TRABAJO FINAL PARA ECONOMETRIA

FELIPE I. TAPPATA

RESUMEN. Este documento contiene la solución al trabajo final de la materia *Econometría* en la Universidad Torcuato Di Tella, tercer trimestre de 2025. El código usado fue entregado junto a este documento, y se puede encontrar también en el repositorio `https://github.com/felipetappata/metrics-final`. Las consignas originales (con pequeñas modificaciones que no alteran el significado) han sido reproducidas dentro de cuadros intercalados con las soluciones para facilitar la lectura. El principal software usado es Julia, y LaTeX para la compilación de gráficos y tablas y del documento final. El archivo `README.md` en la raíz del repositorio contiene instrucciones para la replicación de los resultados. En el apéndice se encuentran reproducidos los *scripts* `ej1.jl`, `ej2.jl` y `ej3.jl`, que contienen el código necesario para replicar los ejercicios 1, 2.1 y 2.2 respectivamente.

1. Propiedades de muestra finita de FGLS (MCGE)

Ejercicio 1. Considere el modelo

$$y_i = \beta_0 + \beta_1 x_i + u_i, \quad i = 1, 2, \ldots, 5N, \tag{1}$$

con $\beta_0 = -3$, $\beta_1 = 0.8$, $u_i \sim N(0, \Omega \otimes I_{N\times N})$, $\Omega = \text{diag}(4, 9, 16, 25, 36)$ y $x_i \sim U(1, 50)$.

1. Genere 5 000 muestras de $5N = 5$ observaciones de corte transversal a partir del modelo (1).
   a) Para cada muestra estime por FGLS los parámetros del modelo y realice un test de hipótesis para contrastar que $H_0 : \beta_1 = 0.8$. Reporte tamaño del test al 1 % y 5 % y el poder del test cuando $\beta_1 = 0$ y $\beta_1 = 0.4$. Adicionalmente, reporte la media, mediana y desvío estándar de las estimaciones de β_0 y β_1.
   b) Utilice la descomposición de Cholesky para encontrar una matriz P que cumpla que se puede escribir a $\Omega = PP'$, aplique la transformación correspondiente a MCG al modelo (1) y estime el modelo por MCC y comente si cambia algún resultado.
2. Repita el punto anterior (solo el inciso a)) con $5N = 10$.
3. Repita el punto anterior (solo el inciso a)) con $5N = 30$.
4. Repita el punto anterior (solo el inciso a)) con $5N = 100$.
5. Repita el punto anterior (solo el inciso a)) con $5N = 200$.
6. Repita el punto anterior (solo el inciso a)) con $5N = 500$.
7. Describa detalladamente las propiedades de muestra finita de FGLS de acuerdo a lo que observó de los cuatro puntos anteriores. En especial, explique cómo cambia el tamaño y la potencia de los tests a medida que aumenta el tamaño de muestra.

1.1. El Modelo.

Podemos escribir el modelo en forma matricial como

$$\boldsymbol{Y} = \boldsymbol{X}\beta + \boldsymbol{u}, \tag{2}$$

Fecha: 23 de febrero de 2026

donde $\boldsymbol{Y} = (y_1, y_2, \ldots, y_{5N})'$ es el vector de observaciones de la variable dependiente, $\boldsymbol{X}$ es la matriz de regresores

$$\boldsymbol{X} = \begin{bmatrix} X_1' \\ X_2' \\ \vdots \\ X_{5N}' \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_{5N} \end{bmatrix},$$

el vector de parámetros es $\beta = (\beta_0, \beta_1)'$ y el vector de errores es $\boldsymbol{u} = (u_1, u_2, \ldots, u_{5N})'$. Tenemos observaciones i.i.d. de un regresor $x \sim U(1, 50)$ y, condicional en la matriz de datos los errores son independientes y normales con media cero, pero heterocedásticos:

$$\boldsymbol{u}|\boldsymbol{X} \sim N(0, \Omega \otimes I_{N\times N}).$$

Tenemos entonces

$$E[\boldsymbol{u} \mid \boldsymbol{X}] = 0,$$
$$\text{var}[\boldsymbol{u} \mid \boldsymbol{X}] = E[\boldsymbol{u}\boldsymbol{u}' \mid \boldsymbol{X}] = \Sigma := \Omega \otimes I_{N\times N}$$

1.2. Mínimos Cuadrados Generalizados. Cuando Σ es conocida, el estimador de Aitken (1935),

$$\widetilde{\beta}_{\text{gls}} = (\boldsymbol{X}'\Sigma^{-1}\boldsymbol{X})^{-1}\boldsymbol{X}'\Sigma^{-1}\boldsymbol{Y}, \tag{3}$$

es insesgado para β y es eficiente en la clase de estimadores insesgados (Hansen, 2022, pág. 108). Este estimador, $\widetilde{\beta}_{\text{gls}}$, es el de *mínimos cuadrados generalizados* (GLS o MCG). El estimador GLS se puede entender como el resultado de estimar por mínimos cuadrados clásicos (OLS) a un modelo transformado, obtenido premultiplicando ambos lados del modelo (2) por $\Sigma^{-1/2}$,[1] obteniendo

$$\widetilde{\boldsymbol{Y}} = \widetilde{\boldsymbol{X}}\beta + \widetilde{\boldsymbol{u}}, \tag{4}$$

donde $\widetilde{\boldsymbol{Y}} = \Sigma^{-1/2}\boldsymbol{Y}$, $\widetilde{\boldsymbol{X}} = \Sigma^{-1/2}\boldsymbol{X}$ y $\widetilde{\boldsymbol{u}} = \Sigma^{-1/2}\boldsymbol{u}$. El estimador OLS es

$$\begin{aligned}\widehat{\beta} &= \left(\widetilde{\boldsymbol{X}}'\widetilde{\boldsymbol{X}}\right)^{-1}\widetilde{\boldsymbol{X}}'\widetilde{\boldsymbol{Y}} \\ &= \left((\Sigma^{-1/2}\boldsymbol{X})'(\Sigma^{-1/2}\boldsymbol{X})\right)^{-1}(\Sigma^{-1/2}\boldsymbol{X})'(\Sigma^{-1/2}\boldsymbol{Y}) \\ &= (\boldsymbol{X}'\Sigma^{-1}\boldsymbol{X})^{-1}\boldsymbol{X}'\Sigma^{-1}\boldsymbol{Y}.\end{aligned}$$

Esta transformación es útil. Por un lado, nos permite establecer que bajo el modelo normal que hemos supuesto, el estimador GLS es el estimador de máxima verosimilitud (MLE) para β.[2] Más relevante para el ejercicio, nos permite establecer que la matriz mencionada por el inciso b) puede ser, tomando raíces cuadradas de los elementos de la diagonal de Ω, $P = \text{diag}(2,\ 3,\ 4,\ 5,\ 6)$, y por lo tanto

$$\Sigma^{-1/2} = \text{diag}(2, 3, 4, 5, 6) \otimes I_{N\times N}$$

Luego, la estimación "directa" de GLS es algebráicamente equivalente a la estimación de OLS del modelo transformado.[3]

[1] La matriz $\Sigma^{-1/2}$ existe y es única para Σ positiva definida.

[2] El estimador OLS es algebráicamente equivalente al estimador de maxima verosimilitud en el modelo normal heterocedástico (Hansen, 2022, pág. 144). El error de nuestro modelo transformado se distribuye como $\widetilde{\boldsymbol{u}} \sim N(0, I)$, y por lo tanto el modelo transformado es un modelo normal homocedástico y vale el resultado. También se puede derivar fácilmente de manera directa minimizando $\ell(\beta) = -(1/2)\big((\boldsymbol{Y}-\boldsymbol{X}\beta)'\Sigma^{-1}(\boldsymbol{Y}-\boldsymbol{X}\beta)\big)$.

[3] Usaremos esto para responder el ejercicio b); las estimaciones del modelo transformado serán mejores que las del modelo original, porque estaremos comparando GLS con FGLS.

Notemos, por último, que en este modelo el estimador de GLS es también el de mínimos cuadrados ponderados (WLS), porque Σ es diagonal. Es decir, $\widetilde{\beta}_{\text{gls}}$ es el argumento minimizador de

$$\min_b \quad \sum_{i=1}^{5N} \sigma_i^{-2}(y_i - X_i'b)^2,$$

donde σ_i denota la varianza del error de la observación i.

En general, Σ es desconocida y por lo tanto la expresión (3) no constituye un estimador (no es función de la muestra), y decimos que no es estimable (es *infeasible*). Un estimador que reemplaza Σ en la ecuación (3) por una estimación consistente $\widehat{\Sigma}$ se denomina un estimador de *mínimos cuadrados generalizados estimables* (FGLS o MCGE). Es decir, si $\widehat{\Sigma}$ es un estimador consistente para Σ, definimos el estimador de FGLS como

$$\widetilde{\beta}_{\text{fgls}} = (\boldsymbol{X}'\widehat{\Sigma}^{-1}\boldsymbol{X})^{-1}\boldsymbol{X}'\widehat{\Sigma}^{-1}\boldsymbol{Y}.$$

Los estimadores FGLS son asintóticamente equivalentes entre sí, y asintóticamente equivalentes al estimador de GLS. El estimador FGLS se puede entender también como OLS ponderado, como en la discusión previa. Mientras que GLS pondera a cada residuo por la inversa de la varianza de su correspondiente error, FGLS pondera por una estimación de dicha varianza:

$$\widetilde{\beta}_{\text{fgls}} = \arg\min_\beta \sum_{i=1}^{5N} \widehat{\sigma}_i^{-2}(y_i - X_i'\beta)^2,$$

1.3. Estimación de varianzas del error. Desde el punto de vista de estimación, asumiremos que la estructura del modelo es considerada correcta, o conocida, con parámetros β y Ω desconocidos. Esto quiere decir que, si bien hay heterocedasticidad y no se conocen los elementos de la matriz Σ, se conoce que Σ tiene la estructura $\Sigma = \Omega \otimes I_{N\times N}$, con Ω diagonal.[4] Para estimar Σ, nos alcanza con estimar los cinco parámetros σ_1^2, σ_2^2, σ_3^2, σ_4^2, σ_5^2 que definen la matriz Ω. Podemos pensar a la muestra como 5 muestras de N observaciones independientes con errores heterocedásticos, a fines de estimar cada una de las varianzas σ_j^2. Es decir, si bien todas las observaciones "llevan información" sobre los coeficientes β, cada bloque de N observaciones "lleva información" sobre una sola de las varianzas σ_j^2.[5]

Los indices originales de la muestra, $i \in \{1, 2, \ldots, 5N\}$, se pueden renombrar con un par (j, k), donde $j \in \{1, 2, 3, 4, 5\}$ indica a qué bloque de N observaciones corresponde la observación i, y $k \in \{1, 2, \ldots, N\}$ indica el número de la observación dentro del bloque de manera que $i = (j-1)N + k$. Luego, dentro de cada bloque j,

$$Y_{jk} = X_{jk}'\beta + u_{jk}, \quad u_{jk} \overset{\text{iid}}{\sim} N(0, \sigma_j^2),$$

lo cual sugiere la estimación de cada σ_j^2 por el estimador de varianza de los residuos de OLS en el bloque j:

$$\widehat{\sigma}_j^2 = \frac{1}{N}\sum_{k=1}^{N} \widehat{u}_{jk}^2,$$

[4]Esto es quizá ambiguo en la consigna original, pero si no asumimos nada sobre la estructura de Σ, salvo diagonalidad, entonces tenemos el problema de estimar $5N$ parámetros con solo $5N$ observaciones.

[5]A nivel intuitivo esto es claro, pero lo podemos formalizar más rigurosamente factorizando la función de verosimilitud conjunta de la muestra completa.

donde $\widehat{u}_{jk} = Y_{jk} - X'_{jk}\widehat{\beta}$ es el residuo de OLS para la observación k del bloque j.[6]

1.4. Test de hipótesis. Para contrastar la hipótesis nula $H_0 : \beta_1 = 0.8$, podemos usar el estadístico T definido como

$$T := \frac{\widetilde{\beta}_{\text{fgls},1} - 0.8}{\sqrt{\widehat{\text{var}}(\widetilde{\beta}_{\text{fgls},1})}}, \tag{5}$$

donde $\widetilde{\beta}_{\text{fgls},1}$ es la componente de $\widetilde{\beta}_{\text{fgls}}$ correspondiente a β_1, y $\widehat{\text{var}}(\widetilde{\beta}_{\text{fgls},1})$ es la estimación de la varianza de esta componente, que se obtiene como el elemento diagonal correspondiente a β_1 de la matriz $(\boldsymbol{X}'\widehat{\Sigma}^{-1}\boldsymbol{X})^{-1}$. Bajo la hipótesis nula, la distribución asintótica de T es normal estándar:

$$T \xrightarrow{d} N(0,1).$$

Por lo tanto, nos referiremos con *test de nivel o tamaño* α al test a dos colas que tiene nivel asintótico α; más precisamente, región de rechazo

$$\mathcal{R} = \{T : |T| > z_{1-\alpha/2}\},$$

donde $z_{1-\alpha/2}$ es el cuantil $1 - \alpha/2$ de la distribución normal estándar.[7]

1.5. Experimento Monte Carlo. El experimento de Monte Carlo consiste en generar $B = 5\,000$ muestras de $5N$ observaciones a partir del modelo (2) con los parámetros y la estructura de varianza especificados, y para cada muestra estimar por FGLS los parámetros del modelo y realizar un test de hipótesis para contrastar que $H_0 : \beta_1 = 0.8$. En particular, realizamos un $B = 5\,000$ simulaciones para cada combinación de tamaño de muestra $5N \in \{5, 10, 30, 100, 200, 500\}$ y valor generador de datos $\beta_1 \in \{0, 0.4, 0.8\}$. Los gráficos se hacen en el loop, para no tener que guardar resultados en la memoria, pero los estadísticos de resumen (tasa de rechazo, media, mediana y desvío estándar de las estimaciones de β_0 y β_1) se guardan.

