

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:

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

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

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

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 (τ):** estimated from the ratio of the first and last peaks of the absolute value $|v - A_0|$ using an exponential decay relationship.
- **Angular frequency (ω):** estimated from the time difference between the first two peaks of $(v - A_0)$.
- **Chirp rate (α):** estimated by measuring how the instantaneous frequency changes across the first few peaks.
- **Phase (ϕ):** 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:

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

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:

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

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:

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

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

```
1
2 # =====
3 # main.py
4 # -----
5 # purpose:
6 # run a line-search optimizer on the given dataset (FFD.csv) to fit
7 # a damped, chirped sinusoid:
8 #  $W(t) = A_0 + A * \exp(-t/\tau) * \sin((\omega + \alpha * t) * t + \phi)$ 
9 # with parameters  $x = [A_0, A, \tau, \omega, \alpha, \phi]$ .
10 #
11 # inputs:
12 # - FFD.csv (two columns:  $t, v$ )
13 #
14 # outputs (saved under ./results_Project1/):
15 # - iteration logs and summaries (produced by the optimizer utilities)
16 # - initial_guess.txt (the initial  $x_0$  used for the run)
17 # =====
18
19 import pandas as pd
20 import numpy as np
21 import os
22
23 import functions as fn
24 from run_logger import RunLogger
25
26
27 # -----
28 # User Settings
29 # -----
30
31 # data and outputs
32 script_dir = os.path.dirname(os.path.abspath(__file__))
33 data_file = os.path.join(script_dir, "FFD.csv")
34 out_dir = os.path.join(script_dir, "results_Project1")
35 os.makedirs(out_dir, exist_ok = True)
```

```

36
37 # settings
38 alpha_bar = 1.0
39 c1 = 0.001
40 c2 = 0.5
41 rho = 0.5
42
43 # initial-guess overrides (set to None to use auto estimation)
44 init_a0 = None
45 init_a = None
46 init_tau = None
47 init_omega = None
48 init_alpha = None
49 init_phi = None
50
51
52 # line search options
53 line_search_dict = {
54     "armijo": (
55         fn.armijo_backtracking,
56         dict(alpha_bar = alpha_bar, c1 = c1, rho = rho),
57     ),
58     "strong_wolfe": (
59         fn.strong_wolfe,
60         dict(alpha_bar = alpha_bar, c1 = c1, c2 = c2),
61     ),
62 }
63
64 # choose line search
65 line_search = "strong_wolfe"
66
67 # optional extra overrides from user settings (can be {} if unused)
68 line_search_options_extra = {}
69 line_search_options_extra = line_search_options_extra if 'LS_EXTRA_OPTS'
    in globals() else {}
70
71 # optimizer options
72 optimizer_dict = {
73     "gd": fn.gradient_descent,
74     "cgd": fn.conjugate_gradient_descent,
75     "bfgs": fn.quasi_newton_bfgs,
76 }
77
78 # choose optimizer
79 optimizer = "bfgs"

```

```

80
81
82 # -----
83 # settings for optimization
84 # -----
85
86 # function wrapper
87 def f_wrapper(x):
88     f, _ = objective_function_and_gradient(x, t, v)
89     return float(f)
90
91 # gradient wrapper
92 def g_wrapper(x):
93     _, g = objective_function_and_gradient(x, t, v)
94     return np.asarray(g, dtype=float)
95
96 # line search selection based on user choice
97 ls_func, ls_options = line_search_dict[line_search]
98
99 # optimizer selection based on user choice
100 opt_func = optimizer_dict[optimizer]
101
102 # logger
103 run_tag = "project01"
104
105 # -----
106 # objective function and gradient
107 # -----
108
109 def objective_function_and_gradient(x, tk, v):
110     """
111     compute least-squares objective  $f(x) = \sum((W - v)^2)$ 
112
113     Given function:
114          $W(t) = A_0 + A * \exp(-t / \tau) * \sin((\omega + \alpha * t) * t + \phi)$ 
115
116     parameters:
117         x = [A0, A, tau, omega, alpha, phi]
118
119     inputs:
120         tk = time samples, 1D array
121         v = observed data, 1D array
122
123     returns:
124         f = objective function value

```

```

125         grad = gradient, 1D array of same length as x
126
127     """
128     # parameters
129     a0, a, tau, w, alpha, phi = x
130
131     phase = (w + alpha * tk) * tk + phi
132     E = np.exp(-tk / tau)
133     Sk = E * np.sin(phase)
134     Ck = E * np.cos(phase)
135
136     # model
137     W = a0 + a * Sk
138
139     # residual
140     r = W - v
141
142     # objective function value
143     f = np.sum(r ** 2)
144
145     # gradient
146     grad = 2 * np.array([
147         np.sum(r * 1.0), # df/dAO
148         np.sum(r * Sk), # df/dA
149         np.sum(r * (a * Sk * (tk / tau**2))), # df/dtau
150         np.sum(r * (a * tk * Ck)), # df/domega
151         np.sum(r * (a * (tk**2) * Ck)), # df/dalpha
152         np.sum(r * (a * Ck)) # df/dphi
153     ])
154
155     return f, grad
156
157 # -----
158 # initial guess estimation
159 # -----
160
161 def estimate_initial_guess(t, v):
162     """
163     Initial guesses for [A0, A, tau, omega, alpha, phi] from (t, v).
164
165     Method:
166     - a0 : Mean of last 10% of samples.
167     - a : Max of first 10% minus a0.
168     - tau : Decay from first/last peak of |v - a0|.
169

