# Math 564 - Project 01

Sajjad Uddin Mahmud

October 14, 2025

## 1 Introduction

The goal of this project is to fit experimental data to a damped sinusoidal function with exponential decay and chirped frequency:

$$W(t) = A_0 + Ae^{-t/\tau} \sin\left[(\omega + \alpha t)t + \phi\right]. \tag{1}$$

The parameters $A_0, A, \tau, \omega, \alpha, \phi$ are determined by minimizing the least-squares objective

$$f(A_0, A, \tau, \omega, \alpha, \phi) = \sum_{k=1}^{n} \Big(W(t) - V_k\Big)^2. \tag{2}$$

## 2 Methods

Three optimization algorithms were tested with Strong Wolfe line search:

- Gradient Descent (GD)
- Conjugate Gradient Descent (CGD)
- Quasi-Newton with BFGS update

## 3 Initial Parameter Estimation

In this project, the initial guesses for the parameters

$$[A_0,\ A,\ \tau,\ \omega,\ \alpha,\ \phi]$$

were estimated automatically from the input data $(t, v)$.

- **DC offset ($A_0$):** estimated as the mean value of the last 10% of the data, assuming the signal has mostly decayed by the end.
- **Amplitude ($A$):** estimated as the maximum value of the first 10% of the data minus $A_0$.
- **Decay constant ($\tau$):** estimated from the ratio of the first and last peaks of the absolute value $|v - A_0|$ using an exponential decay relationship.
- **Angular frequency ($\omega$):** estimated from the time difference between the first two peaks of $(v - A_0)$.
- **Chirp rate ($\alpha$):** estimated by measuring how the instantaneous frequency changes across the first few peaks.
- **Phase ($\phi$):** estimated from the early portion of the data by projecting $v - A_0$ onto damped sine and cosine functions.

The resulting initial parameter vector used for optimization was:

$$x_0 = [\,13.7991429,\ 7.3678571,\ 0.0184512,\ 1745.32925,\ 9199.18445,\ -1.8692734\,].$$

# 4 Results

This section reports the final objective values, gradient norms, convergence status, and final parameter vectors for each method/line search pair using the same initial parameters. Iteration-by-iteration CSVs are included in the repository.

## 4.1 Gradient Descent + Armijo

The run hit the iteration cap (1000) without convergence:

$$\text{success} = \text{False}, \quad n_{\text{iter}} = 1000, \quad f_{\text{final}} = 4.41097748{\times}10^2, \quad \|\nabla f\|_{\text{final}} = 1.257{\times}10^2.$$

Final parameter estimate:

$$x_{\text{final}} = [\,13.793438,\ 7.388136,\ 0.012429,\ 1745.327762,\ 9199.184421,\ -1.945344\,].$$

### 4.2 Gradient Descent + Strong Wolfe

The run hit the iteration cap (1000) without convergence:

$$\text{success} = \text{False}, \quad n_{\text{iter}} = 1000, \quad f_{\text{final}} = 4.40559694{\times}10^2, \quad \|\nabla f\|_{\text{final}} = 2.620{\times}10^2.$$

Final parameter estimate:

$$x_{\text{final}} = [\,13.792896,\ 7.390559,\ 0.012377,\ 1745.327598,\ 9199.184418,\ -1.951098\,].$$

### 4.3 Conjugate Gradient Descent (Polak–Ribière) + Strong Wolfe

The CGD run hit the iteration cap (1000) without convergence:

$$\text{success} = \text{False}, \quad n_{\text{iter}} = 1000, \quad f_{\text{final}} = 4.23388574{\times}10^2, \quad \|\nabla f\|_{\text{final}} = 2.224{\times}10^2.$$

Final parameter estimate:

$$x_{\text{final}} = [\,13.708326,\ 8.079156,\ 0.011575,\ 1745.287823,\ 9199.183591,\ -1.932541\,].$$

### 4.4 BFGS + Strong Wolfe

The BFGS run converged successfully in 31 iterations:

$$\text{success} = \text{True}, \quad n_{\text{iter}} = 31, \quad f_{\text{final}} = 5.83750645, \quad \|\nabla f\|_{\text{final}} = 5.033{\times}10^{-8}.$$

Final parameter estimate:

$$x_{\text{final}} = [\,13.782528,\ 8.257336,\ 0.021865,\ 1835.829905,\ 927.148443,\ -2.036854\,].$$

## 5 Discussion

All four optimization configurations were tested using the same initial parameters and the same objective function. Among them, only the **BFGS with Strong Wolfe line search** successfully converged to a minimum. The other three methods—Gradient Descent (with Armijo and Strong Wolfe) and Conjugate Gradient Descent (with Strong Wolfe)—did not converge and reached at predefined iteration limit. This indicates that first-order methods can struggle on nonlinear problems like this one, while BFGS is far more reliable and efficient for finding the best fit.

# 6 Repository

All code and iteration logs for this project are available in a public GitHub repository:

`https://github.com/sajjad30148/WSU_Math564_Fall2025`

The repository includes the code, and result folders containing iterations from each run.