Para el inciso b), realizamos los mismos experimentos, pero la "estimación" por FGLS es en realidad OLS en el modelo transformado (4), i.e., GLS.

1.6. Implementaciones. El código para la resolución de este ejercicio se encuentra en el script `scripts/ej1.jl`. A seguir, una breve descripción de las funciones principales usadas.

- `draw_sample` genera una muestra de $5N$ observaciones a partir del modelo (2) con los parámetros y la estructura de varianza especificados. En la práctica, usamos la función `draw_sample!`, que funciona de la misma manera, pero muta sus argumentos *in-place* a fines de optimizar la ejecución de los experimentos de Monte Carlo.[8]

[6] El vector $\widehat{\beta}$ es el estimador OLS usando todas las $5N$ observaciones, no solo las del bloque j. Una posible mejora sería iterar el proceso, recalculando $\widehat{\beta}$ por FGLS usando las estimaciones de varianza obtenidas en la iteración previa, y luego recalculando las varianzas usando los residuos de esta nueva estimación de β, y así sucesivamente hasta convergencia. Otra mejora posible sería hacer un ajuste por muestra finita, teniendo en cuenta el *leverage* de un bloque sobre la estimación de $\widehat{\beta}$.

[7] Es decir, z_a es el valor tal que $\mathbb{P}[Z \leq z_a] = a$ para $Z \sim N(0,1)$. Para 1 % y 5 %, usamos valores críticos de 2.576 y 1.96, respectivamente.

[8] El signo de exclamación '!' es una convención en Julia para indicar que la función muta sus argumentos. Otras funciones que usan esta notación son la que fija el *seed*, `Random.seed!(8869)` y la implementación del test de White en `scripts/ej2.jl`.

- `ols_2reg` estima por OLS los coeficientes de un modelo de regresión lineal simple (constante y pendiente) de manera muy eficiente, funcionando mediante sumas escalares en lugar de operaciones matriciales.
- `omega_hat_ols` estima la la matriz Ω usando el procedimiento descrito en la sección 1.3.
- `gls_given_omega` computa los coeficientes de GLS/FGLS según la ecuación (3) dado un valor para Ω. También devuelve los elementos diagonales de $(\boldsymbol{X}'\Sigma^{-1}\boldsymbol{X})^{-1}$, dado Ω, para calcular errores estándar para inferencia. Obviamente Ω puede ser la verdadera matriz o una estimación de ella.
- `fgls` compone las dos funciones anteriores para estimar por FGLS, dado una muestra. También devuelve errores estándar para los parámetros.
- `t_test` construye el estadístico T de la ecuación (5).
- `run_exper` ejecuta un experimento de Monte Carlo. Esto es, dado la cantidad de simulaciones B, el tamaño de muestra $5N$, y los valores "verdaderos" (generadores de datos) de β_0 y β_1 (entre otros argumentos), genera B muestras a partir del modelo (2) con los parámetros y la estructura de varianza especificados, y para cada muestra estima por FGLS los parámetros del modelo y realiza un test de hipótesis para contrastar que $H_0 : \beta_1 = 0.8$.
- `run_monte_carlo` ejecuta `run_exper` para cada combinación de argumentos que requerimos para responder la consigna.

Hay otras funciones relacionadas con la creación y tablas que no son mencionadas en esta sección.

1.7. Resultados. El cuadro 1 muestra las tasas de rechazo empíricas (i.e., la frecuencia relativa de tests rechazados en las $B = 5\,000$ simulaciones) para el test de hipótesis propuesto para cada tamaño de muestra, y para valores generadores de datos $\beta_1 = 0$, $\beta_1 = 0.4$ y $\beta_1 = 0.8$. Las primeras cuatro columnas muestran la frecuencia relativa de rechazo de la

Cuadro 1. Tasas de rechazo empíricas

	$\beta_1 = 0$		$\beta_1 = 0.4$		$\beta_1 = 0.8$	
$5N$	1 %	5 %	1 %	5 %	1 %	5 %
5	98.4 %	99.08 %	86.4 %	90.92 %	37.48 %	47.44 %
10	99.96	99.98	96.56	98.24	20.68	31.76
30	100	100	100	100	5.28	12.7
100	100	100	100	100	1.7	7.26
200	100	100	100	100	1.36	5.58
500	100	100	100	100	1.24	5.5

hipótesis nula cuando el valor generador de datos no es el de la hipótesis nula: una estimación de la probabilidad de rechazar correctamente. Incluso en muestras pequeñas, el test exhibe potencia muy elevada en los dos parámetros presentados. Sin saber el contenido económico del parámetro β_1, es difícil decir si esto es necesariamente bueno—puede ser el caso que por las unidades del modelo, la brecha entre $\beta_1 = 0.4$ y $\beta_1 = 0.8$ sea muy grande, y por lo tanto que no sea difícil distinguir entre ambos valores. Es decir, se valora la capacidad de un test para distinguir entre valores "cercanos" a la hipótesis nula, pero sin un contexto económico es difícil decir si 0.4 es un valor cercano a 0.8 o no. De todos modos, podríamos

decir tentativamente que la potencia del test es buena incluso en muestras pequeñas, al menos para los parámetros probados. Asintóticamente, la probabilidad de rechazar la hipótesis nula cuando el valor generador de datos no es el de la hipótesis nula es igual a uno, y podríamos decir que los valores exhibidos (e.g. 99.4 %, 98.24 %) son razonablemente cercanos a este valor asintótico, incluso para muestras pequeñas.

La última columna de la tabla 1, correspondiente a $\beta_1 = 0.8$, presenta la frecuencia relativa de rechazo de la hipótesis nula cuando el valor generador de datos coincide con el de hipótesis nula: una estimación de la probabilidad de rechazar incorrectamente. A diferencia con la potencia del test, aquí hay una clara discrepancia entre el valor asintótico y los valores empíricos en muestras de tamaño menor a $5N = 200$. Para $5N = 5$, el tamaño empírico (esto es, frecuencia relativa de rechazos bajo el proceso generador de datos de la hipótesis nula) para el test de tamaño nominal de 1 % es de 37.48 %, lo cual rinde inservible el test en este caso. Para el test de tamaño 5 %, también hay un tamaño empírico un orden de magnitud mayor al nominal. A medida que aumenta el tamaño de muestra, el tamaño empírico se acerca al nominal, pero incluso para $5N = 100$ la probabilidad (empírica) de rechazar incorrectamente la hipótesis nula es más de un 50 % mayor a la probabilidad nominal.

La diferencia del tamaño con su valor asintótico, y la convergencia a medida que crece el tamaño muestral, se puede apreciar mirando la figura 2, que muestra la distribución empírica del estadístico T bajo $H_0 : \beta_1 = 0.8$ para cada tamaño de muestra. En particular, los gráficos de esta figura muestran la distribución asintótica de T bajo H_0 (normal estándar, línea punteada) y un *kernel density estimate* (KDE, línea sólida), que es una estimación no paramétrica de la función de densidad de T bajo H_0.[9] Las áreas sombreadas en cada gráfico corresponden a la frecuencia relativa de rechazos empírica, es decir, el área debajo de la densidad empírica de T que cae en la región de rechazo del test. El valor de estas áreas se encuentran arriba y a la izquierda de cada gráfico, y corresponden a la anteúltima columna de la tabla 1. Vemos que para los tamaños más pequeños de $5N$, la distribución de T posee colas más pesadas que la distribución asintótica, lo cual sirve para explicar el tamaño empírico tan elevado. A medida que crece $5N$, la distribución empírica de T se acerca a la normal estándar, y por lo tanto el tamaño empírico se acerca al nominal.

La relación entre el valor de β_1 generador de datos, el tamaño de muestra y la tasa de rechazo empírica se puede ver de manera más completa en la figura 1. Cada punto de la figura corresponde a una combinación $(5N, \beta_1)$, y la intensidad de su color representa la tasa rechazo empírica de un test de tamaño nominal 5 % resultante de un experimento de Monte Carlo con $B = 10\,000$ replicaciones para esa combinación. Los puntos más oscuros corresponden a tasas de rechazo más bajas, y los puntos más claros a tasas de rechazo más altas. Considerando la figura 1, uno se puede dar una noción de qué valores de β_1 son "cercanos" a 0.8 y cuáles no, en el sentido de la capacidad del test para distinguir entre ellos.

El cuadro 4 muestra los estadísticos de resumen de las estimaciones de β_0 y β_1 obtenidas por FGLS en cada una de las $B = 5\,000$ simulaciones para cada tamaño de muestra. No ocurre nada fuera de lo esperable: a medida que aumenta el tamaño de la muestra, la media y la mediana de las estimaciones se acercan a los valores verdaderos, y el desvío estándar de las estimaciones disminuye.

[9] La implementación de KDE se puede ver en la función `plot_t_kde` del script `scripts/ej1.jl`. Se optó por un KDE en lugar de un histograma para ilustrar la distribución empírica de T porque el KDE es una función suave, lo cual facilita la comparación visual con la distribución asintótica. Para $5N = 5$, por ejemplo, solo aparecían dos "*bins*" en el intervalo mostrado en la figura 2a, lo cual no era muy útil.

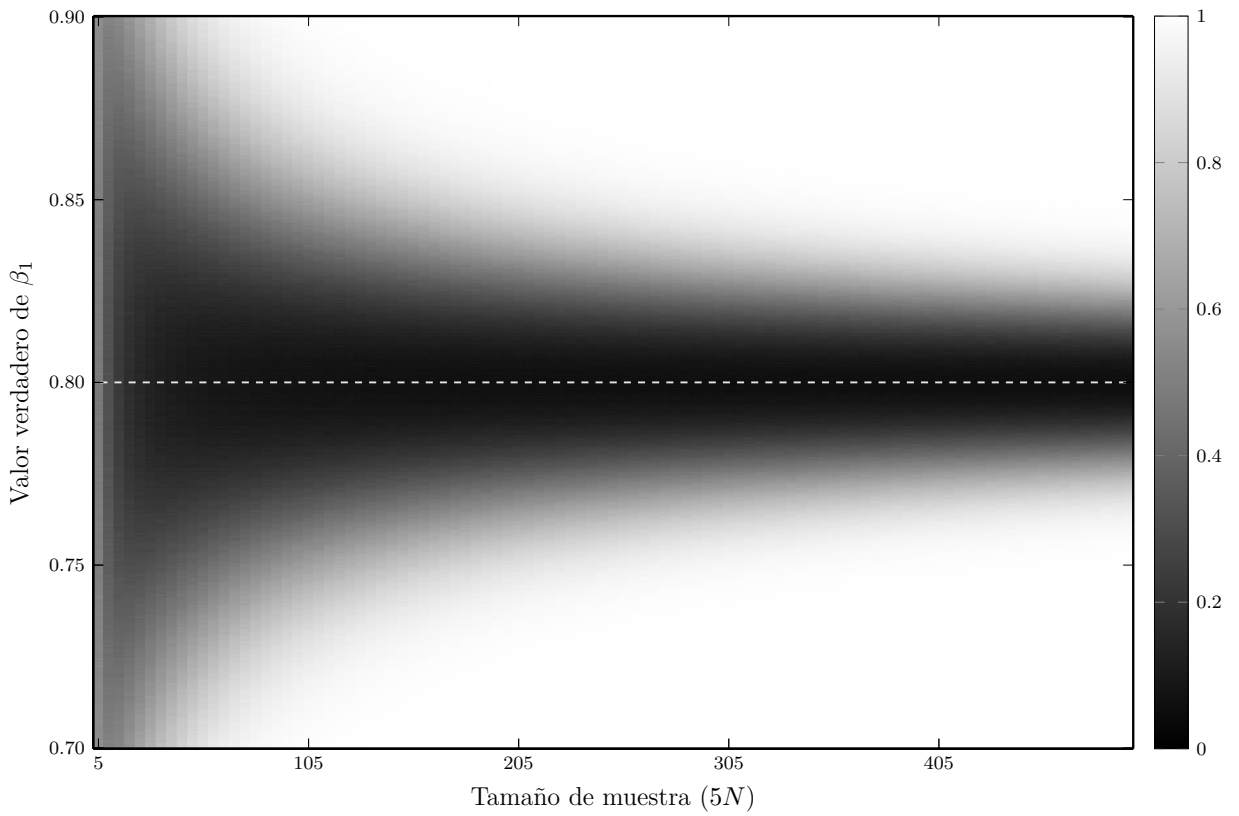


FIGURA 1. Relación entre tamaño de muestra, valor generador de datos y tasa de rechazo empírica