```



```

170     - omega: Period from first two maxima.
171     - alpha: Slope of freq vs. peak time.
172     - phi : Phase from projections onto decay sin/cos.
173
174     """
175
176     # prepare
177     t = np.asarray(t); v = np.asarray(v)
178     order = np.argsort(t)
179     t_sorted = t[order]
180     v_sorted = v[order]
181
182     # a0 (dc offset)
183     tail_frac = 0.10
184     n_tail = max(1, int(np.ceil(v_sorted.size * tail_frac)))
185     init_a0 = float(np.mean(v_sorted[-n_tail:]))
186
187     # a (amplitude near t=0)
188     head_frac = 0.10
189     n_head = max(1, int(np.ceil(v_sorted.size * head_frac)))
190     init_a = float(np.max(v_sorted[:n_head]) - init_a0)
191     if init_a <= 0:
192         init_a = abs(init_a)
193
194     # residual and simple peak indices
195     y = v_sorted - init_a0
196     a = np.abs(y)
197     # peaks of |y| (for decay)
198     pk_abs = np.where((a[1:-1] > a[:-2]) & (a[1:-1] >= a[2:]))[0] + 1
199     # peaks of y itself (for period)
200     pk_y = np.where((y[1:-1] > y[:-2]) & (y[1:-1] >= y[2:]))[0] + 1
201
202     # tau (decay time)
203     if pk_abs.size >= 2:
204         p1 = max(a[pk_abs[0]], 1e-12)
205         pL = max(a[pk_abs[-1]], 1e-12)
206         ratio = max(p1 / pL, 1 + 1e-6) # ensure log > 0
207         init_tau = float((t_sorted[pk_abs[-1]] - t_sorted[pk_abs[0]]) /
208                          np.log(ratio))
209         dt = max(1e-12, float(np.min(np.diff(t_sorted))))
210         init_tau = max(init_tau, dt)
211     else:
212         init_tau = float(t_sorted[-1] - t_sorted[0])
213
214     # omega (base angular freq)

```

```

214     if pk_y.size >= 2:
215         T = float(t_sorted[pk_y[1]] - t_sorted[pk_y[0]])
216         init_omega = float(2.0 * np.pi / max(T, 1e-12))
217     else:
218         init_omega = float(2.0 * np.pi / max(t_sorted[-1] - t_sorted[0],
219                                             1e-12))
220
221     # alpha (chirp)
222     init_alpha = 0.0
223     if pk_y.size >= 2:
224         peak_times = t_sorted[pk_y]
225         periods = np.diff(peak_times)
226         freqs = 1.0 / np.maximum(periods, 1e-12)
227         freq_times = peak_times[1:]
228
229         if freq_times.size > 1:
230             coeffs = np.polyfit(freq_times, freqs, 1)
231             init_alpha = coeffs[0]
232         else:
233             init_alpha = 0.0
234
235     # phi (phase)
236     t0 = 2.0 * np.pi / max(init_omega, 1e-12)
237     t_end = t_sorted[0] + 2.0 * t0
238     mask = t_sorted <= t_end
239     if not np.any(mask):
240         mask = np.ones_like(t_sorted, dtype = bool)
241
242     tt = t_sorted[mask]
243     yy = y[mask]
244     theta = (init_omega + init_alpha * tt) * tt
245     E = np.exp(-tt / max(init_tau, 1e-12))
246     S = float(np.dot(yy, E * np.sin(theta)))
247     C = float(np.dot(yy, E * np.cos(theta)))
248     init_phi = float(np.arctan2(C, S))
249
250     return np.array([init_a0, init_a, init_tau, init_omega, init_alpha,
251                     init_phi], dtype = float)
252
253
254     # -----
255     # load data
256     # -----

