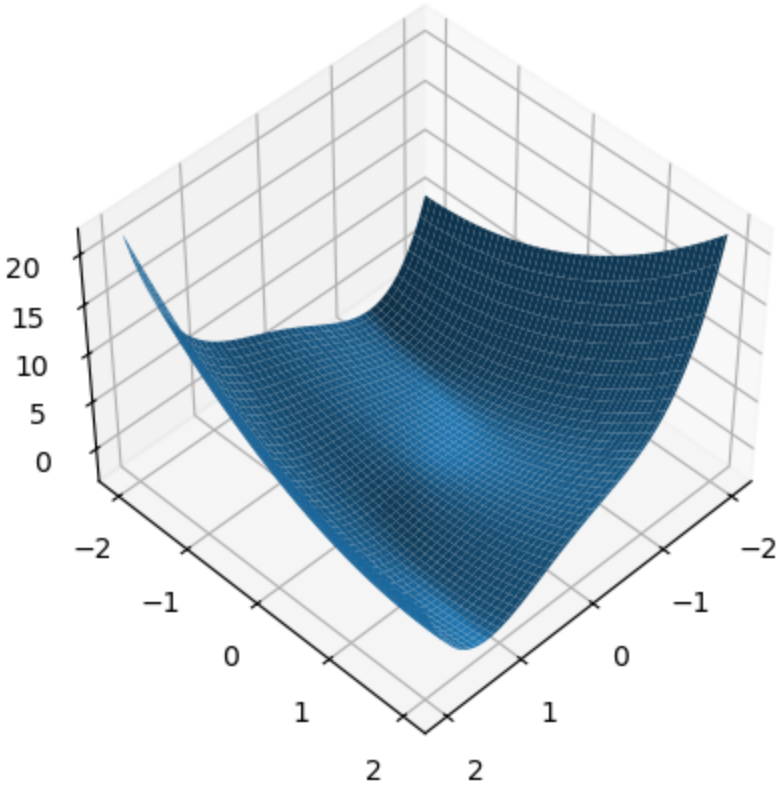


Multivariate function

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

In [5]:
```
def func3d(x, y):
    return x ** 4 - x ** 2 + y ** 2 - 2 * x * y - 2
```

In [6]:
```
a = np.linspace(-2, 2, 100)
x, y = np.meshgrid(a, a) # generate x, y grid
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_surface(x, y, func3d(x,y))
ax.view_init(45,45)
```



Time your function

With a magic method

In [7]:
```
def add(a, b):
    return a + b

%timeit add(1, 2)
```

```
55.4 ns ± 2.04 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

With decorators

Decorators wrap a function, and modify its behavior.

In [8]:
```
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 [9]:
```
@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: 2.0005 sec
```

Minimization using CG/BFGS

The multivariate Rosenbrock function is given below:

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

In [10]:
```
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!
```

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

In [11]:
```
from scipy.optimize import minimize
```

In [12]:
```
?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 the 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) LO	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) LO sp	x	x

where LO=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],  # may 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:      c:\users\artui\miniforge3\envs\c242\lib\site-packages\scipy\optimize\_min
imize.py
Type:      function
```

More on optimization: Scipy tutorial on optimization

In [13]:
```
@timeit
def minimize_rosenbrock(x0, method):
    """
    Minimize Rosenbrock function

    Parameters
```

```
    ---------
    x0: np.ndarray
        Starting point
    method: str
        Method for minimization

    Returns
    ------
    res: OptimzizeResult
        Result object of scipy optimization
    """
    res = minimize(rosenbrock, x0, method=method, options={"gtol": 1e-5, "disp": Tr
    return res
```

In [14]:
```
x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
```

In [15]:
```
res = minimize_rosenbrock(x0, "CG")
print(res.nit)
```

```
         Current function value: 0.000000
         Iterations: 54
         Function evaluations: 882
         Gradient evaluations: 145
func:minimize_rosenbrock took: 0.0515 sec
54
C:\Users\artui\miniforge3\envs\c242\Lib\site-packages\scipy\optimize\_minimize.py:706: OptimizeWarning: Desired error not necessarily achieved due to precision loss.
  res = _minimize_cg(fun, x0, args, jac, callback, **options)
```