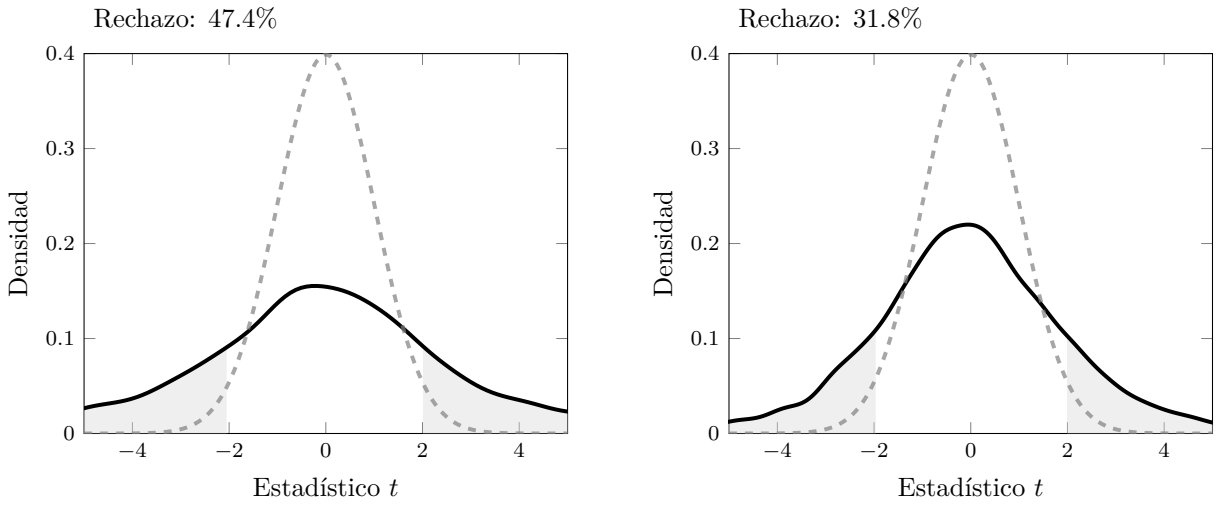
1.8. Comparación entre GLS y FGLS. Como mencionamos en la sección 1.2, la estimación del modelo transformado (4) por OLS es algebráicamente equivalente a la estimación de GLS del modelo original (2). Por lo tanto, comparar los resultados de la estimación de GLS con los de la estimación de FGLS es equivalente a comparar los resultados de la estimación por OLS del modelo transformado con los resultados de la estimación por FGLS del modelo original. La función `run_exper2` del script `scripts/ej1.jl` realiza $B = 5\,000$ simulaciones del modelo, lo transforma premultiplicando por $\Sigma^{-1/2}$, y estima por OLS el modelo transformado, obteniendo así estimaciones de GLS para cada simulación. En el cuadro 2 se muestran los estadísticos de resumen de las estimaciones de β_0 y β_1 obtenidas por GLS y FGLS en cada una de las $B = 5\,000$ simulaciones para cada tamaño de muestra. Podemos observar aquí la dinámica de convergencia en distribución del estimador FGLS al estimador GLS a medida que aumenta el tamaño de muestra. También podemos ver, tanto en la tabla 3 (que muestra las tasas de rechazo) como en la figura 3, que la distribución empírica bajo la hipótesis nula del estadístico T se acerca mucho más a la distribución asintótica (porque la *sampling distribution* de T es exactamente $N(0, 1)$ bajo GLS), y por lo tanto el tamaño empírico también se encuentra cerca de su valor asintótico para todo tamaño de muestra.

CUADRO 2. Estimación por GLS y FGLS

	FGLS				GLS			
	$\widetilde{\beta}_0$		$\widetilde{\beta}_1$		$\widetilde{\beta}_0$		$\widetilde{\beta}_1$	
$5N$	Media (desvío)	Mediana	Media (desvío)	Mediana	Media (desvío)	Mediana	Media (desvío)	Mediana
5	−2.9051 (5.5342)	−2.8620	0.7965 (0.1967)	0.7956	−3.0228 (4.6455)	−2.9909	0.8011 (0.1660)	0.8021
10	−2.9697 (2.9882)	−2.9149	0.7983 (0.1047)	0.7967	−3.0127 (2.4693)	−2.9936	0.8003 (0.0858)	0.7991
30	−3.0070 (1.4067)	−3.0254	0.8001 (0.0483)	0.8013	−3.0147 (1.2645)	−3.0305	0.8001 (0.0437)	0.8003
100	−2.9960 (0.6922)	−3.0048	0.7994 (0.0238)	0.7996	−3.0168 (0.6695)	−3.0042	0.8006 (0.0229)	0.8009
200	−3.0038 (0.4709)	−3.0060	0.8001 (0.0162)	0.8002	−2.9985 (0.4751)	−2.9990	0.7998 (0.0162)	0.7998
500	−2.9997 (0.3002)	−3.0019	0.8002 (0.0103)	0.8002	−2.9993 (0.2979)	−2.9999	0.7999 (0.0102)	0.8000

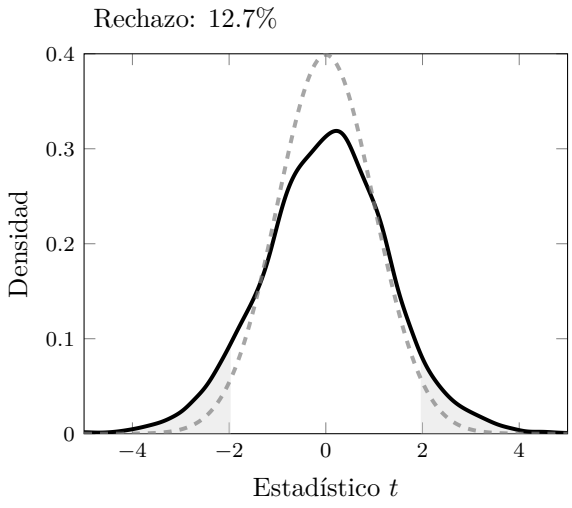
CUADRO 3. Tasas de rechazo empíricas para GLS y FGLS

	FGLS		GLS	
$5N$	1 %	5 %	1 %	5 %
5	37.48 %	47.44 %	0.98 %	5.28 %
10	20.68	31.76	0.84	4.66
30	5.28	12.7	0.86	4.9
100	1.7	7.26	0.94	4.86
200	1.36	5.58	1.32	5.14
500	1.24	5.5	1.1	4.94

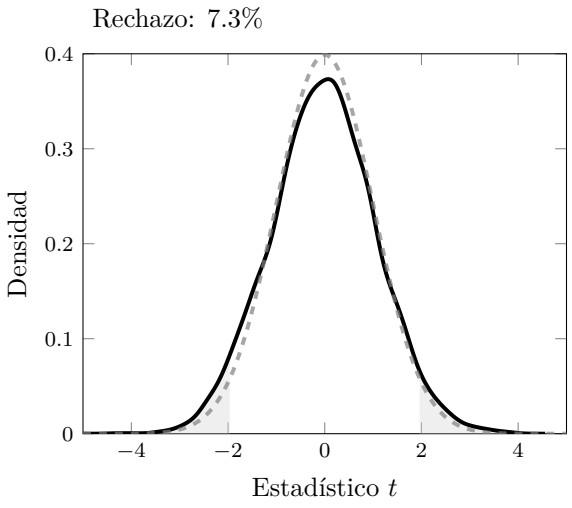


(A) $5N = 5$

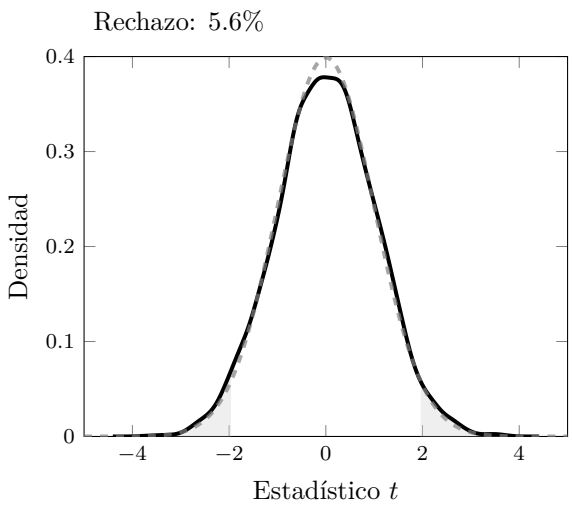
(B) $5N = 10$



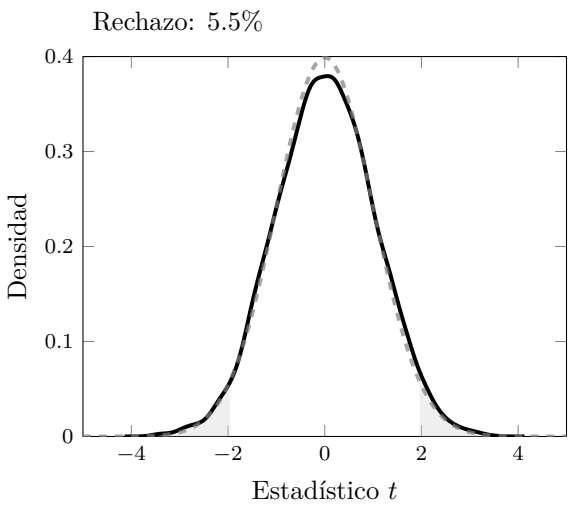
(C) $5N = 30$



(D) $5N = 100$



(E) $5N = 200$

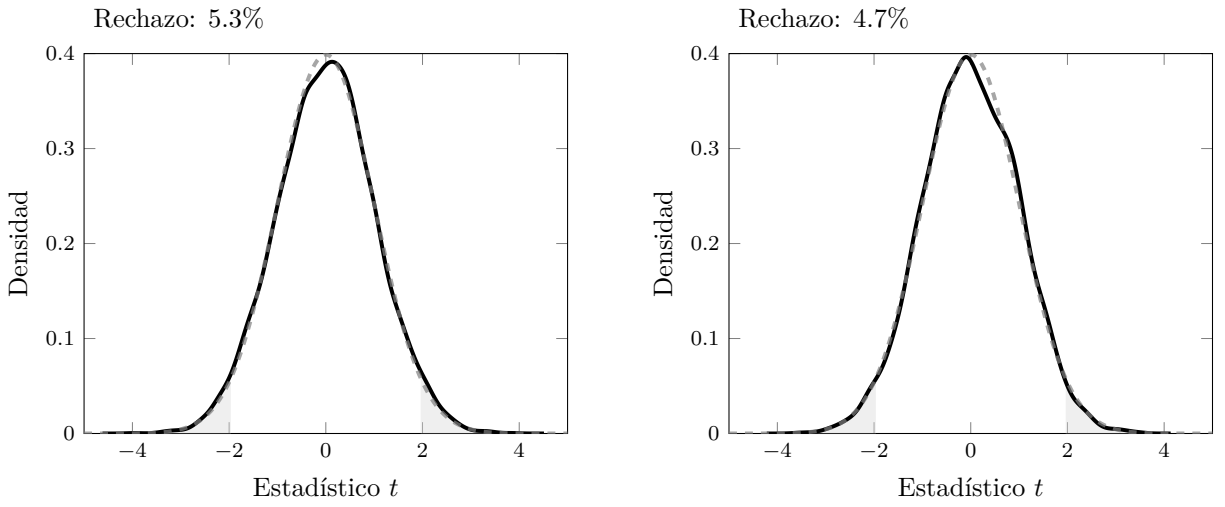


(F) $5N = 500$

FIGURA 2. Distribución empírica del estadístico t bajo $H_0 : \beta_1 = 0.8$ para FGLS

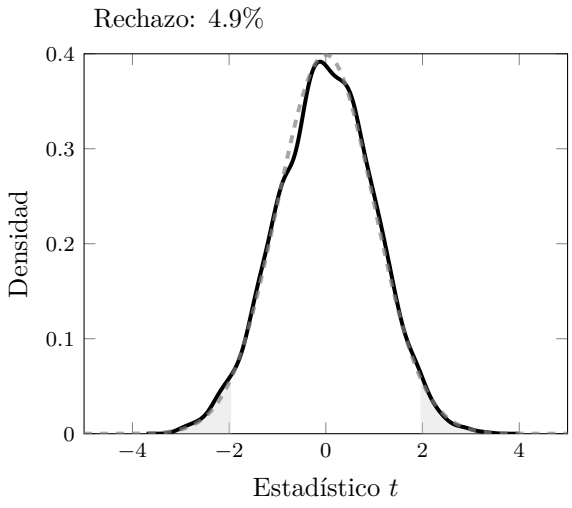
CUADRO 4. Estadísticos de resumen de estimaciones

	$\beta_1 = 0$				$\beta_1 = 0.4$				$\beta_1 = 0.8$			
	$\widetilde{\beta}_{0,\text{fgls}}$		$\widetilde{\beta}_{1,\text{fgls}}$		$\widetilde{\beta}_{0,\text{fgls}}$		$\widetilde{\beta}_{1,\text{fgls}}$		$\widetilde{\beta}_{0,\text{fgls}}$		$\widetilde{\beta}_{1,\text{fgls}}$	
$5N$	Media (desvío)	Mediana	Media (desvío)	Mediana	Media (desvío)	Mediana	Media (desvío)	Mediana	Media (desvío)	Mediana	Media (desvío)	Mediana
5	−3.0213 (5.3683)	−3.0288	−0.0017 (0.1881)	0.0002	−2.9591 (5.4605)	−2.9627	0.3992 (0.1981)	0.3994	−2.9051 (5.5342)	−2.8620	0.7965 (0.1967)	0.7956
10	−3.0515 (3.0641)	−3.0639	0.0026 (0.1057)	0.0014	−3.0590 (2.9895)	−3.0453	0.4017 (0.1044)	0.4025	−2.9697 (2.9882)	−2.9149	0.7983 (0.1047)	0.7967
30	−2.9800 (1.4150)	−2.9936	−0.0008 (0.0484)	−0.0014	−2.9694 (1.3853)	−2.9472	0.3989 (0.0479)	0.3991	−3.0070 (1.4067)	−3.0254	0.8001 (0.0483)	0.8013
100	−2.9918 (0.6972)	−2.9954	−0.0002 (0.0240)	0.0001	−3.0077 (0.6963)	−3.0044	0.4000 (0.0239)	0.4002	−2.9960 (0.6922)	−3.0048	0.7994 (0.0238)	0.7996
200	−3.0005 (0.4783)	−3.0011	−0.0000 (0.0164)	0.0001	−3.0043 (0.4723)	−3.0155	0.4000 (0.0163)	0.4002	−3.0038 (0.4709)	−3.0060	0.8001 (0.0162)	0.8002
500	−3.0004 (0.3008)	−3.0030	0.0000 (0.0103)	−0.0002	−3.0079 (0.2961)	−3.0062	0.4002 (0.0103)	0.4001	−2.9997 (0.3002)	−3.0019	0.8002 (0.0103)	0.8002