# 7 Code

The full Python implementation is provided below. It can also be found in the repository.

## 7.1 Main Python Script

Listing 1: project01_main.py

```

# ================================================================
# main.py
# ---------------------------------------------------------------
# purpose:
# run a line-search optimizer on the given dataset (FFD.csv) to fit
# a damped, chirped sinusoid:
# W(t) = A0 + A * exp(-t/tau) * sin((omega + alpha*t)*t + phi)
# with parameters x = [A0, A, tau, omega, alpha, phi].
#
# inputs:
# - FFD.csv (two columns: t, v)
#
# outputs (saved under ./results_Project1/):
# - iteration logs and summaries (produced by the optimizer utilities)
# - initial_guess.txt (the initial x0 used for the run)
# ================================================================

import pandas as pd
import numpy as np
import os

import functions as fn
from run_logger import RunLogger


# ----------------------------------------------------------------
# User Settings
# ----------------------------------------------------------------

# data and outputs
script_dir = os.path.dirname(os.path.abspath(__file__))
data_file = os.path.join(script_dir, "FFD.csv")
out_dir = os.path.join(script_dir, "results_Project1")
os.makedirs(out_dir, exist_ok = True)
```

```

# settings
alpha_bar = 1.0
c1 = 0.001
c2 = 0.5
rho = 0.5

# initial-guess overrides (set to None to use auto estimation)
init_a0 = None
init_a = None
init_tau = None
init_omega = None
init_alpha = None
init_phi = None


# line search options
line_search_dict = {
    "armijo": (
        fn.armijo_backtracking,
        dict(alpha_bar = alpha_bar, c1 = c1, rho = rho),
    ),
    "strong_wolfe": (
        fn.strong_wolfe,
        dict(alpha_bar = alpha_bar, c1 = c1, c2 = c2),
    ),
}

# choose line search
line_search = "strong_wolfe"

# optional extra overrides from user settings (can be {} if unused)
line_search_options_extra = {}
line_search_options_extra = line_search_options_extra if 'LS_EXTRA_OPTS'
    in globals() else {}

# optimizer options
optimizer_dict = {
    "gd": fn.gradient_descent,
    "cgd": fn.conjugate_gradient_descent,
    "bfgs": fn.quasi_newton_bfgs,
}

# choose optimizer
optimizer = "bfgs"
```

```python


# ---------------------------------------------------------------
# settings for optimization
# ---------------------------------------------------------------

# function wrapper
def f_wrapper(x):
    f, _ = objective_function_and_gradient(x, t, v)
    return float(f)

# gradient wrapper
def g_wrapper(x):
    _, g = objective_function_and_gradient(x, t, v)
    return np.asarray(g, dtype=float)

# line search selection based on user choice
ls_func, ls_options = line_search_dict[line_search]

# optimizer selection based on user choice
opt_func = optimizer_dict[optimizer]

# logger
run_tag = "project01"

# ---------------------------------------------------------------
# objective function and gradient
# ---------------------------------------------------------------

def objective_function_and_gradient(x, tk, v):
    """
    compute least-squares objective f(x) = sum((W - v)^2)

    Given function:
        W(t) = A0 + A * exp(-t / tau) * sin((omega + alpha * t) * t + phi)

    parameters:
        x = [A0, A, tau, omega, alpha, phi]

    inputs:
        tk = time samples, 1D array
        v = observed data, 1D array

    returns:
        f = objective function value
```

```python
        grad = gradient, 1D array of same length as x

    """
    # parameters
    a0, a, tau, w, alpha, phi = x

    phase = (w + alpha * tk) * tk + phi
    E = np.exp(-tk / tau)
    Sk = E * np.sin(phase)
    Ck = E * np.cos(phase)

    # model
    W = a0 + a * Sk

    # residual
    r = W - v

    # objective function value
    f = np.sum(r ** 2)

    # gradient
    grad = 2 * np.array([
        np.sum(r * 1.0), # df/dA0
        np.sum(r * Sk), # df/dA
        np.sum(r * (a * Sk * (tk / tau**2))), # df/dtau
        np.sum(r * (a * tk * Ck)), # df/domega
        np.sum(r * (a * (tk**2) * Ck)), # df/dalpha
        np.sum(r * (a * Ck)) # df/dphi
    ])

    return f, grad


# ----------------------------------------------------------------
# initial guess estimation
# ----------------------------------------------------------------

def estimate_initial_guess(t, v):
    """
    Initial guesses for [A0, A, tau, omega, alpha, phi] from (t, v).

    Method:
      - a0 : Mean of last 10% of samples.
      - a : Max of first 10% minus a0.
      - tau : Decay from first/last peak of |v - a0|.