```

```

257
258 # give error if data file not found
259 if not os.path.exists(data_file):
260     raise FileNotFoundError(f"data file not found: {data_file}")
261
262 # read data
263 df = pd.read_csv(data_file, header = None, usecols = [0,1], names =
    ["t", "v"])
264 t = df["t"].to_numpy(dtype = float)
265 v = df["v"].to_numpy(dtype = float)
266
267
268 # -----
269 # run optimization
270 # -----
271
272
273 # initial guess: use overrides if set, else auto
274 auto_a0, auto_a, auto_tau, auto_omega, auto_alpha, auto_phi =
    estimate_initial_guess(t, v)
275
276 a0 = init_a0 if init_a0 is not None else auto_a0
277 a = init_a if init_a is not None else auto_a
278 tau = init_tau if init_tau is not None else auto_tau
279 omega = init_omega if init_omega is not None else auto_omega
280 alpha = init_alpha if init_alpha is not None else auto_alpha
281 phi = init_phi if init_phi is not None else auto_phi
282
283 x0 = np.array([a0, a, tau, omega, alpha, phi], dtype = float)
284
285 # save the initial guess used
286 with open(os.path.join(out_dir,
    f"{optimizer}_{line_search}initial_guess.txt"), "w") as fh:
287     fh.write("x0 = [A0, A, tau, omega, alpha, phi]\n")
288     fh.write(np.array2string(x0, precision=7, separator=", ") + "\n")
289
290 # optimizer options
291 opts = {
292     "max_iter": 1000,
293     "tol": 1e-6,
294     "line_search_opts": ls_options,
295     "save_flag": True,
296     "optimizer": optimizer,
297     "line_search": line_search,
298     "out_dir": out_dir,

```

```

299     "run_tag": run_tag,
300     "alpha0": alpha_bar,
301     "c1": c1,
302     "c2": c2,
303 }
304
305 # run
306 result = opt_func(f_wrapper, g_wrapper, x0, ls_func, opts=opts)
307
308 # print results
309 print("success:", result["success"])
310 print("iters :", result["n_iter"])
311 print("f_final:", result["f"])
312 print("x_final:", result["x"])

```

7.2 Functions

Listing 2: functions.py

```
1
2 # =====
3 # functions.py
4 # -----
5 # purpose:
6 # provide line-search methods and optimizers.
7 #
8 # inputs expected by callers:
9 # - f(x): scalar objective
10 # - grad(x): gradient vector
11 # - x0: initial parameter vector
12 # - line_search: one of {armijo_backtracking, strong_wolfe}
13 # - opts: dict with keys such as:
14 #   max_iter, tol, line_search_opts, save_flag,
15 #   optimizer, line_search, out_dir, run_tag, parameters for line search
16 #
17 # outputs:
18 # each optimizer returns a dict:
19 # {"x", "f", "g", "n_iter", "n_func_eval", "n_grad_eval", "success"}
20 # =====
21
22
23 import numpy as np
24 import pandas as pd
25 from run_logger import RunLogger
26
27
28 # -----
29 # line search methods
30 # -----
31
32 def armijo_backtracking(f, grad, xk, pk, alpha_bar=1.0, c1=1e-3, rho=0.5,
33                        max_backtracks=50, min_alpha=None, **kwargs):
34     """
35     Armijo backtracking line search (simple, safe).
36
37     Finds  $\alpha > 0$  s.t.
38          $f(x_k + \alpha \cdot pk) \leq f(x_k) + c1 \cdot \alpha \cdot \text{grad}(x_k)^T pk$ 
39     by shrinking  $\alpha \leftarrow \rho \cdot \alpha$ .
40
41     Inputs
42         f, grad : callables (f(x)->scalar, grad(x)->vector)
```

```

43     xk : current point
44     pk : descent direction (requires  $\text{grad}(xk)^T pk < 0$ )
45     alpha_bar : initial trial step
46     c1 : Armijo parameter in (0, 1)
47     rho : shrink factor in (0, 1)
48     max_backtracks : cap on shrink steps
49     min_alpha : optional floor for alpha (default: machine eps)
50
51 Returns
52     alpha : accepted step
53     f_new :  $f(xk + \alpha pk)$ 
54     n_eval : number of f-evals during the search (excludes  $f(xk)$ )
55 """
56 # --- basic checks ---
57 if not (0.0 < c1 < 1.0):
58     raise ValueError("c1 must be in (0, 1).")
59 if not (0.0 < rho < 1.0):
60     raise ValueError("rho must be in (0, 1).")
61
62 xk = np.asarray(xk, dtype=float)
63 pk = np.asarray(pk, dtype=float)
64
65 fk = float(f(xk))
66 gk = np.asarray(grad(xk), dtype=float)
67 slope0 = float(np.dot(gk, pk))
68 if not np.isfinite(slope0):
69     raise ValueError("non-finite directional derivative at xk")
70 if slope0 >= 0.0:
71     raise ValueError("pk is not a descent direction ( $\text{grad}^T pk \geq 0$ )")
72
73 alpha = float(alpha_bar)
74 n_eval = 0
75 alpha_floor = np.finfo(float).eps if min_alpha is None else
76     float(min_alpha)
77
78 # --- backtracking loop ---
79 backtrack_count = 0
80 while True:
81     x_trial = xk + alpha * pk
82     f_trial = float(f(x_trial))
83     n_eval += 1
84
85     # if f_trial is not finite, keep shrinking until finite or floor
86     while not np.isfinite(f_trial) and alpha > alpha_floor and
87         backtrack_count < int(max_backtracks):