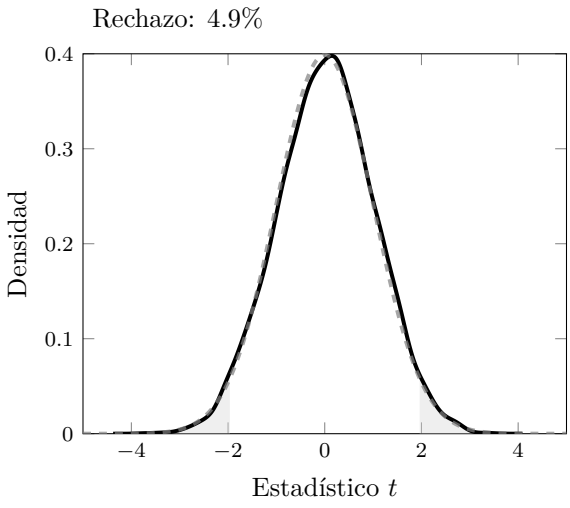


(A) $5N = 5$

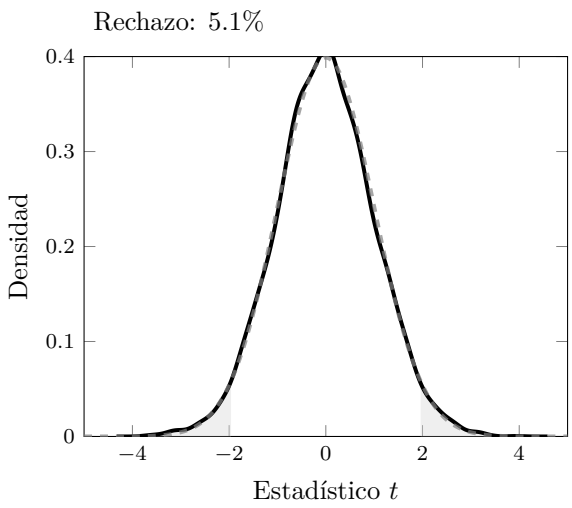
(B) $5N = 10$



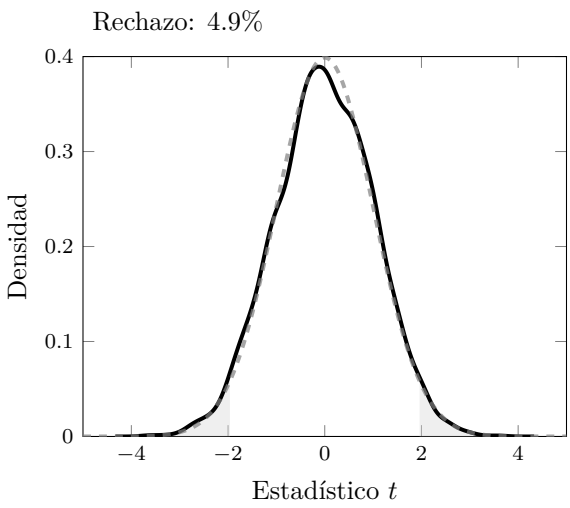
(C) $5N = 30$



(D) $5N = 100$



(E) $5N = 200$



(F) $5N = 500$

FIGURA 3. Distribución empírica del estadístico t bajo $H_0 : \beta_1 = 0.8$ para GLS

2. Propiedades de muestra finita del test de White

En este ejercicio se le pide evaluar el funcionamiento del test de heterocedasticidad de White y de la corrección sugerida por este autor para estimar la matriz de varianzas y covarianzas de los coeficientes estimados por mínimos cuadrados clásicos en muestras finitas. El objetivo final del trabajo será aprender bajo qué condiciones es posible aplicar el test de heterocedasticidad y el cálculo correcto de la matriz de varianzas y covarianzas de los coeficientes estimados por mínimos cuadrados clásicos en presencia de heterocedasticidad y confiar en los resultados obtenidos. Para esto considere el modelo

$$y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \sqrt{\nu_i} u_i, \quad i = 1, 2, \ldots, n. \tag{6}$$

Para $n = 20$, x_1 se determina como una secuencia de 18 puntos espaciados uniformemente entre -1 y 1 con puntos extremos dados por -1.1 y 1.1 mientras que x_2 son cuantiles de una distribución normal estándar elegidos aleatoriamente. Las observaciones se repiten tres veces para obtener una muestra de $n = 60$, cinco veces para una muestra de $n = 100$, diez veces para una muestra de $n = 200$, veinte veces para una muestra de $n = 400$ y treinta veces para una muestra de $n = 600$. La generación de datos de la variable dependiente se hace con $\beta_0 = \beta_1 = \beta_2 = 1$. Hay tres diseños:

- Diseño 0: $u_i \sim N(0, 1)$ y $\nu_i = 1$ (normalidad y homocedasticidad).
- Diseño 1: $u_i \sim N(0, 1)$ y $\nu_i = e^{0.25x_{1i}+0.25x_{2i}}$ (normalidad y heterocedasticidad).
- Diseño 2: $u_i \sim t_5$ y $\nu_i = e^{0.25x_{1i}+0.25x_{2i}}$ (no-normalidad y heterocedasticidad).

Todas las simulaciones se basan en 5 000 replicaciones.

Ejercicio 2 (Test de Heterocedasticidad de White)**.**

a) Para el Diseño 0 genere 5 000 muestras de $n = 20$ observaciones a partir del modelo (6). Para cada muestra estime por MCC los parámetros del modelo y realice el test de hipótesis de White para contrastar que H_0: No hay heterocedasticidad. Reporte tamaño del test al 1 %, 5 % y 10 %.

b) Para los Diseños 1 y 2 genere 5 000 muestras de $n = 20$ observaciones a partir del modelo (6) y reporte el poder del test cuando el diseño utilizado es el modelo poblacional verdadero.

c) Repita el punto anterior con $n = 60$.

d) Repita el punto anterior con $n = 100$.

e) Repita el punto anterior con $n = 200$.

f) Repita el punto anterior con $n = 400$.

g) Repita el punto anterior con $n = 600$.

h) Describa detalladamente las propiedades de muestra finita del test de White de acuerdo a lo observado en los puntos anteriores.

2.1. El modelo.

Para $n = 20$, el modelo es[10]

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \sqrt{\nu_i} u_i, \quad i = 1, 2, \ldots, 20.$$

[10]Nótese que revertimos el orden de los subíndices respecto a la consigna.

donde $x_{i,1}$ es determinístico, siendo una sucesión de veinte puntos equiespaciados entre -1.1 y 1.1, y $x_{i,2} \overset{\text{iid}}{\sim} N(0,1)$.[11] Escribimos el modelo en forma matricial como

$$\boldsymbol{Y} = \boldsymbol{X}\beta + \boldsymbol{e}, \tag{7}$$

donde $\boldsymbol{Y} = (y_1, y_2, \ldots, y_{20})'$ es el vector de observaciones de la variable dependiente, $\boldsymbol{X}$ es la matriz de regresores

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} \\ 1 & x_{2,1} & x_{2,2} \\ \vdots & \vdots & \vdots \\ 1 & x_{20,1} & x_{20,2} \end{bmatrix} = \begin{bmatrix} 1 & -1.1 & x_{1,2} \\ 1 & -1 & x_{2,2} \\ \vdots & \vdots & \vdots \\ 1 & 1.1 & x_{20,2} \end{bmatrix},$$

el vector de parámetros es $\beta = (\beta_0, \beta_1, \beta_2)'$ y el vector de errores es $\boldsymbol{e} = (e_1, e_2, \ldots, e_{20})'$, con $e_i = \sqrt{\nu_i}u_i$. Para $n = 20m > 20$, la muestra se obtiene como m realizaciones del modelo (7) concatenadas verticalmente (*stacked*). Por ejemplo, para $n = 40$,

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{20} \\ y_{21} \\ y_{22} \\ \vdots \\ y_{40} \end{bmatrix} = \begin{bmatrix} 1 & -1.1 & x_{1,2} \\ 1 & -1 & x_{2,2} \\ \vdots & \vdots & \vdots \\ 1 & 1.1 & x_{20,2} \\ 1 & -1.1 & x_{21,2} \\ 1 & -1 & x_{22,2} \\ \vdots & \vdots & \vdots \\ 1 & 1.1 & x_{40,2} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{20} \\ e_{21} \\ e_{22} \\ \vdots \\ e_{40} \end{bmatrix},$$

con $x_{i,2} \overset{\text{iid}}{\sim} N(0,1)$ para $i = 1, 2, \ldots, 40$.

2.2. Test de White. El test de White (1980, pág. 825) es un test de la clase de los tests de multiplicadores de Lagrange (*LM* o *score*), que contrasta la hipótesis nula de homocedasticidad. Usando como referencia la descripción de González-Rozada (2025, pág. 67), definimos la *forma funcional de White* como la proyección lineal de la varianza del error $\sqrt{\nu_i}u_i$ en la expansión de segundo orden de los regresores, i.e.,

$$\sigma_i^2 = \gamma_0 + \gamma_1 x_{1i} + \gamma_2 x_{2i} + \gamma_3 x_{1i}^2 + \gamma_4 x_{2i}^2 + \gamma_5 x_{1i} x_{2i} + \varepsilon_i, \tag{8}$$

donde $\sigma_i^2 = \text{var}[\sqrt{\nu_i}u_i \mid x_{1i}, x_{2i}]$ es la varianza condicional del error y ε_i es un error de proyección lineal. La hipótesis nula de homocedasticidad se puede escribir para este modelo como

$$H_0 : \gamma_1 = \gamma_2 = \gamma_3 = \gamma_4 = \gamma_5 = 0, \tag{9}$$

con la alternativa de heterocedasticidad dada por la negación de esta hipótesis nula: al menos alguno de los parámetros γ_1, γ_2, γ_3, γ_4 o γ_5 es distinto de cero. Puesto que el lado izquierdo

[11] Siendo precisos, la distancia entre los dos puntos terminales y los adyacentes (0.1) es menor a la distancia entre los puntos intermedios ($1/9 \approx 0.11$), pero es irrelevante. Respecto a x_2, interpreto que la consigna se refiere con "cuantiles de una distribución normal estándar elegidos aleatoriamente" a $x_2 = \Phi^{-1}(u)$, donde $u \overset{\text{iid}}{\sim} U(0,1)$ y Φ es la función de distribución acumulada de una normal estándar, lo cual es equivalente a $x_2 \overset{\text{iid}}{\sim} N(0,1)$. Hay una ambigüedad en la consigna respecto a la generación de muestras de tamaño mayor a $20m > 20$: ¿debería generarse una sola muestra de $n = 20$, luego copiarla el número de veces necesario para alcanzar el tamaño deseado, o generarse m muestras aleatorias de $n = 20$? Optamos por la segunda opción, aunque los resultados son robustos a la especificación.

de (8) no es observable, el test de White se basa en la estimación de esta ecuación usando como variable dependiente los residuos al cuadrado de la estimación por MCC del modelo (6). Denotando por $\widehat{e}_i$ al residuo de OLS de la ecuación (6) para la observación i, definimos la *regresión auxiliar*

$$\widehat{e}_i^2 = \gamma_0 + \gamma_1 x_{1i} + \gamma_2 x_{2i} + \gamma_3 x_{1i}^2 + \gamma_4 x_{2i}^2 + \gamma_5 x_{1i} x_{2i} + \varepsilon_i. \tag{10}$$

Estimando por OLS la regresión auxiliar (10) y denotando por R^2 el coeficiente de determinación de esta regresión, el estadístico de test de White es

$$LM := nR^2,$$

que bajo la hipótesis nula de homocedasticidad se distribuye asintóticamente como una chi-cuadrado con 5 grados de libertad (correspondientes a la cantidad de parámetros iguales a cero en la hipótesis nula (9)):

$$LM \xrightarrow{d} \chi_5^2 \quad \text{bajo } H_0.$$

Por lo tanto, nos referimos con *test de nivel o tamaño* α al test a una cola que tiene nivel asintótico α; más precisamente, región de rechazo

$$R_\alpha = \{LM > \chi_{5,1-\alpha}^2\},$$

donde $\chi_{5,1-\alpha}^2$ es el cuantil $1-\alpha$ de una variable chi-cuadrado con 5 grados de libertad.[12]

2.3. Monte Carlo con el test de White. Para analizar las propiedades de muestra finita del test de White, realizamos un experimento de Monte Carlo, generando $B = 5\,000$ muestras de cada diseño y tamaño de muestra, estimando por OLS el modelo (6) para cada muestra, obteniendo los residuos al cuadrado, estimando por OLS la regresión auxiliar (10) para cada muestra, calculando el estadístico de test de White para cada muestra, y finalmente calculando la tasa de rechazo empírica para cada diseño, tamaño de muestra y nivel de significancia asintótico del test.

2.4. Implementaciones. El código para la resolución de este ejercicio se encuentra en el script `scripts/ej2.jl`. A seguir, una breve descripción de las funciones principales usadas.

- `draw_sample` genera una muestra del modelo (7) para un diseño y tamaño de muestra dados. La versión `draw_sample!` muta un *array* pre-asignado para ser más eficiente.
- `ols_fit_k3` es una implementación de OLS optimizada para el caso de $k = 3$ regresores, que es la regresión (6) en este caso. Usa acumulación escalar y *stack-allocated arrays* para ser más eficiente. Resuelve via Cholesky, a diferencia del `ols_2reg` de `scripts/ej1.jl`, que podía aprovechar expresiones escalares del caso simple.
- `ols_fig_k6` es una implementación de OLS optimizada para el caso de $k = 6$ regresores, que es la regresión auxiliar (10) en este caso. También utiliza acumulación escalar y *stack-allocated arrays* para ser más eficiente, y resuelve via Cholesky.
- `white_test` toma una muestra y un nivel de significancia y realiza el test descrito en la sección 2.2. La versión `white_test!` muta *buffers* pre-asignados para ser más eficiente.
- `run_exper` realiza el experimento de Monte Carlo para un diseño y tamaño de muestra dados, generando B muestras.