```

```
      - omega: Period from first two maxima.
      - alpha: Slope of freq vs. peak time.
      - phi : Phase from projections onto decay sin/cos.

    """

    # prepare
    t = np.asarray(t); v = np.asarray(v)
    order = np.argsort(t)
    t_sorted = t[order]
    v_sorted = v[order]

    # a0 (dc offset)
    tail_frac = 0.10
    n_tail = max(1, int(np.ceil(v_sorted.size * tail_frac)))
    init_a0 = float(np.mean(v_sorted[-n_tail:]))

    # a (amplitude near t=0)
    head_frac = 0.10
    n_head = max(1, int(np.ceil(v_sorted.size * head_frac)))
    init_a = float(np.max(v_sorted[:n_head]) - init_a0)
    if init_a <= 0:
        init_a = abs(init_a)

    # residual and simple peak indices
    y = v_sorted - init_a0
    a = np.abs(y)
    # peaks of |y| (for decay)
    pk_abs = np.where((a[1:-1] > a[:-2]) & (a[1:-1] >= a[2:]))[0] + 1
    # peaks of y itself (for period)
    pk_y = np.where((y[1:-1] > y[:-2]) & (y[1:-1] >= y[2:]))[0] + 1

    # tau (decay time)
    if pk_abs.size >= 2:
        p1 = max(a[pk_abs[0]], 1e-12)
        pL = max(a[pk_abs[-1]], 1e-12)
        ratio = max(p1 / pL, 1 + 1e-6) # ensure log > 0
        init_tau = float((t_sorted[pk_abs[-1]] - t_sorted[pk_abs[0]]) /
            np.log(ratio))
        dt = max(1e-12, float(np.min(np.diff(t_sorted))))
        init_tau = max(init_tau, dt)
    else:
        init_tau = float(t_sorted[-1] - t_sorted[0])

    # omega (base angular freq)
```

```
    if pk_y.size >= 2:
        T = float(t_sorted[pk_y[1]] - t_sorted[pk_y[0]])
        init_omega = float(2.0 * np.pi / max(T, 1e-12))
    else:
        init_omega = float(2.0 * np.pi / max(t_sorted[-1] - t_sorted[0], 1e-12))

    # alpha (chirp)
    init_alpha = 0.0
    if pk_y.size >= 2:
        peak_times = t_sorted[pk_y]
        periods = np.diff(peak_times)
        freqs = 1.0 / np.maximum(periods, 1e-12)
        freq_times = peak_times[1:]

        if freq_times.size > 1:
            coeffs = np.polyfit(freq_times, freqs, 1)
            init_alpha = coeffs[0]
        else:
            init_alpha = 0.0


    # phi (phase)
    t0 = 2.0 * np.pi / max(init_omega, 1e-12)
    t_end = t_sorted[0] + 2.0 * t0
    mask = t_sorted <= t_end
    if not np.any(mask):
        mask = np.ones_like(t_sorted, dtype = bool)

    tt = t_sorted[mask]
    yy = y[mask]
    theta = (init_omega + init_alpha * tt) * tt
    E = np.exp(-tt / max(init_tau, 1e-12))
    S = float(np.dot(yy, E * np.sin(theta)))
    C = float(np.dot(yy, E * np.cos(theta)))
    init_phi = float(np.arctan2(C, S))

    return np.array([init_a0, init_a, init_tau, init_omega, init_alpha, init_phi], dtype = float)



# ---------------------------------------------------------------
# load data
# ---------------------------------------------------------------
```

```python

# give error if data file not found
if not os.path.exists(data_file):
    raise FileNotFoundError(f"data file not found: {data_file}")

# read data
df = pd.read_csv(data_file, header = None, usecols = [0,1], names =
    ["t","v"])
t = df["t"].to_numpy(dtype = float)
v = df["v"].to_numpy(dtype = float)


# ----------------------------------------------------------------
# run optimization
# ----------------------------------------------------------------


# initial guess: use overrides if set, else auto
auto_a0, auto_a, auto_tau, auto_omega, auto_alpha, auto_phi =
    estimate_initial_guess(t, v)

a0 = init_a0 if init_a0 is not None else auto_a0
a = init_a if init_a is not None else auto_a
tau = init_tau if init_tau is not None else auto_tau
omega = init_omega if init_omega is not None else auto_omega
alpha = init_alpha if init_alpha is not None else auto_alpha
phi = init_phi if init_phi is not None else auto_phi

x0 = np.array([a0, a, tau, omega, alpha, phi], dtype = float)

# save the initial guess used
with open(os.path.join(out_dir,
    f"{optimizer}_{line_search}initial_guess.txt"), "w") as fh:
    fh.write("x0 = [A0, A, tau, omega, alpha, phi]\n")
    fh.write(np.array2string(x0, precision=7, separator=", ") + "\n")

# optimizer options
opts = {
    "max_iter": 1000,
    "tol": 1e-6,
    "line_search_opts": ls_options,
    "save_flag": True,
    "optimizer": optimizer,
    "line_search": line_search,
    "out_dir": out_dir,
```

```
    "run_tag": run_tag,
    "alpha0": alpha_bar,
    "c1": c1,
    "c2": c2,
}

# run
result = opt_func(f_wrapper, g_wrapper, x0, ls_func, opts=opts)

# print results
print("success:", result["success"])
print("iters :", result["n_iter"])
print("f_final:", result["f"])
print("x_final:", result["x"])
```

## 7.2 Functions

Listing 2: functions.py

```

# ================================================================
# functions.py
# ---------------------------------------------------------------
# purpose:
# provide line-search methods and optimizers.
#
# inputs expected by callers:
# - f(x): scalar objective
# - grad(x): gradient vector
# - x0: initial parameter vector
# - line_search: one of {armijo_backtracking, strong_wolfe}
# - opts: dict with keys such as:
# max_iter, tol, line_search_opts, save_flag,
# optimizer, line_search, out_dir, run_tag, parameters for line search
#
# outputs:
# each optimizer returns a dict:
# {"x", "f", "g", "n_iter", "n_func_eval", "n_grad_eval", "success"}
# ================================================================


import numpy as np
import pandas as pd
from run_logger import RunLogger


# ----------------------------------------------------------------
# line search methods
# ----------------------------------------------------------------

def armijo_backtracking(f, grad, xk, pk, alpha_bar=1.0, c1=1e-3, rho=0.5,
                        max_backtracks=50, min_alpha=None, **kwargs):
    """
    Armijo backtracking line search (simple, safe).

    Finds alpha > 0 s.t.
        f(xk + alpha*pk) <= f(xk) + c1 * alpha * grad(xk)^T pk
    by shrinking alpha <- rho * alpha.

    Inputs
        f, grad : callables (f(x)->scalar, grad(x)->vector)
```

```
        xk : current point
        pk : descent direction (requires grad(xk)^T pk < 0)
        alpha_bar : initial trial step
        c1 : Armijo parameter in (0, 1)
        rho : shrink factor in (0, 1)
        max_backtracks : cap on shrink steps
        min_alpha : optional floor for alpha (default: machine eps)

    Returns
        alpha : accepted step
        f_new : f(xk + alpha*pk)
        n_eval : number of f-evals during the search (excludes f(xk))
    """
    # --- basic checks ---
    if not (0.0 < c1 < 1.0):
        raise ValueError("c1 must be in (0, 1).")
    if not (0.0 < rho < 1.0):
        raise ValueError("rho must be in (0, 1).")

    xk = np.asarray(xk, dtype=float)
    pk = np.asarray(pk, dtype=float)

    fk = float(f(xk))
    gk = np.asarray(grad(xk), dtype=float)
    slope0 = float(np.dot(gk, pk))
    if not np.isfinite(slope0):
        raise ValueError("non-finite directional derivative at xk")
    if slope0 >= 0.0:
        raise ValueError("pk is not a descent direction (grad^T pk >= 0)")

    alpha = float(alpha_bar)
    n_eval = 0
    alpha_floor = np.finfo(float).eps if min_alpha is None else float(min_alpha)

    # --- backtracking loop ---
    backtrack_count = 0
    while True:
        x_trial = xk + alpha * pk
        f_trial = float(f(x_trial))
        n_eval += 1

        # if f_trial is not finite, keep shrinking until finite or floor
        while not np.isfinite(f_trial) and alpha > alpha_floor and backtrack_count < int(max_backtracks):
```

```
            alpha *= rho
            backtrack_count += 1
            x_trial = xk + alpha * pk
            f_trial = float(f(x_trial))
            n_eval += 1

        # Armijo sufficient decrease
        if np.isfinite(f_trial) and f_trial <= fk + c1 * alpha * slope0:
            return alpha, f_trial, n_eval

        # shrink step
        alpha *= rho
        backtrack_count += 1

        # stop if hit floor or backtrack cap
        if (alpha <= alpha_floor) or (backtrack_count >= int(max_backtracks)):
            # return best effort at current (possibly shrunk) alpha
            x_trial = xk + alpha * pk
            f_trial = float(f(x_trial))
            n_eval += 1
            return alpha, f_trial, n_eval




def strong_wolfe(f, grad, xk, pk, alpha_bar = 1.0, c1 = 1e-4, c2 = 0.9):
    """
    strong wolfe line search

    purpose
        find a step length alpha > 0 that satisfies the strong wolfe conditions
        along a given descent direction pk at the current point xk

    conditions enforced
        armijo sufficient descent:
            f(xk + alpha pk) <= f(xk) + c1 * alpha * grad(xk)^T pk
        strong curvature:
            abs(grad(xk + alpha pk)^T pk) <= -c2 * grad(xk)^T pk
        with 0 < c1 < c2 < 1

    inputs
        f callable, f(x) -> scalar objective value
        grad callable, grad(x) -> gradient vector at x
```

```
        xk current point
        pk search direction at xk, should satisfy grad(xk)^T pk < 0
        alpha_bar initial trial step length, default 1.0
        c1 armijo parameter in (0, 1), default 0.001
        c2 curvature parameter in (c1, 1), default 0.5

    returns
        alpha accepted step length satisfying strong wolfe, or a fallback if limits are hit
        f_new objective value at xk + alpha pk
        n_eval number of objective evaluations performed inside the line search

    notes
        this function evaluates both f and grad at trial points
        only the count of f evaluations is returned as n_eval for consistency with armijo_backtracking

    references
        nocedal and wright, numerical optimization, algorithms 3.5 and 3.6
    """

    # evaluate objective, gradient and search direction at current point
    xk = np.asarray(xk, dtype = float)
    pk = np.asarray(pk, dtype = float)
    fk = float(f(xk))
    gk = np.asarray(grad(xk), dtype = float)

    # initial directional derivative (grad^T pk)
    slope0 = float(np.dot(gk, pk))

    if not np.isfinite(fk) or not np.all(np.isfinite(gk)): # guard: directional derivative must be finite
        raise ValueError("non finite fk or grad(xk)")

    if slope0 >= 0.0: # guard: pk must be a descent direction (grad^T pk < 0)
        raise ValueError("pk is not a descent direction at xk since grad(xk)^T pk >= 0")

    # helper subfunctions for phi: phi(alpha) = f(xk + alpha pk)
    def phi(alpha):
        return float(f(xk + alpha * pk))