In [16]:
```
res = minimize_rosenbrock(x0, "BFGS")
print(res.nit)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 21
         Function evaluations: 150
         Gradient evaluations: 25
func:minimize_rosenbrock took: 0.0130 sec
21
```

Golden section example

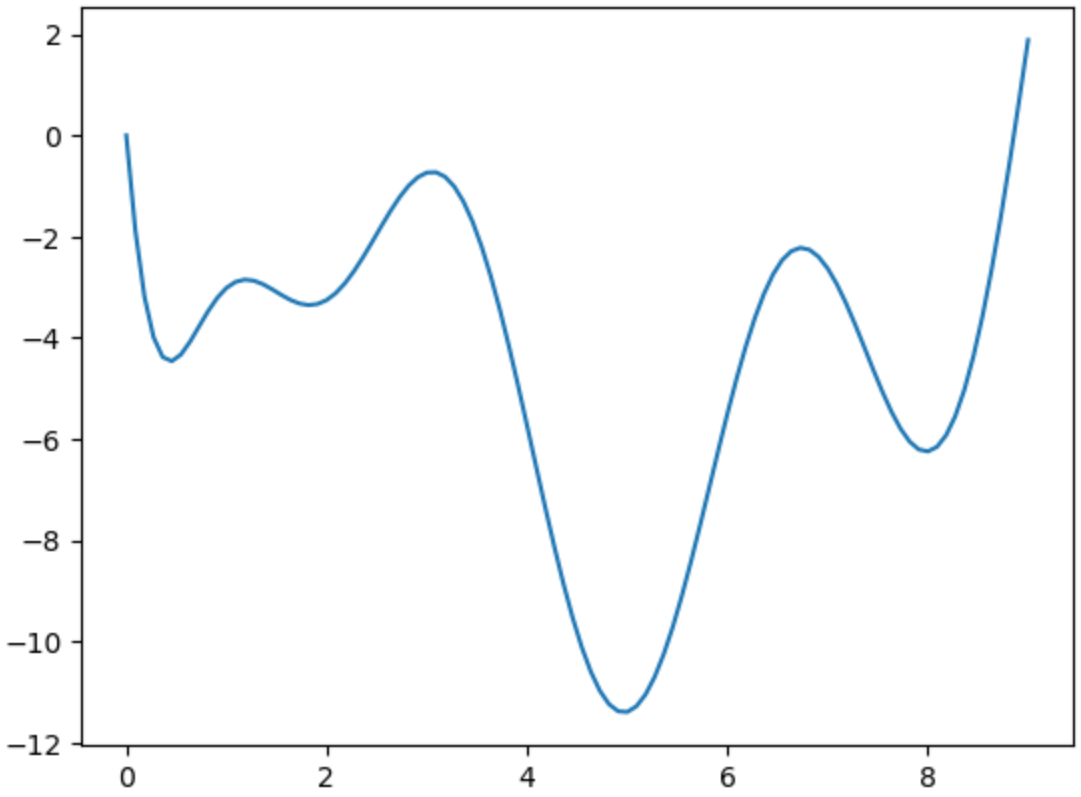
Given the following function, can you find its minimum value in range $[0, 9]$ using golden section?

In [17]:
```
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 [18]:
```
x = np.linspace(0, 9, 100)
plt.plot(x, func(x))
```

Out[18]:
```
[<matplotlib.lines.Line2D at 0x19dd4949fd0>]
```



In [19]:
```
def golden_section(func, start, end, reference, tol):
    if end - start < tol:
        return {'x':reference, 'y':func(reference)}
    else:
        if reference - start < end - reference:
            new_reference = end - (end - reference) * 0.618
            if func(new_reference) > func(reference):
                return golden_section(func, start, new_reference, reference, tol)
            else:
                return golden_section(func, reference, end, new_reference, tol)
        else:
            new_reference = start + (reference - start) * 0.618
            if func(new_reference) > func(reference):
                return golden_section(func, new_reference, end, reference, tol)
            else:
                return golden_section(func, start, reference, new_reference, tol)
```

In [20]:
```
golden_section(func, 0, 9, 9*0.618, 1e-5)
```

Out[20]:
```
{'x': 4.9642569571739665, 'y': -11.400276695145294}
```

In [ ]: