Math 564 - Project 04

Sajjad Uddin Mahmud

Dec 08, 2025

1 Initial Value

A simple, physically motivated feasible initial guess was constructed by placing the available atoms into stable diatomic molecules:

- All hydrogen atoms in H_2: $x_2 = 2.0$ gives $2 \times 2 = 4$ H atoms.
- All nitrogen atoms in N_2: $x_5 = 1.0$ gives $2 \times 1 = 2$ N atoms.
- All oxygen atoms in O_2: $x_9 = 1.0$ gives $2 \times 1 = 2$ O atoms.

Small positive values are assigned to the remaining species to avoid evaluating the logarithm at zero:

$$x_0 = \big(0.01,\ 2.0,\ 0.01,\ 0.01,\ 1.0,\ 0.01,\ 0.01,\ 0.01,\ 1.0,\ 0.01\big)^\top.$$

2 Log Term and Smoothing

The Gibbs energy contains terms $x_j \ln(x_j/s)$ with $s = \sum_j x_j$. Directly using $\ln(x_j/s)$ is problematic in an infeasible-point method, because iterates may have $x_j \le 0$ or $s \le 0$, where the log is not defined. To handle this gracefully, we keep

$$g_j(x) = x_j \ln(x_j/s) \quad \text{for } x_j \ge \varepsilon,$$

and replace it by a simple quadratic for $x_j < \varepsilon$ that matches the value and slope at $x_j = \varepsilon$. This keeps the objective real-valued and smooth for all iterates, while agreeing with the physical model away from zero.

3 Results

We solve the resulting constrained nonlinear problem with an infeasible-point Sequential Quadratic Programming (SQP) method. At each iteration, we solve a quadratic subproblem with linearized constraints to obtain a search direction, then apply a line search on a merit function that combines the objective and constraint violation.

The optimal mole numbers obtained are:

$$\begin{aligned}
x_1 &= 0.08133617 && (\mathrm{H}),\\
x_2 &= 0.29546069 && (\mathrm{H_2}),\\
x_3 &= 1.56630674 && (\mathrm{H_2O}),\\
x_4 &= 0.00282844 && (\mathrm{N}),\\
x_5 &= 0.97049330 && (\mathrm{N_2}),\\
x_6 &= 0.00138634 && (\mathrm{NH}),\\
x_7 &= 0.05479862 && (\mathrm{NO}),\\
x_8 &= 0.03589456 && (\mathrm{O}),\\
x_9 &= 0.07462873 && (\mathrm{O_2}),\\
x_{10} &= 0.19374263 && (\mathrm{OH}).
\end{aligned}$$

The total number of moles at equilibrium is

$$s = \sum_{j=1}^{10} x_j = 3.27687621.$$

The elemental balance constraints are satisfied to numerical precision:

$$\begin{aligned}
x_1 + 2x_2 + 2x_3 + x_6 + x_{10} - 4 &= 0,\\
x_4 + 2x_5 + x_6 + x_7 - 2 &= 0,\\
x_3 + x_7 + x_8 + 2x_9 + x_{10} - 2 &= 0.
\end{aligned}$$

The minimized Gibbs free energy at the solution is

$$f(x^*) = -95.52218172.$$

4 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.

5 Code

Listing 1: project01_main.py

```python
import numpy as np
import os

# ============================================================================
# STRONG WOLFE LINE SEARCH (FROM PROJECT 01)
# ============================================================================

def strong_wolfe(f, grad, xk, pk, alpha_bar = 1.0, c1 = 1e-4, c2 = 0.9):
    """
    strong wolfe line search
    """
    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(fk) or not np.all(np.isfinite(gk)):
        raise ValueError("non finite fk or grad(xk)")

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

    def phi(alpha):
        return float(f(xk + alpha * pk))

    def dphi(alpha):
        return float(np.dot(grad(xk + alpha * pk), pk))

    n_eval = 0

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

```

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

        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

        slope_alpha = dphi(alpha)

        if abs(slope_alpha) <= c2 * abs(slope0):
            return alpha, f_current, n_eval

        if slope_alpha >= 0.0:
            alpha_lo = alpha
            alpha_hi = alpha_prev
            f_lo = f_current
            f_hi = f_prev
            break

        alpha_prev = alpha
        f_prev = f_current
        alpha = 2.0 * alpha
        i += 1