```

```
    # helper subfunctions for derivative of phi: dphi(alpha) = grad(xk+alpha pk)^T pk
    def dphi(alpha):
        return float(np.dot(grad(xk + alpha * pk), pk))

    n_eval = 0 # to count of objective evaluations performed inside line search

    # phase a: bracketing as in algorithm 3.5
    alpha_prev = 0.0
    f_prev = fk
    alpha = float(alpha_bar)
    i = 0

    while True:
        f_current = phi(alpha)
        n_eval += 1

        # check armijo condition, modify bracket and enter zoom phase if satisfied
        if (f_current > fk + c1 * alpha * slope0) or (i > 0 and f_current >= f_prev):
            alpha_lo = alpha_prev
            alpha_hi = alpha
            f_lo = f_prev
            f_hi = f_current
            break

        # compute directional derivative at current alpha
        slope_alpha = dphi(alpha)

        # check strong wolfe curvature condition, if satisfied return alpha
        if abs(slope_alpha) <= c2 * abs(slope0):
            return alpha, f_current, n_eval

        # check for too large step (slope is nonnegative), modify bracket and enter zoom phase if so
        if slope_alpha >= 0.0:
            alpha_lo = alpha
            alpha_hi = alpha_prev
            f_lo = f_current
            f_hi = f_prev
            break

        # otherwise, expand the step and continue the bracketing phase
```

```
        alpha_prev = alpha
        f_prev = f_current
        alpha = 2.0 * alpha # double the step
        i += 1

    # phase b: zoom as in algorithm 3.6
    while True:

        # bisection to find a trial alpha in (alpha_lo, alpha_hi)
        alpha_j = 0.5 * (alpha_lo + alpha_hi)

        # evaluate function at trial alpha_j
        f_j = phi(alpha_j)
        n_eval += 1

        # Armijo condition check inside zoom (use fk + c1 * alpha_j * slope0)
        if (f_j > fk + c1 * alpha_j * slope0) or (f_j >= f_lo):
            alpha_hi = alpha_j
            continue

        # evaluate derivative at trial alpha_j
        slope_j = dphi(alpha_j)

        # check strong wolfe curvature condition inside zoom
        if abs(slope_j) <= c2 * abs(slope0):
            return alpha_j, f_j, n_eval

        # sign test to decide which side to continue the search
        if slope_j * (alpha_hi - alpha_lo) >= 0.0:
            alpha_hi = alpha_lo

        # move the lower bracket to alpha_j since Armijo passed and curvature failed
        alpha_lo = alpha_j
        f_lo = f_j

    # If the loop exits unexpectedly, return the low end of the bracket which satisfies Armijo
    f_star = phi(alpha_lo)
    n_eval += 1

    return alpha_lo, f_star, n_eval
```

```

# ---------------------------------------------------------------
# optimization drivers
# ---------------------------------------------------------------

def gradient_descent(f, grad, x0, line_search, opts=None):
    """
    Gradient descent with line search (simple, safe).

    Returns a dict with: x, f, g, n_iter, n_func_eval, n_grad_eval, success
    """
    opts = {} if opts is None else dict(opts)

    # --- user options ---
    max_iter = int(opts.get('max_iter', 1000))
    tol = float(opts.get('tol', 1e-6))
    ls_opts = dict(opts.get('line_search_opts', {})) # will be merged with meta defaults
    save_flag = bool(opts.get('save_flag', True))

    optimizer = str(opts.get('optimizer', 'gradient_descent'))
    ls_name = str(opts.get('line_search', 'armijo_backtracking'))
    out_dir = str(opts.get('out_dir', './results'))
    run_tag = str(opts.get('run_tag', 'None'))

    # meta defaults for common line searches
    meta_alpha0 = float(opts.get('alpha0', 1.0))
    meta_c1 = float(opts.get('c1', ls_opts.get('c1', 1e-3)))
    meta_c2 = float(opts.get('c2', ls_opts.get('c2', 0.5))) # used by strong-wolfe

    # merge meta defaults only if not provided explicitly
    ls_opts.setdefault('alpha_bar', meta_alpha0)
    ls_opts.setdefault('c1', meta_c1)
    # c2 is harmless for Armijo; strong-wolfe will use it
    ls_opts.setdefault('c2', meta_c2)

    # tiny movement floor (prevents endless micro-steps)
    step_floor = float(opts.get('step_floor', 1e-16))

    # --- optional logger ---
    logger = None
    if save_flag:
        try:
```

```
            logger = RunLogger(out_dir=out_dir,
                               optimizer=optimizer,
                               line_search=ls_name,
                               alpha0=meta_alpha0, c1=meta_c1, c2=meta_c2,
                               max_iter=max_iter, gtol=tol, run_tag=run_tag)
        except NameError:
            # if RunLogger is not defined, disable logging quietly
            logger = None
            save_flag = False

    # --- initialization ---
    xk = np.asarray(x0, dtype=float)
    fk = float(f(xk))
    gk = np.asarray(grad(xk), dtype=float)

    n_func_eval = 1 # f(x0)
    n_grad_eval = 1 # grad(x0)

    for k in range(1, max_iter + 1):
        gk_norm = float(np.linalg.norm(gk, ord=2))
        if gk_norm <= tol:
            success = True
            if logger is not None:
                logger.add_eval_counts(f_evals=n_func_eval - 1,
                    g_evals=n_grad_eval - 1)
                logger.finalize(success=success, n_iter=k - 1, f_final=fk,
                                g_final=gk_norm, x_final=xk)
                logger.close()
            return {"x": xk, "f": fk, "g": gk, "n_iter": k - 1,
                    "n_func_eval": n_func_eval, "n_grad_eval": n_grad_eval,
                    "success": success}

        pk = -gk
        p_norm = float(np.linalg.norm(pk, ord=2))