```

```

86         alpha *= rho
87         backtrack_count += 1
88         x_trial = xk + alpha * pk
89         f_trial = float(f(x_trial))
90         n_eval += 1
91
92         # Armijo sufficient decrease
93         if np.isfinite(f_trial) and f_trial <= fk + c1 * alpha * slope0:
94             return alpha, f_trial, n_eval
95
96         # shrink step
97         alpha *= rho
98         backtrack_count += 1
99
100        # stop if hit floor or backtrack cap
101        if (alpha <= alpha_floor) or (backtrack_count >=
102            int(max_backtracks)):
103            # return best effort at current (possibly shrunk) alpha
104            x_trial = xk + alpha * pk
105            f_trial = float(f(x_trial))
106            n_eval += 1
107            return alpha, f_trial, n_eval
108
109
110
111 def strong_wolfe(f, grad, xk, pk, alpha_bar = 1.0, c1 = 1e-4, c2 = 0.9):
112     """
113     strong wolfe line search
114
115     purpose
116         find a step length  $\alpha > 0$  that satisfies the strong wolfe
117         conditions
118         along a given descent direction  $\mathbf{pk}$  at the current point  $\mathbf{xk}$ 
119
120     conditions enforced
121         armijo sufficient descent:
122              $f(\mathbf{xk} + \alpha \mathbf{pk}) \leq f(\mathbf{xk}) + c1 * \alpha * \text{grad}(\mathbf{xk})^T \mathbf{pk}$ 
123         strong curvature:
124              $\text{abs}(\text{grad}(\mathbf{xk} + \alpha \mathbf{pk})^T \mathbf{pk}) \leq -c2 * \text{grad}(\mathbf{xk})^T \mathbf{pk}$ 
125             with  $0 < c1 < c2 < 1$ 
126
127     inputs
128         f callable,  $f(\mathbf{x}) \rightarrow$  scalar objective value
129         grad callable,  $\text{grad}(\mathbf{x}) \rightarrow$  gradient vector at  $\mathbf{x}$ 

```

```

129     xk current point
130     pk search direction at xk, should satisfy  $\text{grad}(xk)^T pk < 0$ 
131     alpha_bar initial trial step length, default 1.0
132     c1 armijo parameter in (0, 1), default 0.001
133     c2 curvature parameter in (c1, 1), default 0.5
134
135 returns
136     alpha accepted step length satisfying strong wolfe, or a fallback
137     if limits are hit
138     f_new objective value at  $xk + \alpha pk$ 
139     n_eval number of objective evaluations performed inside the line
140     search
141
142 notes
143     this function evaluates both f and grad at trial points
144     only the count of f evaluations is returned as n_eval for
145     consistency with armijo_backtracking
146
147 references
148     Nocedal and Wright, numerical optimization, algorithms 3.5 and 3.6
149
150 """
151
152 # evaluate objective, gradient and search direction at current point
153 xk = np.asarray(xk, dtype = float)
154 pk = np.asarray(pk, dtype = float)
155 fk = float(f(xk))
156 gk = np.asarray(grad(xk), dtype = float)
157
158 # initial directional derivative ( $\text{grad}^T pk$ )
159 slope0 = float(np.dot(gk, pk))
160
161 if not np.isfinite(fk) or not np.all(np.isfinite(gk)): # guard:
162     # directional derivative must be finite
163     raise ValueError("non finite fk or grad(xk)")
164
165 if slope0 >= 0.0: # guard: pk must be a descent direction ( $\text{grad}^T pk$ 
166     < 0)
167     raise ValueError("pk is not a descent direction at xk since
168          $\text{grad}(xk)^T pk \geq 0$ ")
169
170 # helper subfunctions for phi:  $\phi(\alpha) = f(xk + \alpha pk)$ 
171 def phi(alpha):
172     return float(f(xk + alpha * pk))

```



```

167     # helper subfunctions for derivative of phi: dphi(alpha) =
        grad(xk+alpha*pk)^T pk
168 def dphi(alpha):
169     return float(np.dot(grad(xk + alpha * pk), pk))
170
171 n_eval = 0 # to count of objective evaluations performed inside line
        search
172
173 # phase a: bracketing as in algorithm 3.5
174 alpha_prev = 0.0
175 f_prev = fk
176 alpha = float(alpha_bar)
177 i = 0
178
179 while True:
180     f_current = phi(alpha)
181     n_eval += 1
182
183     # check armijo condition, modify bracket and enter zoom phase if
        satisfied
184     if (f_current > fk + c1 * alpha * slope0) or (i > 0 and f_current
        >= f_prev):
185         alpha_lo = alpha_prev
186         alpha_hi = alpha
187         f_lo = f_prev
188         f_hi = f_current
189         break
190
191     # compute directional derivative at current alpha
192     slope_alpha = dphi(alpha)
193
194     # check strong wolfe curvature condition, if satisfied return alpha
195     if abs(slope_alpha) <= c2 * abs(slope0):
196         return alpha, f_current, n_eval
197
198     # check for too large step (slope is nonnegative), modify bracket
        and enter zoom phase if so
199     if slope_alpha >= 0.0:
200         alpha_lo = alpha
201         alpha_hi = alpha_prev
202         f_lo = f_current
203         f_hi = f_prev
204         break
205
206     # otherwise, expand the step and continue the bracketing phase

```

```

207     alpha_prev = alpha
208     f_prev = f_current
209     alpha = 2.0 * alpha # double the step
210     i += 1
211
212 # phase b: zoom as in algorithm 3.6
213 while True:
214
215     # bisection to find a trial alpha in (alpha_lo, alpha_hi)
216     alpha_j = 0.5 * (alpha_lo + alpha_hi)
217
218     # evaluate function at trial alpha_j
219     f_j = phi(alpha_j)
220     n_eval += 1
221
222     # Armijo condition check inside zoom (use fk + c1 * alpha_j * slope0)
223     if (f_j > fk + c1 * alpha_j * slope0) or (f_j >= f_lo):
224         alpha_hi = alpha_j
225         continue
226
227     # evaluate derivative at trial alpha_j
228     slope_j = dphi(alpha_j)
229
230     # check strong wolfe curvature condition inside zoom
231     if abs(slope_j) <= c2 * abs(slope0):
232         return alpha_j, f_j, n_eval
233
234     # sign test to decide which side to continue the search
235     if slope_j * (alpha_hi - alpha_lo) >= 0.0:
236         alpha_hi = alpha_lo
237
238     # move the lower bracket to alpha_j since Armijo passed and curvature failed
239     alpha_lo = alpha_j
240     f_lo = f_j
241
242     # If the loop exits unexpectedly, return the low end of the bracket which satisfies Armijo
243     f_star = phi(alpha_lo)
244     n_eval += 1
245
246     return alpha_lo, f_star, n_eval
247
248

```

```

249
250 # -----
251 # optimization drivers
252 # -----
253
254 def gradient_descent(f, grad, x0, line_search, opts=None):
255     """
256     Gradient descent with line search (simple, safe).
257
258     Returns a dict with: x, f, g, n_iter, n_func_eval, n_grad_eval,
259     success
260     """
261     opts = {} if opts is None else dict(opts)
262
263     # --- user options ---
264     max_iter = int(opts.get('max_iter', 1000))
265     tol = float(opts.get('tol', 1e-6))
266     ls_opts = dict(opts.get('line_search_opts', {})) # will be merged
267     # with meta defaults
268     save_flag = bool(opts.get('save_flag', True))
269
270     optimizer = str(opts.get('optimizer', 'gradient_descent'))
271     ls_name = str(opts.get('line_search', 'armijo_backtracking'))
272     out_dir = str(opts.get('out_dir', './results'))
273     run_tag = str(opts.get('run_tag', 'None'))
274
275     # meta defaults for common line searches
276     meta_alpha0 = float(opts.get('alpha0', 1.0))
277     meta_c1 = float(opts.get('c1', ls_opts.get('c1', 1e-3)))
278     meta_c2 = float(opts.get('c2', ls_opts.get('c2', 0.5))) # used by
279     # strong-wolfe
280
281     # merge meta defaults only if not provided explicitly
282     ls_opts.setdefault('alpha_bar', meta_alpha0)
283     ls_opts.setdefault('c1', meta_c1)
284     # c2 is harmless for Armijo; strong-wolfe will use it
285     ls_opts.setdefault('c2', meta_c2)
286
287     # tiny movement floor (prevents endless micro-steps)
288     step_floor = float(opts.get('step_floor', 1e-16))
289
290     # --- optional logger ---
291     logger = None
292     if save_flag:
293         try:

