

Appendix B

R and Python Code Library

This appendix collects, in one place, the reusable R and Python functions developed across the chapters, for convenient reference and reuse in coursework or research.

B.1: Fieller's Theorem (Chapter 3)

R:

```
fieller_ci <-function(a, b, s2, v11, v22,
v12, df, conf =0.95) {
  t0 <-qt(1- (1-conf)/2, df)
  g <- t0^2* s2 * v22 / b^2
  m <- a / b
  centre <- m - g * v12 / v22
  root <-sqrt(v11 -2*m*v12 + m^2*v22 - g*(v11
- v12^2/v22))
  lower <- (centre - t0*sqrt(s2)/b * root) /
(1- g)
  upper <- (centre + t0*sqrt(s2)/b * root) /
(1- g)
  list(g = g, m = m, lower = lower, upper =
upper)
}
```

Python:

```

import numpy as np
from scipy import stats

def fieller_ci(a, b, s2, v11, v22, v12, df,
conf=0.95):
    t0 = stats.t.ppf(1- (1-conf)/2, df)
    g = t0**2* s2 * v22 / b**2
    m = a / b
    centre = m - g * v12 / v22
    root = np.sqrt(v11 - 2*m*v12 + m**2*v22 -
g*(v11 - v12**2/v22))
    lower = (centre - t0*np.sqrt(s2)/b *
root) / (1- g)
    upper = (centre + t0*np.sqrt(s2)/b *
root) / (1- g)
    return {"g": g, "m": m, "lower": lower,
"upper": upper}

```

B.2: Direct-Assay Potency Estimate (Chapter 4)**R:**

```

direct_assay_potency <-function(standard,
test) {
    nS <-length(standard); nT <-length(test)
    xS <-mean(standard); xT <-mean(test)
    SSs <-sum((standard-xS)^2); SSt <-
sum((test-xT)^2)
    s2 <- (SSs+SSt)/((nS-1)+(nT-1))
    R <- xS/xT
    ci <-fieller_ci(xS, xT, s2, 1/nS, 1/nT, 0,
(nS-1)+(nT-1))
    list(R = R, s2 = s2, df = (nS-1)+(nT-1), ci =
ci)
}

```

B.3: Behrens–Fisher–Sukhatme / Welch Adjustment (Chapter 5)

R:

```
welch_df <-function(s1sq, n1, s2sq, n2) {
  num <- (s1sq/n1 + s2sq/n2)^2
  den <- (s1sq/n1)^2/(n1-1) +
(s2sq/n2)^2/(n2-1)
  num/den
}

fieller_bfs <-function(a, b, var_a, var_b,
cov_ab, nu, conf =0.95) {
  t0 <-qt(1- (1-conf)/2, nu)
  g <- t0^2* var_b / b^2
  m <- a / b
  centre <- m - g * cov_ab / var_b
  root <-sqrt(var_a -2*m*cov_ab + m^2*var_b -
g*(var_a - cov_ab^2/var_b))
  lower <- (centre - t0*root) / (1- g)
  upper <- (centre + t0*root) / (1- g)
  list(g = g, m = m, lower = lower, upper =
upper, df = nu)
}
```

B.4: Parallel-Line Contrast Coefficients (Chapter 6)

Python:

```
import numpy as np

def parallel_line_contrasts(m):
  theta = m*(m**2-1)
  w = np.arange(1, m+1) - (m+1)/2
```

```

    prep = np.concatenate([np.ones(m), -
np.ones(m)])
    comb_reg = np.concatenate([w, w])
    parallel = np.concatenate([w, -w])
  return {"preparation": prep,
"combined_regression": comb_reg,
"parallelism": parallel, "theta": theta}

```

B.5: Parallel-Line Potency From a Fitted Model (Chapter 6)

R:

```

parallel_line_potency <-function(fit,
prep_coef_name, logdose_coef_name) {
  b_coef <-coef(fit)
  alpha_diff <- b_coef[prep_coef_name]
  beta      <- b_coef[logdose_coef_name]
  rho_hat <-exp(-alpha_diff/beta)
  list(rho = rho_hat, alpha_diff = alpha_diff,
beta = beta)
}

```

B.6: Slope-Ratio Potency (Chapter 7)