        # line search (alpha, f_new, n_eval_f)
        alpha, f_new, n_eval_f = line_search(f, grad, xk, pk, **ls_opts)
        n_func_eval += int(n_eval_f)

        # guard: non-finite trial or microscopic step
        if not np.isfinite(f_new) or alpha * p_norm <= step_floor:
            success = False
            if logger:
                logger.add_eval_counts(f_evals=n_eval_f)
                logger.finalize(success=success, n_iter=k - 1, f_final=fk,
```

```
                                g_final=gk_norm, x_final=xk)
                logger.close()
            return {"x": xk, "f": fk, "g": gk, "n_iter": k - 1,
                    "n_func_eval": n_func_eval, "n_grad_eval": n_grad_eval,
                    "success": success}

        # take step
        xk_next = xk + alpha * pk
        fk_next = float(f_new)
        gk_next = np.asarray(grad(xk_next), dtype=float)
        n_grad_eval += 1

        if logger:
            df_abs = float(abs(fk - fk_next))
            logger.add_eval_counts(f_evals=n_eval_f)
            logger.log_iter(int(k), float(fk_next),
                            float(np.linalg.norm(gk_next, 2)),
                            float(alpha), float(p_norm), float(df_abs))

        # prepare next iteration
        xk, fk, gk = xk_next, fk_next, gk_next

    # max iterations reached
    success = False
    if logger:
        logger.finalize(success=success, n_iter=max_iter, f_final=fk,
                        g_final=float(np.linalg.norm(gk, 2)), x_final=xk)
        logger.close()

    return {"x": xk, "f": fk, "g": gk, "n_iter": max_iter,
            "n_func_eval": n_func_eval, "n_grad_eval": n_grad_eval,
            "success": success}


def conjugate_gradient_descent(f, grad, x0, line_search, opts = None): # Polak-Ribiere apporach
    """
    Polak-Ribiere nonlinear conjugate gradient method with line search

    purpose
        minimize a smooth unconstrained function f using the nonlinear conjugate
        gradient method (Polak--Ribiere formula) with a line search. The
        implementation applies a restart (beta set to zero) when the Polak--Ribiere
```

```
        coefficient becomes negative (commonly used to maintain descent).

    inputs
        f callable, f(x) -> scalar objective value
        grad callable, grad(x) -> gradient vector at x
        x0 starting point
        line_search callable, line_search(f, grad, xk, pk, ...) -> (alpha, f_new, n_eval)
                    a line search function that takes f, grad, current point xk and search direction pk
                    and returns a step length alpha > 0 satisfying some conditions along with
                    the new objective value f_new = f(xk + alpha pk) and the number of objective evaluations n_eval

        opts dictionary of all the options for the optimization run, possible keys are
                       max_iter maximum number of iterations
                       tol tolerance on gradient norm for termination
                       line_search_opts dictionary of options for the line search function
                       save_flag whether to save iteration history, default True
                       optimizer name of the optimizer, default 'conjugate_gradient_descent'
                       line_search name of the line search, default 'armijo_backtracking'
                       out_dir directory to save results, default './results'
                       run_tag tag to identify the run, default 'None'
                       alpha0 initial step length guess for line search, default 1.0


    outputs
        x best point found
        f_val objective value at best point
        k number of iterations performed
        n_eval total number of objective evaluations performed
        grad_norm norm of gradient at best point
        msg termination message

    """

    opts = {} if opts is None else dict(opts)
```

```python
    max_iter = int(opts.get('max_iter', 1000)) # maximum number of iterations
    tol = float(opts.get('tol', 1e-6)) # tolerance on gradient norm for termination
    ls_opts = dict(opts.get('line_search_opts', {})) # options for the line search function
    save_flag = bool(opts.get('save_flag', True)) # whether to save iteration history

    optimizer = str(opts.get('optimizer', 'conjugate_gradient_descent')) # name of the optimizer
    ls_name = str(opts.get('line_search', 'armijo_backtracking')) # name of the line search
    out_dir = str(opts.get("out_dir", "./results")) # directory to save results
    run_tag = str(opts.get("run_tag", "None")) # tag to identify the run

    meta_alpha0 = float(opts.get("alpha0", 1.0)) # initial step length guess for line search
    meta_c1 = opts.get("c1",ls_opts.get("c1", 0.001)) # armijo parameter
    meta_c2 = opts.get("c2",ls_opts.get("c2", 0.5)) # curvature parameter for strong wolfe

    logger = None
    if save_flag:
        logger = RunLogger(out_dir = out_dir,
                           optimizer = optimizer,
                           line_search = ls_name,
                           alpha0 = meta_alpha0,
                           c1 = meta_c1,
                           c2 = meta_c2,
                           max_iter = max_iter,
                           gtol = tol,
                           run_tag = run_tag)

    # initialization
    xk = np.asarray(x0, dtype = float)
    fk = float(f(xk))
    gk = np.asarray(grad(xk), dtype = float)

    n_func_eval = 1 # count f evaluation at initial point
    n_grad_eval = 1 # count grad evaluation at initial point

    pk = -gk # initial steepest descent direction