```

```

291         logger = RunLogger(out_dir=out_dir,
292                             optimizer=optimizer,
293                             line_search=ls_name,
294                             alpha0=meta_alpha0, c1=meta_c1, c2=meta_c2,
295                             max_iter=max_iter, gtol=tol, run_tag=run_tag)
296     except NameError:
297         # if RunLogger is not defined, disable logging quietly
298         logger = None
299         save_flag = False
300
301     # --- initialization ---
302     xk = np.asarray(x0, dtype=float)
303     fk = float(f(xk))
304     gk = np.asarray(grad(xk), dtype=float)
305
306     n_func_eval = 1 # f(x0)
307     n_grad_eval = 1 # grad(x0)
308
309     for k in range(1, max_iter + 1):
310         gk_norm = float(np.linalg.norm(gk, ord=2))
311         if gk_norm <= tol:
312             success = True
313             if logger is not None:
314                 logger.add_eval_counts(f_evals=n_func_eval - 1,
315                                       g_evals=n_grad_eval - 1)
316                 logger.finalize(success=success, n_iter=k - 1, f_final=fk,
317                               g_final=gk_norm, x_final=xk)
317             logger.close()
318             return {"x": xk, "f": fk, "g": gk, "n_iter": k - 1,
319                   "n_func_eval": n_func_eval, "n_grad_eval": n_grad_eval,
320                   "success": success}
321
322     pk = -gk
323     p_norm = float(np.linalg.norm(pk, ord=2))
324
325     # line search (alpha, f_new, n_eval_f)
326     alpha, f_new, n_eval_f = line_search(f, grad, xk, pk, **ls_opts)
327     n_func_eval += int(n_eval_f)
328
329     # guard: non-finite trial or microscopic step
330     if not np.isfinite(f_new) or alpha * p_norm <= step_floor:
331         success = False
332         if logger:
333             logger.add_eval_counts(f_evals=n_eval_f)
334             logger.finalize(success=success, n_iter=k - 1, f_final=fk,

```

```

335         g_final=gk_norm, x_final=xk)
336     logger.close()
337     return {"x": xk, "f": fk, "g": gk, "n_iter": k - 1,
338           "n_func_eval": n_func_eval, "n_grad_eval": n_grad_eval,
339           "success": success}
340
341     # take step
342     xk_next = xk + alpha * pk
343     fk_next = float(f_new)
344     gk_next = np.asarray(grad(xk_next), dtype=float)
345     n_grad_eval += 1
346
347     if logger:
348         df_abs = float(abs(fk - fk_next))
349         logger.add_eval_counts(f_evals=n_eval_f)
350         logger.log_iter(int(k), float(fk_next),
351                        float(np.linalg.norm(gk_next, 2)),
352                        float(alpha), float(p_norm), float(df_abs))
353
354     # prepare next iteration
355     xk, fk, gk = xk_next, fk_next, gk_next
356
357     # max iterations reached
358     success = False
359     if logger:
360         logger.finalize(success=success, n_iter=max_iter, f_final=fk,
361                        g_final=float(np.linalg.norm(gk, 2)), x_final=xk)
362     logger.close()
363
364     return {"x": xk, "f": fk, "g": gk, "n_iter": max_iter,
365           "n_func_eval": n_func_eval, "n_grad_eval": n_grad_eval,
366           "success": success}
367
368
369 def conjugate_gradient_descent(f, grad, x0, line_search, opts = None): #
370     Polak-Ribiere apporach
371     """
372     Polak-Ribiere nonlinear conjugate gradient method with line search
373
374     purpose
375     minimize a smooth unconstrained function f using the nonlinear
376         conjugate
377         gradient method (Polak--Ribiere formula) with a line search. The
378         implementation applies a restart (beta set to zero) when the
379         Polak--Ribiere

```

```

377         coefficient becomes negative (commonly used to maintain descent).
378
379 inputs
380     f callable, f(x) -> scalar objective value
381     grad callable, grad(x) -> gradient vector at x
382     x0 starting point
383     line_search callable, line_search(f, grad, xk, pk, ...) -> (alpha,
384         f_new, n_eval)
385         a line search function that takes f, grad, current
386         point xk and search direction pk
387         and returns a step length alpha > 0 satisfying some
388         conditions along with
389         the new objective value f_new = f(xk + alpha pk) and
390         the number of objective evaluations n_eval
391
392     opts dictionary of all the options for the optimization run,
393     possible keys are
394         max_iter maximum number of iterations
395         tol tolerance on gradient norm for termination
396         line_search_opts dictionary of options for the line
397         search function
398         save_flag whether to save iteration history,
399         default True
400         optimizer name of the optimizer, default
401         'conjugate_gradient_descent'
402         line_search name of the line search, default
403         'armijo_backtracking'
404         out_dir directory to save results, default
405         './results'
406         run_tag tag to identify the run, default 'None'
407         alpha0 initial step length guess for line search,
408         default 1.0
409
410 outputs
411     x best point found
412     f_val objective value at best point
413     k number of iterations performed
414     n_eval total number of objective evaluations performed
415     grad_norm norm of gradient at best point
416     msg termination message
417
418 """
419
420 opts = {} if opts is None else dict(opts)