    # phase b: zoom
    while True:
        alpha_j = 0.5 * (alpha_lo + alpha_hi)
        f_j = phi(alpha_j)
        n_eval += 1

        if (f_j > fk + c1 * alpha_j * slope0) or (f_j >= f_lo):
            alpha_hi = alpha_j
            continue

        slope_j = dphi(alpha_j)

        if abs(slope_j) <= c2 * abs(slope0):
            return alpha_j, f_j, n_eval

        if slope_j * (alpha_hi - alpha_lo) >= 0.0:
```

```
            alpha_hi = alpha_lo

        alpha_lo = alpha_j
        f_lo = f_j

    f_star = phi(alpha_lo)
    n_eval += 1

    return alpha_lo, f_star, n_eval


#
    ================================================================================
# BFGS (FROM PROJECT 01)
#
    ================================================================================

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))
    tol = float(opts.get('tol', 1e-6))
    ls_opts = dict(opts.get('line_search_opts', {}))
    save_flag = bool(opts.get('save_flag', False))

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

    n_func_eval = 1
    n_grad_eval = 1

    n_dim = xk.size
    Hk = np.eye(n_dim)

    for k in range(1, max_iter + 1):
        gk_norm = float(np.linalg.norm(gk, ord = 2))

        if gk_norm <= tol:
            success = True
            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 = -np.dot(Hk, gk)
        p_norm = float(np.linalg.norm(pk, ord = 2))

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

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

        # BFGS update
        sk = xk_next - xk
        yk = gk_next - gk
        syk = np.dot(sk, yk)

        if syk > 0.0:
            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

        xk = xk_next
        fk = fk_next
        gk = gk_next

    success = False
    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
    }
```

```
171
172 #
    ================================================================================
173 # QP SOLVERS (FROM PROJECT 03)
174 #
    ================================================================================
175
176 def solve_equality_qp(G, c, Ae, be):
177     """
178     Solve equality-constrained QP using KKT system
179     min 0.5*x'Gx + c'x subject to Ae*x = be
180     """
181     n = G.shape[0]
182     m = Ae.shape[0] if Ae.size > 0 else 0
183
184     if m == 0:
185         # Unconstrained
186         try:
187             x = np.linalg.solve(G, -c)
188             return x
189         except np.linalg.LinAlgError:
190             x = -np.linalg.pinv(G) @ c
191             return x
192
193     # Build KKT matrix
194     KKT = np.zeros((n + m, n + m))
195     KKT[:n, :n] = G
196     KKT[:n, n:] = Ae.T
197     KKT[n:, :n] = Ae
198
199     rhs = np.concatenate([-c, be])
200
201     try:
202         sol = np.linalg.solve(KKT, rhs)
203         return sol[:n]
204     except np.linalg.LinAlgError:
205         sol, _, _, _ = np.linalg.lstsq(KKT, rhs, rcond=None)
206         return sol[:n]
207
208
209 #
    ================================================================================
210 # UNCONSTRAINED OPTIMIZATION USING BFGS
211 #
    ================================================================================
```

```python

def solve_unconstrained(f, grad_f, x0, line_search, tol=1e-8):
    """
    Solve unconstrained optimization using custom BFGS.
    """
    opts = {
        'max_iter': 1000,
        'tol': tol,
        'save_flag': False
    }

    result = quasi_newton_bfgs(f, grad_f, x0, line_search, opts)
    return result['x'], result['success']


# ============================================================================
# SEQUENTIAL QUADRATIC PROGRAMMING (SQP)
# ============================================================================

def sqp_solver(objective_func, constraint_func, x0, line_search,
               max_iter=100, tol=1e-6, verbose=True, use_qp=True):
    """
    General Sequential Quadratic Programming solver for:
        min f(x)
        s.t. c(x) = 0 (equality constraints)

    Parameters:
    -----------
    objective_func : callable
        Returns (f, grad_f, hess_f)
    constraint_func : callable
        Returns (c, jac_c)
    x0 : array
        Initial guess
    line_search : callable
        Line search function for BFGS
    use_qp : bool
        If True, use QP subproblem; if False, use BFGS for subproblem

    Returns:
    --------
    x : array
        Optimal solution
```

```
    """
    x = np.copy(x0)
    n = len(x)

    # Initialize Lagrange multipliers
    c, jac_c = constraint_func(x)
    m = len(c)
    lam = np.zeros(m)

    for k in range(max_iter):
        # Evaluate objective and constraints
        f, grad_f, hess_f = objective_func(x)
        c, jac_c = constraint_func(x)

        # Check convergence
        grad_L = grad_f - jac_c.T @ lam
        constraint_norm = np.linalg.norm(c)
        gradient_norm = np.linalg.norm(grad_L)

        if verbose and k % 10 == 0:
            print(f"Iter {k}: f={f:.6f}, ||c||={constraint_norm:.2e},
                ||L||={gradient_norm:.2e}")

        if constraint_norm < tol and gradient_norm < tol:
            if verbose:
                print(f"\nConverged in {k} iterations!")
            break