```

```python
    for k in range(1, max_iter + 1):

        gk_norm = float(np.linalg.norm(gk, ord = 2)) # compute gradient norm

        if gk_norm <= tol: # check optimality
            success = True

            if logger is not None:
                # exclude initial evaluations from the counts
                logger.add_eval_counts(f_evals = n_func_eval - 1, g_evals = n_grad_eval - 1)
                logger.finalize(success = success, n_iter = k - 1, f_final = fk, g_final = float(np.linalg.norm(gk, 2)), x_final = xk)
                logger.close()

            return {"x" : xk,
                    "f" : fk,
                    "g" : gk,
                    "n_iter" : k - 1,
                    "n_func_eval" : n_func_eval,
                    "n_grad_eval" : n_grad_eval,
                    "success" : success
                    }

        p_norm = float(np.linalg.norm(pk, ord = 2)) # norm of search direction

        if np.dot(gk, pk) >= 0:
            pk = -gk

        alpha, f_new, n_eval_f = line_search(f, grad, xk, pk, **ls_opts) # call line search
        n_func_eval += int(n_eval_f) # update function evaluation count

        xk_next = xk + alpha * pk # new point
        fk_next = float(f_new) # new objective value
        gk_next = np.asarray(grad(xk_next), dtype = float) # new gradient
        n_grad_eval += 1 # update gradient evaluation count

        df_abs = abs(fk - fk_next) # absolute change in objective

        if logger:
            logger.add_eval_counts(f_evals = n_eval_f)
```

```
            logger.log_iter(int(k), float(fk_next),
                float(np.linalg.norm(gk_next, 2)), float(alpha),
                float(p_norm), float(df_abs))

        # compute beta using Polak-Ribiere formula
        yk = gk_next - gk
        eps = 1e-12
        beta = np.dot(gk_next, yk) / max(np.dot(gk, gk), eps) # beta_pr = (g_{k+1}^t (g_{k+1} - g_k)) / (g_k^t g_k)
        beta = max(beta, 0) # ensure beta is non-negative, (natural restart)


        # # update search direction
        pk = -gk_next + beta * pk

        # Restart condition with v = 0.2
        cos_angle = abs(np.dot(gk_next, pk)) / (np.linalg.norm(gk_next) * np.linalg.norm(pk) + 1e-12)
        if cos_angle > 0.2:
            pk = -gk_next

        # accept the new point
        xk = xk_next
        fk = fk_next
        gk = gk_next

    # iteration limit reached without convergence
    success = False

    if logger:
        logger.finalize(success = success, n_iter = max_iter, f_final = fk, g_final = float(np.linalg.norm(gk, 2)), x_final = xk)
        logger.close()

    return {"x" : xk,
            "f" : fk,
            "g" : gk,
            "n_iter" : max_iter,
            "n_func_eval" : n_func_eval,
            "n_grad_eval" : n_grad_eval,
            "success" : success
    }
```

```python
def quasi_newton_bfgs(f, grad, x0, line_search, opts = None):
    """
    quasi-newton method with BFGS update and line search
    """
    opts = {} if opts is None else dict(opts)
    max_iter = int(opts.get('max_iter', 1000)) # maximum number of iterations
    tol = float(opts.get('tol', 1e-6)) # tolerance on gradient norm for termination
    ls_opts = dict(opts.get('line_search_opts', {})) # options for the line search function
    save_flag = bool(opts.get('save_flag', True)) # whether to save iteration history

    optimizer = str(opts.get('optimizer', 'quasi_newton_bfgs')) # name of the optimizer
    ls_name = str(opts.get('line_search', 'strong_wolfe')) # name of the line search
    out_dir = str(opts.get("out_dir", "./results")) # directory to save results
    run_tag = str(opts.get("run_tag", "None")) # tag to identify the run

    meta_alpha0 = float(opts.get("alpha0", 1.0)) # initial step length guess for line search
    meta_c1 = opts.get("c1",ls_opts.get("c1", 0.001)) # armijo parameter
    meta_c2 = opts.get("c2",ls_opts.get("c2", 0.5)) # curvature parameter for strong wolfe

    logger = None
    if save_flag:
        logger = RunLogger(out_dir = out_dir,
                           optimizer = optimizer,
                           line_search = ls_name,
                           alpha0 = meta_alpha0,
                           c1 = meta_c1,
                           c2 = meta_c2,
                           max_iter = max_iter,
                           gtol = tol,
                           run_tag = run_tag)

    # initialization
    xk = np.asarray(x0, dtype = float)
    fk = float(f(xk))
    gk = np.asarray(grad(xk), dtype = float)