```

```

411 max_iter = int(opts.get('max_iter', 1000)) # maximum number of
      iterations
412 tol = float(opts.get('tol', 1e-6)) # tolerance on gradient norm for
      termination
413 ls_opts = dict(opts.get('line_search_opts', {})) # options for the
      line search function
414 save_flag = bool(opts.get('save_flag', True)) # whether to save
      iteration history
415
416 optimizer = str(opts.get('optimizer', 'conjugate_gradient_descent'))
      # name of the optimizer
417 ls_name = str(opts.get('line_search', 'armijo_backtracking')) # name
      of the line search
418 out_dir = str(opts.get("out_dir", "./results")) # directory to save
      results
419 run_tag = str(opts.get("run_tag", "None")) # tag to identify the run
420
421 meta_alpha0 = float(opts.get("alpha0", 1.0)) # initial step length
      guess for line search
422 meta_c1 = opts.get("c1", ls_opts.get("c1", 0.001)) # armijo parameter
423 meta_c2 = opts.get("c2", ls_opts.get("c2", 0.5)) # curvature parameter
      for strong wolfe
424
425 logger = None
426 if save_flag:
427     logger = RunLogger(out_dir = out_dir,
428                         optimizer = optimizer,
429                         line_search = ls_name,
430                         alpha0 = meta_alpha0,
431                         c1 = meta_c1,
432                         c2 = meta_c2,
433                         max_iter = max_iter,
434                         gtol = tol,
435                         run_tag = run_tag)
436
437 # initialization
438 xk = np.asarray(x0, dtype = float)
439 fk = float(f(xk))
440 gk = np.asarray(grad(xk), dtype = float)
441
442 n_func_eval = 1 # count f evaluation at initial point
443 n_grad_eval = 1 # count grad evaluation at initial point
444
445 pk = -gk # initial steepest descent direction
446

```

```

447 for k in range(1, max_iter + 1):
448
449     gk_norm = float(np.linalg.norm(gk, ord = 2)) # compute gradient
         norm
450
451     if gk_norm <= tol: # check optimality
452         success = True
453
454         if logger is not None:
455             # exclude initial evaluations from the counts
456             logger.add_eval_counts(f_evals = n_func_eval - 1, g_evals
                                     = n_grad_eval - 1)
457             logger.finalize(success = success, n_iter = k - 1, f_final
                             = fk, g_final = float(np.linalg.norm(gk, 2)), x_final =
                                     xk)
458             logger.close()
459
460         return {"x" : xk,
461                 "f" : fk,
462                 "g" : gk,
463                 "n_iter" : k - 1,
464                 "n_func_eval" : n_func_eval,
465                 "n_grad_eval" : n_grad_eval,
466                 "success" : success
467                 }
468
469     p_norm = float(np.linalg.norm(pk, ord = 2)) # norm of search
         direction
470
471     if np.dot(gk, pk) >= 0:
472         pk = -gk
473
474     alpha, f_new, n_eval_f = line_search(f, grad, xk, pk, **ls_opts) #
         call line search
475     n_func_eval += int(n_eval_f) # update function evaluation count
476
477     xk_next = xk + alpha * pk # new point
478     fk_next = float(f_new) # new objective value
479     gk_next = np.asarray(grad(xk_next), dtype = float) # new gradient
480     n_grad_eval += 1 # update gradient evaluation count
481
482     df_abs = abs(fk - fk_next) # absolute change in objective
483
484     if logger:
485         logger.add_eval_counts(f_evals = n_eval_f)

```



```

486         logger.log_iter(int(k), float(fk_next),
487                          float(np.linalg.norm(gk_next, 2)), float(alpha),
488                          float(p_norm), float(df_abs))
489
490     # compute beta using Polak-Ribiere formula
491     yk = gk_next - gk
492     eps = 1e-12
493     beta = np.dot(gk_next, yk) / max(np.dot(gk, gk), eps) # beta_pr =
494         (g_{k+1}^t (g_{k+1} - g_k)) / (g_k^t g_k)
495     beta = max(beta, 0) # ensure beta is non-negative, (natural
496         restart)
497
498     # # update search direction
499     pk = -gk_next + beta * pk
500
501     # Restart condition with v = 0.2
502     cos_angle = abs(np.dot(gk_next, pk)) / (np.linalg.norm(gk_next) *
503         np.linalg.norm(pk) + 1e-12)
504     if cos_angle > 0.2:
505         pk = -gk_next
506
507     # accept the new point
508     xk = xk_next
509     fk = fk_next
510     gk = gk_next
511
512     # iteration limit reached without convergence
513     success = False
514
515     if logger:
516         logger.finalize(success = success, n_iter = max_iter, f_final =
517             fk, g_final = float(np.linalg.norm(gk, 2)), x_final = xk)
518         logger.close()
519
520     return {"x" : xk,
521            "f" : fk,
522            "g" : gk,
523            "n_iter" : max_iter,
524            "n_func_eval" : n_func_eval,
525            "n_grad_eval" : n_grad_eval,
526            "success" : success
527           }