        # Formulate subproblem
        H = hess_f + 1e-6 * np.eye(n) # Regularization

        if use_qp:
            # Solve QP subproblem using your equality-constrained QP solver
            # min 0.5 * p^T * H * p + grad_f^T * p s.t. jac_c * p = -c
            p = solve_equality_qp(H, grad_f, jac_c, -c)
            success = True
        else:
            # Solve via unconstrained BFGS with penalty
            def aug_obj(p):
                return 0.5 * p @ H @ p + grad_f @ p + 1e3 * np.sum((jac_c
                    @ p + c)**2)

            def aug_grad(p):
                return H @ p + grad_f + 2e3 * jac_c.T @ (jac_c @ p + c)

```

```
            p, success = solve_unconstrained(aug_obj, aug_grad,
                np.zeros(n), line_search)

        # Line search on merit function
        alpha = 1.0
        merit_0 = f + 10.0 * constraint_norm

        for _ in range(20):
            x_new = x + alpha * p
            f_new, _, _ = objective_func(x_new)
            c_new, _ = constraint_func(x_new)
            merit_new = f_new + 10.0 * np.linalg.norm(c_new)

            if merit_new < merit_0 - 1e-4 * alpha * np.abs(grad_f @ p):
                break
            alpha *= 0.5

        # Update x
        x = x + alpha * p

        # Update Lagrange multipliers
        if m > 0:
            try:
                lam = np.linalg.lstsq(jac_c.T, grad_f, rcond=None)[0]
            except:
                lam = lam - 0.1 * c

    return x


# ============================================================================
# CHEMICAL EQUILIBRIUM PROBLEM
# ============================================================================

# Problem data
c_values = np.array([
    -6.089, # H
    -17.164, # H2
    -34.054, # H2O
    -5.914, # N
    -24.721, # N2
    -14.986, # NH
    -24.100, # NO
```

```
    -10.708, # O
    -26.662, # O2
    -22.179 # OH
])

# Constraint matrix: A @ x = b
A_constraint = np.array([
    [1, 2, 2, 0, 0, 1, 0, 0, 0, 1], # H
    [0, 0, 0, 1, 2, 1, 1, 0, 0, 0], # N
    [0, 0, 1, 0, 0, 0, 1, 1, 2, 1] # O
])

b_constraint = np.array([4.0, 2.0, 2.0])


def smoothed_log_barrier(x, s, epsilon=1e-3):
    """
    Smoothed logarithmic barrier with C^1 continuity.

    For x >= epsilon: g(x) = x * ln(x/s)
    For x < epsilon: quadratic approximation
    """
    if x >= epsilon:
        return x * np.log(x / s)
    else:
        g_eps = epsilon * np.log(epsilon / s)
        g_prime_eps = np.log(epsilon / s) + 1.0
        delta = x - epsilon
        return g_eps + g_prime_eps * delta + 0.5 * delta**2 / epsilon


def smoothed_log_gradient(x, s, epsilon=1e-3):
    """Gradient of smoothed log barrier."""
    if x >= epsilon:
        return np.log(x / s) + 1.0
    else:
        g_prime_eps = np.log(epsilon / s) + 1.0
        return g_prime_eps + (x - epsilon) / epsilon


def smoothed_log_hessian(x, s, epsilon=1e-3):
    """Hessian of smoothed log barrier."""
    if x >= epsilon:
        return 1.0 / x
    else:
```

```
        return 1.0 / epsilon


def chemical_equilibrium_objective(x, epsilon=1e-3):
    """
    Objective function with smoothed barrier.
    Returns: (f, grad_f, hess_f)
    """
    n = len(x)
    s = np.sum(x)

    # Compute objective
    f = 0.0
    for j in range(n):
        f += c_values[j] * x[j] + smoothed_log_barrier(x[j], s, epsilon)

    # Compute gradient
    grad_f = np.zeros(n)
    for j in range(n):
        term1 = c_values[j]
        term2 = smoothed_log_gradient(x[j], s, epsilon)
        term3 = 0.0
        for k in range(n):
            if x[k] >= epsilon:
                term3 -= x[k] / s
        grad_f[j] = term1 + term2 + term3

    # Compute Hessian
    hess_f = np.zeros((n, n))
    for j in range(n):
        hess_f[j, j] = smoothed_log_hessian(x[j], s, epsilon)

    for j in range(n):
        for k in range(n):
            if x[j] >= epsilon:
                hess_f[j, k] -= x[j] / (s**2)
            if j == k and x[j] >= epsilon:
                hess_f[j, j] += 1.0 / s

    return f, grad_f, hess_f


def chemical_equilibrium_constraints(x):
    """
    Constraint function: A @ x - b = 0
```