```

```python
    n_func_eval = 1 # count f evaluation at initial point
    n_grad_eval = 1 # count grad evaluation at initial point

    n_dim = xk.size
    Hk = np.eye(n_dim) # initial inverse hessian approximation (identity)

    for k in range(1, max_iter + 1):

        gk_norm = float(np.linalg.norm(gk, ord = 2)) # compute gradient norm

        if gk_norm <= tol: # check optimality
            success = True

            if logger is not None:
                # exclude initial evaluations from the counts
                logger.add_eval_counts(f_evals = n_func_eval - 1, g_evals = n_grad_eval - 1)
                logger.finalize(success = success, n_iter = k - 1, f_final = fk, g_final = float(np.linalg.norm(gk, 2)), x_final = xk)
                logger.close()

            return {"x" : xk,
                    "f" : fk,
                    "g" : gk,
                    "n_iter" : k - 1,
                    "n_func_eval" : n_func_eval,
                    "n_grad_eval" : n_grad_eval,
                    "success" : success
                    }

        # compute search direction
        pk = -np.dot(Hk, gk)
        p_norm = float(np.linalg.norm(pk, ord = 2))

        # call line search
        alpha, f_new, n_eval_f = line_search(f, grad, xk, pk, **ls_opts)
        n_func_eval += int(n_eval_f)

        # update step
        xk_next = xk + alpha * pk
        fk_next = float(f_new)
        gk_next = np.asarray(grad(xk_next), dtype = float)
        n_grad_eval += 1
```

```python

        df_abs = abs(fk - fk_next) # absolute change in objective
        if logger:
            logger.add_eval_counts(f_evals = n_eval_f)
            logger.log_iter(int(k), float(fk_next),
                float(np.linalg.norm(gk_next, 2)), float(alpha),
                float(p_norm), float(df_abs))

        # BFGS update
        sk = xk_next - xk # step
        yk = gk_next - gk # gradient difference
        syk = np.dot(sk, yk) # sk^T yk

        if syk > 0.0: # update Hk only if sk^T yk is sufficiently positive
             to maintain positive definiteness
            rho_k = 1.0 / syk
            I = np.eye(n_dim)
            syT = np.outer(sk, yk)
            ysT = np.outer(yk, sk)
            ssT = np.outer(sk, sk)
            v = I - rho_k * syT
            Hk = np.dot(v, np.dot(Hk, v.T)) + rho_k * ssT # BFGS formula

        # accept the new point
        xk = xk_next
        fk = fk_next
        gk = gk_next

    # iteration limit reached without convergence
    success = False
    if logger:
        logger.finalize(success = success, n_iter = max_iter, f_final =
            fk, g_final = float(np.linalg.norm(gk, 2)), x_final = xk)
        logger.close()

    return {"x" : xk,
            "f" : fk,
            "g" : gk,
            "n_iter" : max_iter,
            "n_func_eval" : n_func_eval,
            "n_grad_eval" : n_grad_eval,
            "success" : success
    }
```