[12]Es decir, $\chi_{5,1-\alpha}^2$ es el valor tal que $\mathbb{P}[\chi_5^2 \leq \chi_{5,1-\alpha}^2] = 1-\alpha$. Para $\alpha = 0.01$, 0.05 y 0.10, usamos valores críticos de 15.09, 11.07 y 9.24, respectivamente.

CUADRO 5. Tasas de rechazo empíricas

	Diseño 0			Diseño 1			Diseño 2		
n	1 %	5 %	10 %	1 %	5 %	10 %	1 %	5 %	10 %
20	0.28 %	4.04 %	8.94 %	0.7 %	6.06 %	12.16 %	0.66 %	5.7 %	11.44 %
60	1.34	5.24	9.76	5.08	15.42	23.56	4.86	12.02	18.3
100	1.46	4.9	8.82	10.84	24.72	36.32	7.52	15.68	23.4
200	1.36	5.22	9.82	30.56	53.5	66.32	12.66	25.82	36.22
400	1.2	5.12	9.66	73.12	89.3	94.5	27.8	47.48	58.42
600	1.18	4.82	9.8	92.86	98.28	99.18	43.04	63.94	74.36

- `run_monte_carlo` realiza el experimento de Monte Carlo para todos los diseños y tamaños de muestra, generando B muestras para cada combinación, y generando gráficos y formando tablas para exportación.

Hay otras funciones relacionadas con la creación y tablas que no son mencionadas en esta sección.

2.5. Resultados del test de White. El cuadro 5 muestra las tasas de rechazo empíricas (i.e., promediadas sobre las 5 000 iteraciones del experimento) para el test de White para cada tamaño de muestra y para cada diseño, expresadas como porcentajes del total. Los nombres de las columnas, 1 %, 5 % y 10 %, corresponden al nivel de significancia asintótico del test, como fue descrito en la sección 2.2. Para el Diseño 0, que corresponde a la hipótesis nula de homocedasticidad, las tasas de rechazo empíricas comienzan relativamente cerca del valor asintótico correspondiente, y se acercan a medida que n aumenta, aunque no de manera monótona.[13] La figura **??** muestra que la distribución empírica del estadístico de test se acerca a la distribución asintótica chi-cuadrado con 5 grados de libertad a medida que aumenta el tamaño de muestra, y también que no se encuentra demasiado alejada de la distribución asintótica incluso para n chico (al menos comparándose con la relación entre el estadístico T del primer ejercicio y su distribución asintótica en la figura 2).

Para los Diseños 1 y 2, que corresponden a la hipótesis alternativa de heterocedasticidad, las tasas de rechazo empíricas comienzan relativamente bajas, tan cercanas al nivel correspondiente al Diseño 0 que es difícil distinguirlas, y aumentan a medida que n aumenta, de manera monótona. Además, como las figuras 5–7 exhiben visualmente, la tasa de rechazo del Diseño 2 se encuentra entre la del Diseño 0 y la del Diseño 1.[14] Esto sugiere que la heterocedasticidad podría ser más difícil de detectar mediante el test de White cuando hay "colas pesadas" en la distribución del error (véase la figura 4). En contraste con el ejercicio anterior, el problema asociado con el tamaño pequeño de la muestra no es el de tamaño grande (i.e., tasas de rechazo empíricas mayores a los niveles correspondientes al Diseño 0), sino el de poder pequeño (i.e., tasas de rechazo empíricas bajas para los Diseños 1 y 2—dificultad para detectar heterocedasticidad). Para valores de n chicos, como por ejemplo $n = 20$, la

[13]Esto se debe, uno presume, a la aleatoriedad de las simulaciones de Monte Carlo, pero aún con un millón de replicaciones en lugar de cinco mil, la convergencia no es monótona.

[14]Cada figura muestra la tasa de rechazo empírica para cada valor crítico (correspondiente a un nivel de significancia asintótico) para cada tamaño de muestra, con una línea para cada diseño. Las simulaciones se realizaron con $B = 100\,000$ simulaciones en lugar de $B = 5\,000$ para eliminar mejor la aleatoriedad del experimento.

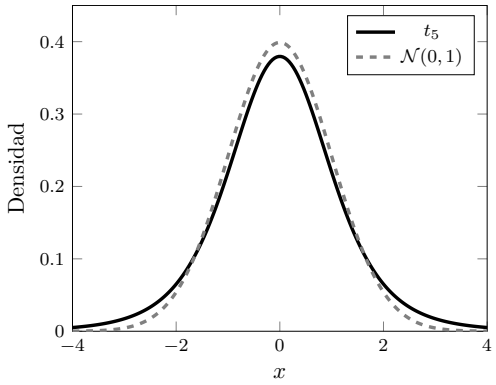


FIGURA 4. Distribución normal estándar y t_5 de Student

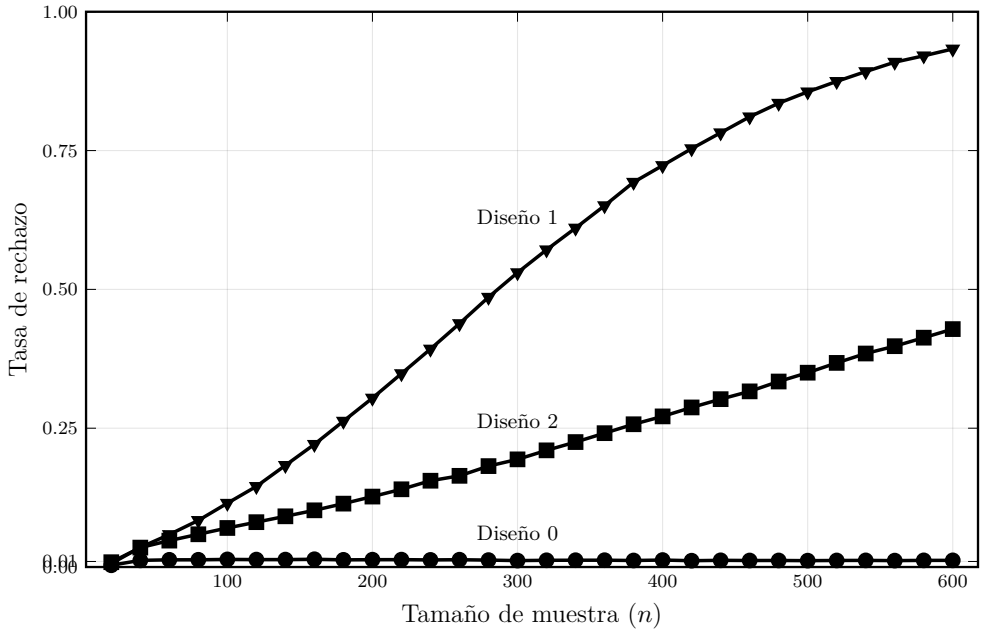


FIGURA 5. Tasas de rechazo empíricas al 1 %

probabilidad de rechazar correctamente homocedasticidad es solo un poco mayor que la probabilidad de rechazar incorrectamente homocedasticidad. A medida que aumenta n, este problema de poder pequeño se resuelve, aunque más rápido para el Diseño 1 que para el Diseño 2.

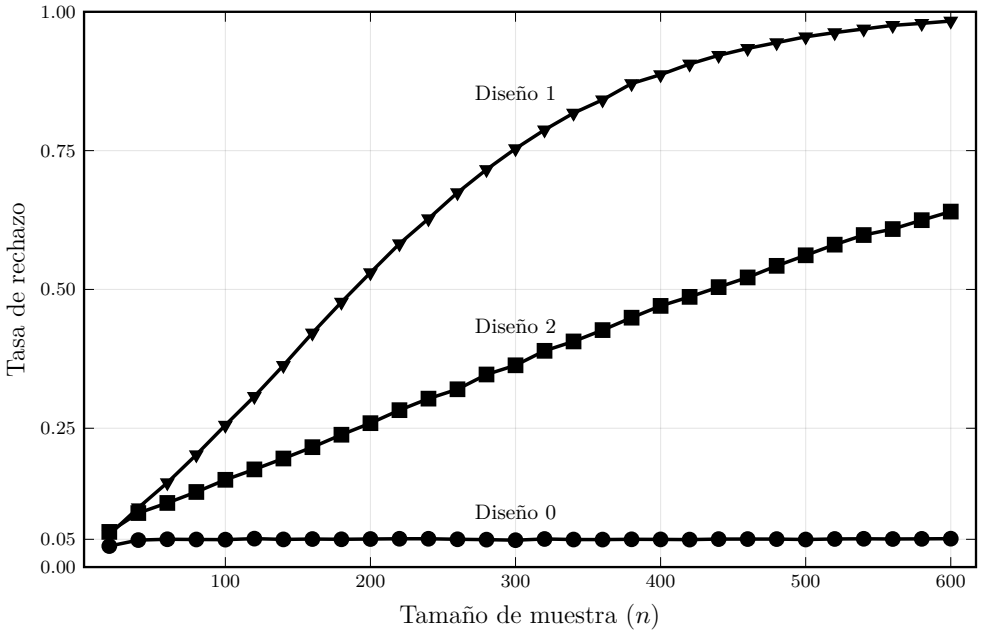


FIGURA 6. Tasas de rechazo empíricas al 5 %

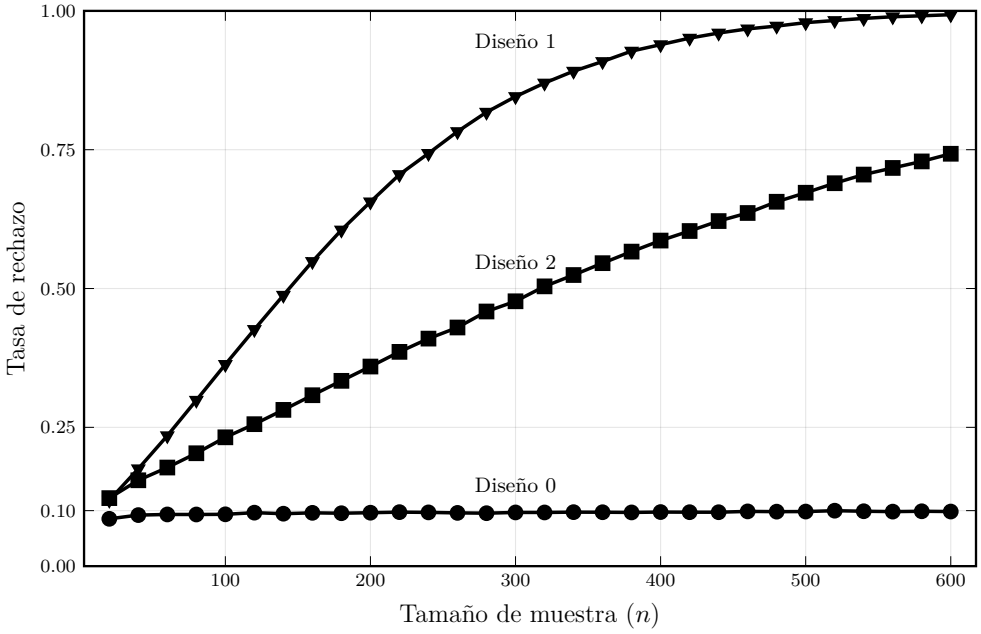
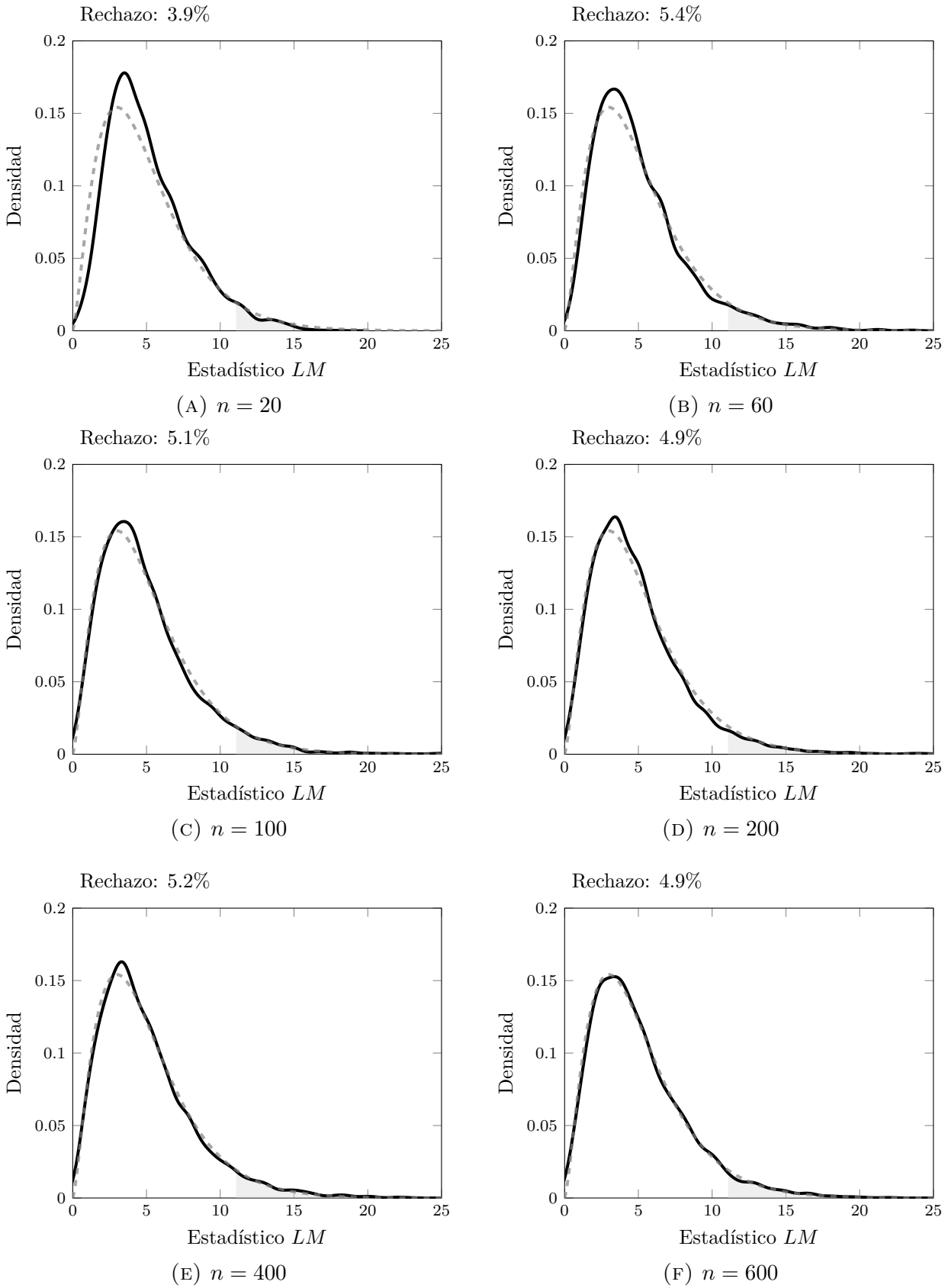