```

```

525 def quasi_newton_bfgs(f, grad, x0, line_search, opts = None):
526     """
527     quasi-newton method with BFGS update and line search
528     """
529     opts = {} if opts is None else dict(opts)
530     max_iter = int(opts.get('max_iter', 1000)) # maximum number of
        iterations
531     tol = float(opts.get('tol', 1e-6)) # tolerance on gradient norm for
        termination
532     ls_opts = dict(opts.get('line_search_opts', {})) # options for the
        line search function
533     save_flag = bool(opts.get('save_flag', True)) # whether to save
        iteration history
534
535     optimizer = str(opts.get('optimizer', 'quasi_newton_bfgs')) # name of
        the optimizer
536     ls_name = str(opts.get('line_search', 'strong_wolfe')) # name of the
        line search
537     out_dir = str(opts.get("out_dir", "./results")) # directory to save
        results
538     run_tag = str(opts.get("run_tag", "None")) # tag to identify the run
539
540     meta_alpha0 = float(opts.get("alpha0", 1.0)) # initial step length
        guess for line search
541     meta_c1 = opts.get("c1", ls_opts.get("c1", 0.001)) # armijo parameter
542     meta_c2 = opts.get("c2", ls_opts.get("c2", 0.5)) # curvature parameter
        for strong wolfe
543
544     logger = None
545     if save_flag:
546         logger = RunLogger(out_dir = out_dir,
547                             optimizer = optimizer,
548                             line_search = ls_name,
549                             alpha0 = meta_alpha0,
550                             c1 = meta_c1,
551                             c2 = meta_c2,
552                             max_iter = max_iter,
553                             gtol = tol,
554                             run_tag = run_tag)
555
556     # initialization
557     xk = np.asarray(x0, dtype = float)
558     fk = float(f(xk))
559     gk = np.asarray(grad(xk), dtype = float)
560

```

```

561     n_func_eval = 1 # count f evaluation at initial point
562     n_grad_eval = 1 # count grad evaluation at initial point
563
564     n_dim = xk.size
565     Hk = np.eye(n_dim) # initial inverse hessian approximation (identity)
566
567     for k in range(1, max_iter + 1):
568
569         gk_norm = float(np.linalg.norm(gk, ord = 2)) # compute gradient
                    norm
570
571         if gk_norm <= tol: # check optimality
572             success = True
573
574             if logger is not None:
575                 # exclude initial evaluations from the counts
576                 logger.add_eval_counts(f_evals = n_func_eval - 1, g_evals
                    = n_grad_eval - 1)
577                 logger.finalize(success = success, n_iter = k - 1, f_final
                    = fk, g_final = float(np.linalg.norm(gk, 2)), x_final =
                    xk)
578                 logger.close()
579
580             return {"x" : xk,
581                    "f" : fk,
582                    "g" : gk,
583                    "n_iter" : k - 1,
584                    "n_func_eval" : n_func_eval,
585                    "n_grad_eval" : n_grad_eval,
586                    "success" : success
587                    }
588
589         # compute search direction
590         pk = -np.dot(Hk, gk)
591         p_norm = float(np.linalg.norm(pk, ord = 2))
592
593         # call line search
594         alpha, f_new, n_eval_f = line_search(f, grad, xk, pk, **ls_opts)
595         n_func_eval += int(n_eval_f)
596
597         # update step
598         xk_next = xk + alpha * pk
599         fk_next = float(f_new)
600         gk_next = np.asarray(grad(xk_next), dtype = float)
601         n_grad_eval += 1

```

```

602
603     df_abs = abs(fk - fk_next) # absolute change in objective
604     if logger:
605         logger.add_eval_counts(f_evals = n_eval_f)
606         logger.log_iter(int(k), float(fk_next),
607                         float(np.linalg.norm(gk_next, 2)), float(alpha),
608                         float(p_norm), float(df_abs))
609
610     # BFGS update
611     sk = xk_next - xk # step
612     yk = gk_next - gk # gradient difference
613     syk = np.dot(sk, yk) #  $sk^T yk$ 
614
615     if syk > 0.0: # update Hk only if  $sk^T yk$  is sufficiently positive
616         # to maintain positive definiteness
617         rho_k = 1.0 / syk
618         I = np.eye(n_dim)
619         syT = np.outer(sk, yk)
620         ysT = np.outer(yk, sk)
621         ssT = np.outer(sk, sk)
622         v = I - rho_k * syT
623         Hk = np.dot(v, np.dot(Hk, v.T)) + rho_k * ssT # BFGS formula
624
625     # accept the new point
626     xk = xk_next
627     fk = fk_next
628     gk = gk_next
629
630     # iteration limit reached without convergence
631     success = False
632     if logger:
633         logger.finalize(success = success, n_iter = max_iter, f_final =
634                         fk, g_final = float(np.linalg.norm(gk, 2)), x_final = xk)
635     logger.close()
636
637     return {"x" : xk,
638           "f" : fk,
639           "g" : gk,
640           "n_iter" : max_iter,
641           "n_func_eval" : n_func_eval,
642           "n_grad_eval" : n_grad_eval,
643           "success" : success
644         }

```