```
    Returns: (c, jac_c)
    """
    c = A_constraint @ x - b_constraint
    jac_c = A_constraint
    return c, jac_c


#
    ================================================================================
# MAIN SOLVER
#
    ================================================================================

if __name__ == "__main__":
    # Initial guess
    x0 = np.array([0.01, 2.0, 0.01, 0.01, 1.0, 0.01, 0.01, 0.01, 1.0,
        0.01])
    # x0 = np.ones(10) * 0.5

    print("="*70)
    print("Chemical Equilibrium Problem using SQP")
    print("Using custom BFGS, Strong Wolfe, and QP solvers from Project
        3")
    print("="*70)

    # Wrapper functions for SQP
    def obj_wrapper(x):
        return chemical_equilibrium_objective(x, epsilon=1e-4)

    def con_wrapper(x):
        return chemical_equilibrium_constraints(x)

    # Solve using SQP with QP subproblems and Strong Wolfe line search
    x_opt = sqp_solver(obj_wrapper, con_wrapper, x0,
                       line_search=strong_wolfe,
                       max_iter=100, tol=1e-6, verbose=True,
                       use_qp=True)

    print("\n" + "="*70)
    print("RESULTS")
    print("="*70)
    print("\nOptimal mole quantities:")
    components = ['H', 'H2', 'H2O', 'N', 'N2', 'NH', 'NO', 'O', 'O2',
        'OH']
    for i, comp in enumerate(components):
```

```
        print(f" x[{i+1:2d}] ({comp:4s}) = {x_opt[i]:.8f}")

    print(f"\nTotal moles s = {np.sum(x_opt):.8f}")

    # Verify constraints
    print("\nConstraint satisfaction:")
    c_final, _ = chemical_equilibrium_constraints(x_opt)
    print(f" H balance (x1 + 2x2 + 2x3 + x6 + x10 = 4): error =
        {c_final[0]:.6e}")
    print(f" N balance (x4 + 2x5 + x6 + x7 = 2): error =
        {c_final[1]:.6e}")
    print(f" O balance (x3 + x7 + x8 + 2x9 + x10 = 2): error =
        {c_final[2]:.6e}")

    # Final objective
    f_final, _, _ = chemical_equilibrium_objective(x_opt)
    print(f"\nFinal Gibbs free energy: {f_final:.8f}")

    #
        ============================================================================
    # SAVE RESULTS TO FILE
    #
        ============================================================================

    script_dir = os.path.dirname(os.path.abspath(__file__))
    output_file = os.path.join(script_dir,
        'chemical_equilibrium_results.txt')

    with open(output_file, 'w') as f:
        f.write("="*70 + "\n")
        f.write("Chemical Equilibrium Problem - Results\n")
        f.write("="*70 + "\n\n")

        f.write("Optimal mole quantities:\n")
        f.write("-"*40 + "\n")
        for i, comp in enumerate(components):
            f.write(f" x[{i+1:2d}] ({comp:4s}) = {x_opt[i]:.8f}\n")

        f.write(f"\nTotal moles s = {np.sum(x_opt):.8f}\n")

        f.write("\n" + "="*70 + "\n")
        f.write("Constraint Verification\n")
        f.write("="*70 + "\n")
        f.write(f" H balance (x1 + 2x2 + 2x3 + x6 + x10 = 4): error =
            {c_final[0]:.6e}\n")
```

```
        f.write(f" N balance (x4 + 2x5 + x6 + x7 = 2): error =
            {c_final[1]:.6e}\n")
        f.write(f" O balance (x3 + x7 + x8 + 2x9 + x10 = 2): error =
            {c_final[2]:.6e}\n")

        f.write("\n" + "="*70 + "\n")
        f.write("Objective Function\n")
        f.write("="*70 + "\n")
        f.write(f" Final Gibbs free energy f(x) = {f_final:.8f}\n")

        f.write("\n" + "="*70 + "\n")
        f.write("Problem Parameters\n")
        f.write("="*70 + "\n")
        f.write(f" Temperature: T = 3500 K\n")
        f.write(f" Pressure: P = 750 psi\n")
        f.write(f" Epsilon (smoothing parameter): {1e-4}\n")

        f.write("\n" + "="*70 + "\n")
        f.write("Free Energy Coefficients (cj)\n")
        f.write("="*70 + "\n")
        for i, comp in enumerate(components):
            f.write(f" c[{i+1:2d}] ({comp:4s}) = {c_values[i]:8.3f}\n")

    print(f"\nResults saved to '{output_file}'")
```