FIGURA 7. Tasas de rechazo empíricas al 10 %



(A) $n = 20$ (B) $n = 60$

(C) $n = 100$ (D) $n = 200$

(E) $n = 400$ (F) $n = 600$

FIGURA 8. Distribución empírica del estadístico LM bajo Diseño 0

Ejercicio 3 (Corrección de la matriz de varianzas y covarianzas en presencia de heterocedasticidad)**.** En el modelo de regresión lineal (6) la estimación de MCC es

$$\widehat{\beta} = \begin{pmatrix} \widehat{\beta}_0 \\ \widehat{\beta}_1 \\ \widehat{\beta}_2 \end{pmatrix} = (X'X)^{-1}X'y,$$

donde X es la matriz de variables explicativas de dimensión $n \times 3$ e y es el vector de observaciones de la variable dependiente de dimensión $n \times 1$. Bajo heterocedasticidad, la matriz de varianzas y covarianzas de los estimadores de MCC es

$$\text{var}[\widehat{\beta}] = (X'X)^{-1}X'\Omega X(X'X)^{-1}, \tag{11}$$

con $\Omega = \text{diag}(\sigma_1^2, \ldots, \sigma_n^2)$. Una estimación consistente de esta matriz está dada por la denominada matriz de White (1980),

$$\widehat{\text{var}[\widehat{\beta}]} = (X'X)^{-1}X'\widehat{\Omega}X(X'X)^{-1},$$

con $\widehat{\Omega} = \text{diag}(\widehat{u}_1^2, \ldots, \widehat{u}_n^2)$ y $\widehat{u}$ el vector de dimensión $n \times 1$ de residuos de la estimación por MCC.

a) Para cada uno de los Diseños 1 y 2 genere 5 000 muestras de $n = 20$ observaciones a partir del modelo (6). Para cada muestra estime por MCC los parámetros del modelo y reporte el sesgo relativo de la estimación de las varianzas de $\widehat{\beta}_0$, $\widehat{\beta}_1$ y $\widehat{\beta}_2$ y los sesgos relativos totales. El sesgo relativo (b_0, b_1, b_2) se define como el promedio, a lo largo de las 5 000 simulaciones, de la varianza estimada menos la varianza verdadera dividido por la varianza verdadera, y el sesgo relativo total es la suma del valor absoluto de b_0, b_1 y b_2. El sesgo relativo total es una medida del sesgo agregado de las tres variables.
b) Repita el punto anterior con $n = 60$.
c) Repita el punto anterior con $n = 100$.
d) Repita el punto anterior con $n = 200$.
e) Repita el punto anterior con $n = 400$.
f) Repita el punto anterior con $n = 600$.
g) Repita los puntos a) a f) pero ahora construya la estimación de la matriz de White usando una matriz diagonal con los errores verdaderos elevados al cuadrado en lugar de utilizar la matriz diagonal con los residuos elevados al cuadrado.
h) Describa detalladamente las propiedades de muestra finita de la estimación de las varianzas de los coeficientes de MCC robustas ante la presencia de heterocedasticidad (con el procedimiento de White) de acuerdo a lo observado en los puntos anteriores.

2.6. Estimadores de matriz de varianzas y covarianzas. La siguiente discusión de estimadores alternativos de la matriz de varianzas y covarianzas de los coeficientes estimados por OLS en presencia de heterocedasticidad es esencialmente la de Hansen (2022, págs. 114-116).

Como dice la ecuación (11), la matriz de varianzas y covarianzas del estimador OLS $\widehat{\beta}$ se puede escribir

$$\boldsymbol{V}_{\widehat{\beta}} := \text{var}[\widehat{\beta} \mid \boldsymbol{X}] = (\boldsymbol{X'X})^{-1}(\boldsymbol{X'DX})(\boldsymbol{X'X})^{-1}, \tag{12}$$

donde

$$\boldsymbol{D} := \Omega = \operatorname{diag}(\sigma_1^2, \ldots, \sigma_n^2) = E[\boldsymbol{e}\boldsymbol{e}' \mid \boldsymbol{X}].$$

Si pudiésemos observar los errores, podríamos estimar D de manera insesgada con $\boldsymbol{e}\boldsymbol{e}'$, y construir el estimador

$$\widehat{\boldsymbol{V}}_{\widehat{\beta}}^{\text{ideal}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\Big(\sum_{i=1}^{n} X_i X_i' e_i^2\Big)(\boldsymbol{X}'\boldsymbol{X})^{-1}. \tag{13}$$

El estimador de la ecuación (13) es el que propone el inciso g). No es un estimador, estrictamente hablando, porque no es función de la muestra, pero como estimador *infeasible* sirve como una referencia contra la cual comparar el desempeño de los estimadores factibles.

Desarrollada en su primera instancia por Eicker (1963) e introducida a la econometría por White (1980), la denominada *matriz de White* (o de Eicker-White) se define

$$\widehat{\boldsymbol{V}}_{\widehat{\beta}}^{\text{HC0}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\Big(\sum_{i=1}^{n} X_i X_i' \widehat{e}_i^2\Big)(\boldsymbol{X}'\boldsymbol{X})^{-1}. \tag{14}$$

El "HC0" en el supraíndice se refiere a que es el estimador *baseline* en la clase *heteroskedasticity-consistent*. En la práctica, como en la implementación de la opción `robust` de Stata, se suele hacer una corrección *ad-hoc* propuesta por Hinkley (1977),

$$\widehat{\boldsymbol{V}}_{\widehat{\beta}}^{\text{HC1}} = \frac{n}{n-k}(\boldsymbol{X}'\boldsymbol{X})^{-1}\Big(\sum_{i=1}^{n} X_i X_i' \widehat{e}_i^2\Big)(\boldsymbol{X}'\boldsymbol{X})^{-1}. \tag{15}$$

Otros estimadores que se utilizan son el "HC2" de Horn et al. (1975) y el "HC3" propuesto por MacKinnon y White (1985) y por Andrews (1991), que utilizan en lugar de los residuos de OLS, residuos estandarizados y errores de predicción, respectivamente.

La consigna hace referencia a "la matriz de White". En la implementación, usamos las dos versiones: la original (HC0) y la que incluye la corrección de Hinkley (HC1).

2.7. Sesgo relativo. Consideremos el contexto de un experimento Monte Carlo con replicaciones $b = 1, 2, \ldots, B$. Sea $\widehat{\boldsymbol{V}}^{(b)}$ la b-ésima instancia de un estimador de la matriz de varianzas y covarianzas de $\widehat{\beta}$, y $\widehat{\boldsymbol{V}}_{jj}^{(b)}$ la varianza estimada de $\widehat{\beta}_j$ dada por el elemento j-ésimo de la diagonal de $\widehat{\boldsymbol{V}}^{(b)}$. De la misma manera, la verdadera varianza condicional de $\widehat{\beta}_j$ dada la b-ésima matriz de datos $X^{(b)}$ es el elemento j-ésimo de la diagonal de $\boldsymbol{V}_{\widehat{\beta}}^{(b)}$, que denotamos por $\boldsymbol{V}_{\widehat{\beta},jj}^{(b)}$.[15] El sesgo relativo de $\widehat{\boldsymbol{V}}_{jj}$ denotado por b_j se define como el promedio a lo largo de las B simulaciones de la varianza estimada menos la varianza verdadera dividido por la varianza verdadera, es decir,

$$b_j = \frac{1}{B}\sum_{b=1}^{B} \frac{\widehat{\boldsymbol{V}}_{jj}^{(b)} - \boldsymbol{V}_{\widehat{\beta},jj}^{(b)}}{\boldsymbol{V}_{\widehat{\beta},jj}^{(b)}}.$$

El sesgo relativo total es la suma del valor absoluto de b_0, b_1 y b_2:

$$\|\boldsymbol{b}\|_1 = |b_0| + |b_1| + |b_2|.$$

La notación alude a que el sesgo relativo total es la norma ℓ_1 del vector de sesgos relativos $\boldsymbol{b} = (b_0, b_1, b_2)'$ en $\mathbb{R}^3$.

[15] La varianza condicional varía con la replicación porque varía la matriz de datos $X^{(b)}$.

2.8. El Experimento. El experimento consiste en generar $B = 5\,000$ muestras de cada diseño y cada tamaño de muestra, estimar por OLS el modelo (6) para cada muestra, y luego, a lo largo de las B muestras, calcular el sesgo relativo de la matriz de White (HC0 y HC1, de las ecuaciones(14) y (15)) y del estimador *infeasible* ideal de la ecuación (13).

2.9. Implementaciones. El código para la resolución de este ejercicio se encuentra en el script `scripts/ej3.jl`. A seguir, una breve descripción de las funciones principales usadas.

- `draw_sample_e` genera una muestra de la misma manera que la función `draw_sample` de `scripts/ej2.jl`, pero también devuelve los errores verdaderos de la muestra, que son necesarios para el inciso g). La razón por la cual no se usa la misma función para ambos es que el error no es necesario para el experimento del inciso anterior y ocupa memoria. La versión `draw_sample_e!` muta un *array* pre-asignado para ser más eficiente.
- `vcov_diags` computa los elementos diagonales de las matrices en las ecuaciones (12), (13), (14) y (15), i.e., computan varianzas de coeficientes o estimaciones de varianzas de coeficientes.
- `run_exper` realiza el experimento de Monte Carlo para un diseño y tamaño de muestra dados, generando B muestras.
- `run_monte_carlo` realiza el experimento de Monte Carlo para todos los diseños y tamaños de muestra, generando B muestras para cada combinación, y generando gráficos y formando tablas para exportación.

Hay otras funciones relacionadas con la creación y tablas que no son mencionadas en esta sección.

2.10. Resultados. El cuadro 6 muestra los sesgos relativos de la matriz de White (HC0) y del estimador *infeasible* ideal, para cada diseño y cada tamaño de muestra. El cuadro 7 compara la matriz de White sin la corrección de Hinkley (HC0) con la que incluye esta corrección (HC1). Las figuras 9 y 10 muestran el sesgo relativo total para cada diseño y cada tamaño de muestra, mientras que las figuras 11a–13b muestran el sesgo relativo de cada componente de la matriz de varianzas y covarianzas.[16] En primer lugar, notemos que HC1 domina a HC0 en términos de sesgo relativo: la estimación de la matriz de varianzas y covarianzas con la corrección de Hinkley tiene menor sesgo en valor absoluto para todo n, para todo diseño, y para cada componente de la matriz de varianzas y covarianzas. Además, el sesgo relativo para la constante, b_0, es prácticamente idéntico entre HC1 e $V_{\widehat{\beta}}^{\text{ideal}}$, y muy cercano a cero, mientras que el sesgo de HC0 es considerable. Todos los sesgos de HC0 son negativos, y la mayoría de los sesgos de HC1, lo cual quiere decir que subestiman la varianza de los coeficientes, aunque el sesgo de HC1 es bastante menor que el de HC0, en particular para las varianzas de $\widehat{\beta}_0$ y $\widehat{\beta}_1$.

[16]Las tablas usan la cantidad de simulaciones que dice la consigna, $B = 5\,000$, mientras que las figuras, para obtener una mejor representación gráfica y eliminar características espurias del experimento, como lineas que se cruzan para algunos n, usan $B = 100\,000$ simulaciones.