R:

```

slope_ratio_potency <-function(fit,
betaS_name, betaT_name, lambda =1) {
  b <-coef(fit)
  rho <-
(b[betaT_name]/b[betaS_name])^(1/lambda)
  rho
}

```

B.7: Probit/Logit ED50 and Relative Potency (Chapter 8)

R:

```
ed50_from_glm <-function(fit)
{a <--coef(fit)[1]; b <-coef(fit)[2]
  V <-vcov(fit)
fieller_ci(a, b, s2 =1, v11 = V[1,1], v22 =
V[2,2], v12 =-V[1,2], df =Inf)}
```

B.8: Bartlett's Test (Chapter 9) — Manual
Implementation

Python:

```
import numpy as np

def bartlett_manual(groups):
    k =len(groups)
    n = np.array([len(g) for g in groups])
    s2 = np.array([np.var(g, ddof=1) for g in
groups])
    N = n.sum()
    pooled = np.sum((n-1)*s2) / (N-k)
    M = (N-k)*np.log(pooled) - np.sum((n-
1)*np.log(s2))
    C =1+ (1/(3*(k-1))) * (np.sum(1/(n-1)) -
1/(N-k))
    chi2_stat = M / C
    return {"statistic": chi2_stat, "df": k-1,
"pooled_var": pooled}
```

B.9: Validity-Report Pipeline for Parallel-Line Assays (Chapter 9)

Python:

```
import numpy as np, statsmodels.api as sm
from scipy import stats

def parallel_line_validity_report(df,
response_col, logdose_col, prep_col):
    df = df.copy()
    df['cell'] = df[prep_col].astype(str)
    "+"_"+ df[logdose_col].astype(str)
    full_model = sm.OLS.from_formula(
f"{response_col} ~
C({prep_col})*C({logdose_col})", df).fit()
    lin_model = sm.OLS.from_formula(
f"{response_col} ~
C({prep_col})*{logdose_col}", df).fit()
    lof = sm.stats.anova_lm(lin_model,
full_model)
    groups = [g[response_col].values for _, g
in df.groupby('cell')]
    bart_stat, bart_p =
stats.bartlett(*groups)
    return {
"lack_of_fit_p": lof['Pr(>F)'][1]
if 'Pr(>F)' in lof else None,
"parallelism_p": lin_model.pvalues.get(
f"C({prep_col})[T.T]:{logdose_col}", np.nan),
"bartlett_p": bart_p,
"valid": bart_p > 0.05}
```

B.10: 2×2 Cross-Over Design Analysis (Chapter 11)**Python:**

```

import numpy as np

def crossover_2x2(g1_p1, g1_p2, g2_p1,
g2_p2):
    g1_p1, g1_p2, g2_p1, g2_p2
=map(np.asarray, (g1_p1, g1_p2, g2_p1,
g2_p2))
    n1, n2 =len(g1_p1), len(g2_p1)
    T1, T2 = g1_p1+g1_p2, g2_p1+g2_p2
    d1, d2 = g1_p1-g1_p2, g2_p1-g2_p2

    sT2 = ((n1-1)*T1.var(ddof=1) + (n2-
1)*T2.var(ddof=1)) / (n1+n2-2)
    t_co = (T1.mean()-T2.mean()) /
np.sqrt(sT2*(1/n1+1/n2))

    sd2 = ((n1-1)*d1.var(ddof=1) + (n2-
1)*d2.var(ddof=1)) / (n1+n2-2)
    t_trt = ((d1.mean()+d2.mean())/2) /
np.sqrt(sd2/4*(1/n1+1/n2))
    t_period = ((d1.mean()-d2.mean())/2) /
np.sqrt(sd2/4*(1/n1+1/n2))

    df = n1+n2-2
return {"t_carryover": t_co, "t_treatment":
t_trt, "t_period": t_period,
"df": df, "F_carryover": t_co**2,
"F_treatment": t_trt**2,
"F_period": t_period**2}

```

B.11: Notes on Reuse

All functions above are written for clarity, not maximum numerical robustness; students building a production analysis pipeline should add input validation (e.g., checking $g < 1$ before reporting Fieller limits, as discussed in Chapter 3) and consider vectorizing across many simultaneous assays where relevant. Where R and Python versions of a function are given, they are algebraically identical and will return the same numerical results on the same input data, differing only in language-specific syntax.