CUADRO 6. Sesgo relativo de HC0 e $V_{\widehat{\beta}}^{\text{ideal}}$

	Diseño 1								Diseño 2							
	$\widehat{V}^{\text{HC0}}$				$\widehat{V}^{\text{ideal}}$				$\widehat{V}^{\text{HC0}}$				$\widehat{V}^{\text{ideal}}$			
n	b_0	b_1	b_2	$\|\boldsymbol{b}\|_1$	b_0	b_1	b_2	$\|\boldsymbol{b}\|_1$	b_0	b_1	b_2	$\|\boldsymbol{b}\|_1$	b_0	b_1	b_2	$\|\boldsymbol{b}\|_1$
20	−0.154	−0.191	−0.233	0.579	0.002	0.003	−0.003	0.008	−0.152	−0.190	−0.227	0.569	0.004	−0.003	0.018	0.025
60	−0.052	−0.067	−0.091	0.209	−0.001	−0.002	−0.006	0.009	−0.051	−0.065	−0.082	0.199	0.001	−0.002	0.006	0.008
100	−0.030	−0.040	−0.051	0.121	0.002	−0.001	0.003	0.005	−0.030	−0.033	−0.049	0.112	0.001	0.006	0.004	0.011
200	−0.015	−0.021	−0.029	0.065	0.000	−0.001	−0.002	0.003	−0.017	−0.017	−0.033	0.067	−0.002	0.002	−0.007	0.011
400	−0.005	−0.007	−0.009	0.021	0.002	0.003	0.005	0.010	−0.007	−0.010	−0.009	0.027	0.001	−0.000	0.004	0.005
600	−0.006	−0.008	−0.009	0.023	−0.001	−0.002	0.000	0.002	−0.003	−0.004	−0.001	0.008	0.002	0.003	0.008	0.013

CUADRO 7. Comparación entre HC0 y HC1

	Diseño 1								Diseño 2							
	$\widehat{V}^{\text{HC0}}$				$\widehat{V}^{\text{HC1}}$				$\widehat{V}^{\text{HC0}}$				$\widehat{V}^{\text{HC1}}$			
n	b_0	b_1	b_2	$\|\boldsymbol{b}\|_1$	b_0	b_1	b_2	$\|\boldsymbol{b}\|_1$	b_0	b_1	b_2	$\|\boldsymbol{b}\|_1$	b_0	b_1	b_2	$\|\boldsymbol{b}\|_1$
20	−0.154	−0.191	−0.233	0.579	−0.005	−0.048	−0.098	0.151	−0.152	−0.190	−0.227	0.569	−0.003	−0.047	−0.090	0.140
60	−0.052	−0.067	−0.091	0.209	−0.002	−0.018	−0.043	0.063	−0.051	−0.065	−0.082	0.199	−0.001	−0.016	−0.034	0.051
100	−0.030	−0.040	−0.051	0.121	0.000	−0.011	−0.021	0.032	−0.030	−0.033	−0.049	0.112	−0.000	−0.003	−0.020	0.023
200	−0.015	−0.021	−0.029	0.065	−0.000	−0.006	−0.014	0.020	−0.017	−0.017	−0.033	0.067	−0.002	−0.002	−0.019	0.023
400	−0.005	−0.007	−0.009	0.021	0.002	0.001	−0.001	0.004	−0.007	−0.010	−0.009	0.027	0.000	−0.003	−0.002	0.005
600	−0.006	−0.008	−0.009	0.023	−0.001	−0.003	−0.004	0.008	−0.003	−0.004	−0.001	0.008	0.002	0.001	0.004	0.007

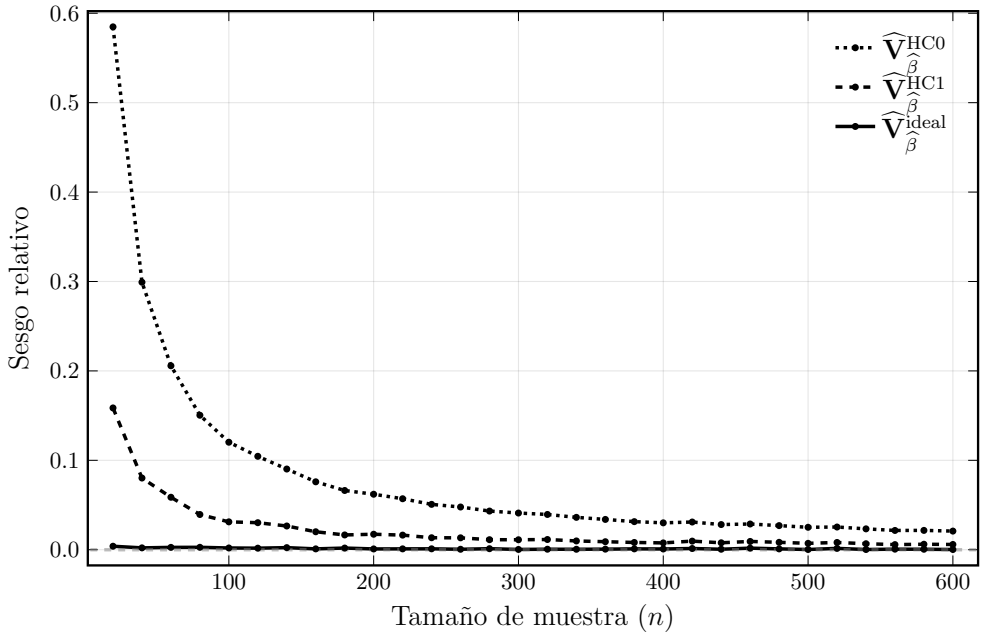


FIGURA 9. Sesgo relativo total $\|\boldsymbol{b}\|_1$ para el Diseño 1

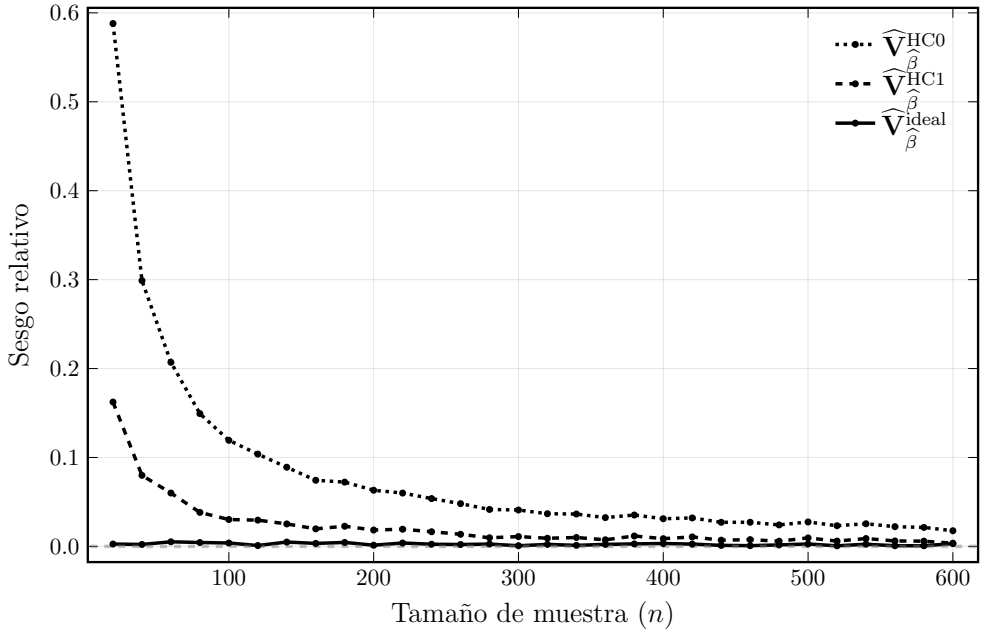
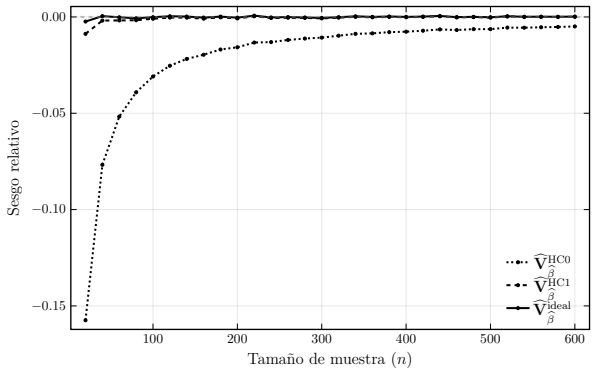
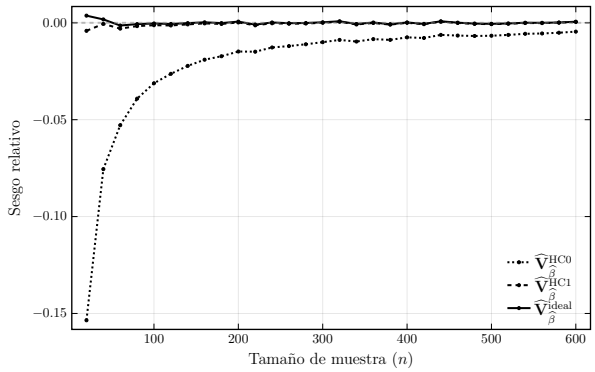


FIGURA 10. Sesgo relativo total $\|\boldsymbol{b}\|_1$ para el Diseño 2

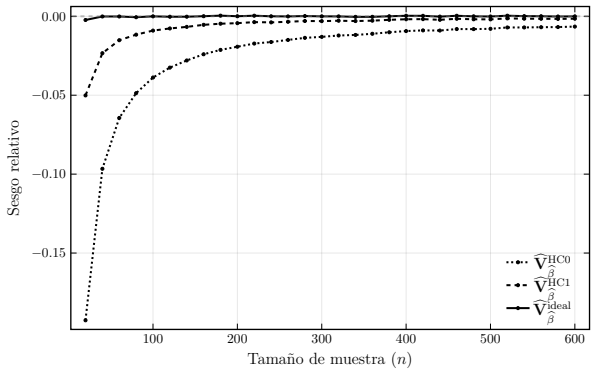


(A) Diseño 1

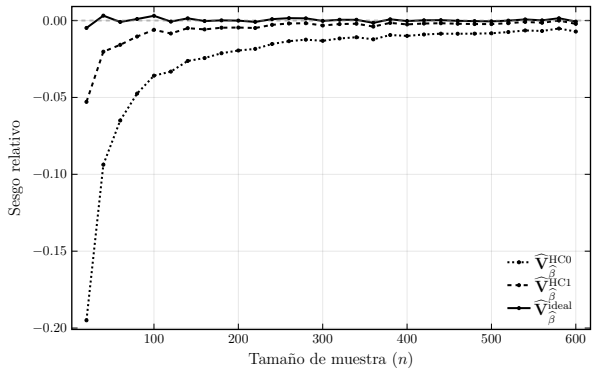


(B) Diseño 2

FIGURA 11. Sesgo relativo b_0

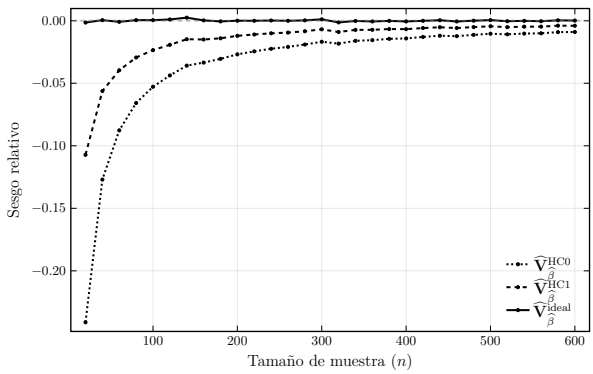


(A) Diseño 1

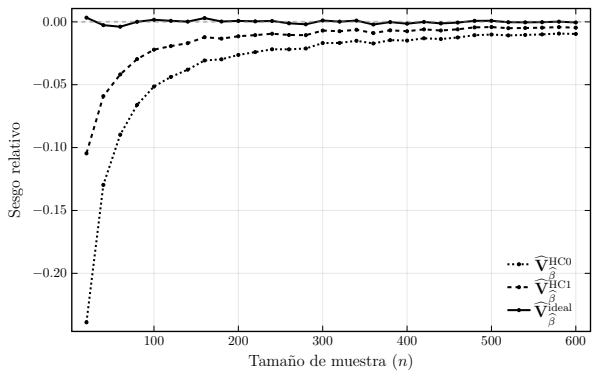


(B) Diseño 2

FIGURA 12. Sesgo relativo b_1



(A) Diseño 1



(B) Diseño 2

FIGURA 13. Sesgo relativo b_2

Referencias

Aitken, A. C. (1935). "On Least Squares and Linear Combination of Observations". En: *Proceedings of the Royal Society of Edinburgh* 55, págs. 42-48. ISSN: 0370-1646. DOI: 10.1017/S0370164600014346. URL: https://www.cambridge.org/core/product/identifier/S0370164600014346/type/journal_article (visitado 15-02-2025).

Andrews, Donald W. K. (1991). "Asymptotic Normality of Series Estimators for Nonparametric and Semiparametric Regression Models". En: *Econometrica* 59.2, págs. 307-345. ISSN: 0012-9682. DOI: 10.2307/2938259. JSTOR: 2938259. URL: https://www.jstor.org/stable/2938259 (visitado 21-02-2026).

Eicker, F. (jun. de 1963). "Asymptotic Normality and Consistency of the Least Squares Estimators for Families of Linear Regressions". En: *The Annals of Mathematical Statistics* 34.2, págs. 447-456. ISSN: 0003-4851. DOI: 10.1214/aoms/1177704156. URL: http://projecteuclid.org/euclid.aoms/1177704156 (visitado 20-02-2026).

González-Rozada, Martín (2025). "Lecture 2". Lecture notes, Econometría, Universidad Torcuato Di Tella (Buenos Aires).

Hansen, Bruce E. (2022). *Econometrics*. Princeton Oxford: Princeton University Press. 1044 págs. ISBN: 978-0-691-23589-9.

Hinkley, David V. (1977). "Jackknifing in Unbalanced Situations". En: *Technometrics* 19.3, págs. 285-292. ISSN: 0040-1706. DOI: 10.2307/1267698. JSTOR: 1267698. URL: https://www.jstor.org/stable/1267698 (visitado 20-02-2026).

Horn, Susan D., Roger A. Horn y David B. Duncan (1975). "Estimating Heteroscedastic Variances in Linear Models". En: *Journal of the American Statistical Association* 70.350, págs. 380-385. ISSN: 0162-1459. DOI: 10.2307/2285827. JSTOR: 2285827. URL: https://www.jstor.org/stable/2285827 (visitado 21-02-2026).

MacKinnon, James G y Halbert White (1 de sep. de 1985). "Some Heteroskedasticity-Consistent Covariance Matrix Estimators with Improved Finite Sample Properties". En: *Journal of Econometrics* 29.3, págs. 305-325. ISSN: 0304-4076. DOI: 10.1016/0304-4076(85)90158-7. URL: https://www.sciencedirect.com/science/article/pii/0304407685901587 (visitado 21-02-2026).

White, Halbert (1980). "A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity". En: *Econometrica* 48.4, págs. 817-838. ISSN: 0012-9682. DOI: 10.2307/1912934. JSTOR: 1912934. URL: https://www.jstor.org/stable/1912934 (visitado 16-02-2025).

Apéndice A. Código

A.1. Ejercicio 1. El contenido del archivo `ej1.jl` es el siguiente

```
# ================================================================
# TRABAJO FINAL ECONOMETRIA
# Ejercicio 1: Propiedades de muestra finita de FGLS
#
# Autor: Felipe Tappata
# Fecha: Febrero 2026
# ================================================================

# Load dependencies
using Random, Plots, Statistics
using LinearAlgebra
using Printf, LaTeXStrings
using KernelDensity
using Distributions
using PGFPlotsX

# Set seed
const SEED = 8869
Random.seed!(SEED)

# Set plotting backend
pgfplotsx()
default(tex_output_standalone=false)


# === ESTIMATION FUNCTIONS ===

"""
    draw_sample(N; rng=Random.default_rng(),
                    beta0=-3.0,
                    beta1=0.8,
                    omega=(4.0, 9.0, 16.0, 25.0, 36.0))

Generate one sample of length `5N` from the heteroskedastic linear model
`y = beta0 + beta1*x + u`, with `x ~ Uniform(1,50)` and `Var(u) = Ω ⊗ I_N`.

# Arguments
- `N::Integer`: number of replications per type (total sample size is `5N`).

# Keyword arguments
- `rng::AbstractRNG`: random number generator (default: `Random.default_rng()`).
- `beta0::Real`: intercept.
- `beta1::Real`: slope.
```

```
- `omega::NTuple{5,<:Real}`: variances `(ω₁,…,ω₅)` for the five types.

# Returns
- `x::Vector{Float64}`: regressor values, length `5N`.
- `y::Vector{Float64}`: dependent variable values, length `5N`.
- `k::Vector{Int}`: type index in `{1,…,5}` for each observation.

The stacking order is consistent with `Σ = Ω ⊗ I_N`:
all observations of type `k` are contiguous.
"""
function draw_sample(
    N::Integer;
    rng::AbstractRNG = Random.default_rng(),
    beta0::Real = -3.0,
    beta1::Real = 0.8,
    omega::NTuple{5,<:Real} = (4.0, 9.0, 16.0, 25.0, 36.0),
)
    N = Int(N)
    n = 5N
    x = Vector{Float64}(undef, n)
    y = Vector{Float64}(undef, n)
    k = Vector{Int}(undef, n)

    idx = 0
    @inbounds for type in 1:5
        σ = sqrt(Float64(omega[type]))
        for _ in 1:N
            idx += 1
            xi = 1.0 + 49.0 * rand(rng)      # U(1,50)
            ui = σ * randn(rng)              # N(0, ω_type)
            x[idx] = xi
            y[idx] = Float64(beta0) + Float64(beta1)*xi + ui
            k[idx] = type
        end
    end

    return x, y, k
end

"""
    draw_sample!(x, y, k, N; rng=Random.default_rng(),
                     beta0=-3.0,
                     beta1=0.8,
                     omega=(4.0, 9.0, 16.0, 25.0, 36.0))

In-place version of `draw_sample`. Mutates pre-allocated arrays `x`, `y`, `k`.
More memory-efficient for Monte Carlo simulations with repeated sampling.
```

```
91
92 # Arguments
93 - `x::AbstractVector{Float64}`: pre-allocated regressor array, length `5N`.
94 - `y::AbstractVector{Float64}`: pre-allocated dependent variable array, length
   ↪  `5N`.
95 - `k::AbstractVector{Int}`: pre-allocated type index array, length `5N`.
96 - `N::Integer`: number of replications per type.
97
98 # Keyword arguments
99 - `rng::AbstractRNG`: random number generator (default: `Random.default_rng()`).
100 - `beta0::Real`: intercept.
101 - `beta1::Real`: slope.
102 - `omega::NTuple{5,<:Real}`: variances `(ω₁,…,ω₅)` for the five types.
103 """
104 function draw_sample!(
105     x::AbstractVector{Float64},
106     y::AbstractVector{Float64},
107     k::AbstractVector{Int},
108     N::Integer;
109     rng::AbstractRNG = Random.default_rng(),
110     beta0::Real = -3.0,
111     beta1::Real = 0.8,
112     omega::NTuple{5,<:Real} = (4.0, 9.0, 16.0, 25.0, 36.0),
113 )
114     N = Int(N)
115     n = 5N
116     @assert length(x) == n && length(y) == n && length(k) == n
117
118     idx = 0
119     @inbounds for type in 1:5
120         σ = sqrt(Float64(omega[type]))
121         for _ in 1:N
122             idx += 1
123             xi = 1.0 + 49.0 * rand(rng)      # U(1,50)
124             ui = σ * randn(rng)              # N(0, ω_type)
125             x[idx] = xi
126             y[idx] = Float64(beta0) + Float64(beta1)*xi + ui
127             k[idx] = type
128         end
129     end
130
131     return nothing
132 end
133
134 """
135     ols_2reg(x, y)
136
```

```
OLS for y = b0 + b1*x with scalar sums. Returns (b0, b1).
"""
function ols_2reg(x::AbstractVector{<:Real}, y::AbstractVector{<:Real})
    n = length(x)
    @assert length(y) == n

    S1  = Float64(n)
    Sx  = 0.0
    Sxx = 0.0
    Sy  = 0.0
    Sxy = 0.0
    @inbounds for i in 1:n
        xi = Float64(x[i]); yi = Float64(y[i])
        Sx  += xi
        Sxx += xi*xi
        Sy  += yi
        Sxy += xi*yi
    end

    det = S1*Sxx - Sx*Sx
    @assert abs(det) > eps(Float64) * (S1 * Sxx + Sx * Sx) "Singular or
    ↪  near-singular design matrix (det ≈ 0)"
    b0 = ( Sxx*Sy - Sx*Sxy) / det
    b1 = (-Sx*Sy + S1*Sxy) / det
    return b0, b1
end

"""
    omega_hat_ols(x, y, k; ols_engine=ols_2reg)

Estimate Ω̂_k by pooling squared OLS residuals within each type k∈{1,…,5}:
    Ω̂_k = (1/N_k) ∑_{i: k_i=k} ê_i^2.
Returns a 5-vector Ω̂.
"""
function omega_hat_ols(
    x::AbstractVector{<:Real},
    y::AbstractVector{<:Real},
    k::AbstractVector{<:Integer};
    ols_engine::Function = ols_2reg
)
    n = length(x)
    @assert length(y) == n
    @assert length(k) == n

    b0, b1 = ols_engine(x, y)

    sse = zeros(Float64, 5)
```

```
    cnt = zeros(Int, 5)

    @inbounds for i in 1:n
        ki = Int(k[i])
        e  = Float64(y[i]) - (b0 + b1*Float64(x[i]))
        sse[ki] += e*e
        cnt[ki] += 1
    end

    ω̂  = Vector{Float64}(undef, 5)
    @inbounds for kk in 1:5
        ω̂[kk] = sse[kk] / cnt[kk]   # pooling within type
        # Groups are assumed to all be populated as in our example.
        # i.e. kk ≥ 1 ∀ kk
    end
    return ω̂
end

"""
    gls_given_omega(x, y, k, ω̂)

Compute GLS for y = b0 + b1*x with Var(u) = Ω ⊗ I (diagonal by type),
given ω̂ = (ω̂_1,…,ω̂_5). Uses weights w_k = 1/ω̂_k.

Returns (b0, b1, A11, A12, A22) where A = X'Σ̂^{-1}X has entries Aij,
useful for inference later without recomputing.
"""
function gls_given_omega(
    x::AbstractVector{<:Real},
    y::AbstractVector{<:Real},
    k::AbstractVector{<:Integer},
    ω̂::AbstractVector{<:Real}
)
    n = length(x)
    @assert length(y) == n
    @assert length(k) == n
    @assert length(ω̂) == 5

    # A = X'Σ̂^{-1}X has entries Aij
    A11 = 0.0; A12 = 0.0; A22 = 0.0
    b1v = 0.0; b2v = 0.0

    @inbounds for i in 1:n
        ki = Int(k[i])
        w  = 1.0 / Float64(ω̂[ki])
        xi = Float64(x[i]); yi = Float64(y[i])

```

```
        A11 += w
        A12 += w*xi
        A22 += w*xi*xi
        b1v += w*yi
        b2v += w*xi*yi
    end

    detA = A11*A22 - A12*A12

    b0 = ( A22*b1v - A12*b2v) / detA
    b1 = (-A12*b1v + A11*b2v) / detA

    return b0, b1, A11, A12, A22
end

"""
    fgls(x, y, k; omega_hat=omega_hat_ols)

Two-step FGLS:
1) Ω̂ ← omega_hat(x,y,k)
2) β̂ ← gls_given_omega(x,y,k,Ω̂ )

Returns (b0, b1, se_b0, se_b1, Ω̂ ) where se_b0 and se_b1 are the standard errors
computed from Var(β̂ ) = (X'Σ̂⁻¹X)⁻¹.
"""
function fgls(
    x::AbstractVector{<:Real},
    y::AbstractVector{<:Real},
    k::AbstractVector{<:Integer};
    omega_hat = omega_hat_ols
)
    Ω̂ = omega_hat(x, y, k)
    b0, b1, A11, A12, A22 = gls_given_omega(x, y, k, Ω̂ )

    # Compute standard errors from (X'Σ̂⁻¹X)⁻¹
    # A = [A11 A12; A12 A22], A⁻¹ = (1/det) * [A22 -A12; -A12 A11]
    detA = A11*A22 - A12*A12
    var_b0 = A22 / detA
    var_b1 = A11 / detA
    se_b0 = sqrt(var_b0)
    se_b1 = sqrt(var_b1)

    return b0, b1, se_b0, se_b1, Ω̂
end

"""    oracle_gls(x, y, k)

```

```
GLS estimator using the true variance structure Ω = (4, 9, 16, 25, 36).

Has the same interface as `fgls` for compatibility with `run_exper`.
Returns (b0, b1, se_b0, se_b1, ω_true) where ω_true are the known true variances.
"""
function oracle_gls(
    x::AbstractVector{<:Real},
    y::AbstractVector{<:Real},
    k::AbstractVector{<:Integer}
)
    # True variance structure from DGP
    ω_true = [4.0, 9.0, 16.0, 25.0, 36.0]

    b0, b1, A11, A12, A22 = gls_given_omega(x, y, k, ω_true)

    # Compute standard errors from (X'Σ⁻¹X)⁻¹
    detA = A11*A22 - A12*A12
    var_b0 = A22 / detA
    var_b1 = A11 / detA
    se_b0 = sqrt(var_b0)
    se_b1 = sqrt(var_b1)

    return b0, b1, se_b0, se_b1, ω_true
end

"""    t_test(estimate, se, null_value)

Constructs t-statistic for testing H₀: θ = null_value.

Returns t = (estimate - null_value) / se.
"""
function t_test(estimate::Real, se::Real, null_value::Real)
    return (Float64(estimate) - Float64(null_value)) / Float64(se)
end


# === MONTE CARLO ===

"""
    run_exper(N, B; β0_true=-3.0, β1_true=0.8, β1_null=0.8, gls_engine=fgls,
    ↪  rng=Random.default_rng())

Run B simulations with DGP parameters (β0_true, β1_true), estimating via specified
↪  GLS method
and computing t-statistics for H₀: β₁ = β1_null.

# Arguments
```

```
- `N::Integer`: sample size is 5N.
- `B::Integer`: number of simulations.

# Keyword arguments
- `β0_true::Real`: true intercept for data generation (default: -3.0).
- `β1_true::Real`: true slope for data generation (default: 0.8).
- `β1_null::Real`: null hypothesis value for β₁ in t-tests (default: 0.8).
- `gls_engine::Function`: estimation function taking (x,y,k) and returning (b0,b1,se_b0,se_b1,Ω ) (default: fgls).
- `rng::AbstractRNG`: random number generator (default: Random.default_rng()).

# Returns
- `NamedTuple`: contains N, β0_true, β1_true, β1_null, β0 (estimates), β1 (estimates), t (statistics).
"""
function run_exper(
    N::Integer,
    B::Integer;
    β0_true::Real = -3.0,
    β1_true::Real = 0.8,
    β1_null::Real = 0.8,
    gls_engine::Function = fgls,
    rng::AbstractRNG = Random.default_rng(),
)
    β0_vec = Vector{Float64}(undef, B)
    β1_vec = Vector{Float64}(undef, B)
    t_vec  = Vector{Float64}(undef, B)

    # Preallocate sample arrays once (reused across all B iterations), i.e. written in pl
    n = 5 * Int(N)
    x = Vector{Float64}(undef, n)
    y = Vector{Float64}(undef, n)
    k = Vector{Int}(undef, n)

    @inbounds for b in 1:B
        draw_sample!(x, y, k, N; rng=rng, beta0=β0_true, beta1=β1_true)
        b0, b1, se_b0, se_b1, _ = gls_engine(x, y, k)
        t = t_test(b1, se_b1, β1_null)

        β0_vec[b] = b0
        β1_vec[b] = b1
        t_vec[b]  = t
    end

    return (N=Int(N), β0_true=Float64(β0_true), β1_true=Float64(β1_true),
            β1_null=Float64(β1_null), β0=β0_vec, β1=β1_vec, t=t_vec)
```

```
end


"""    plot_t_kde(t_vec, N, beta_val; outdir=..., shade=true, alpha=0.05,
    null=true, gls_engine=fgls)

Plot kernel density estimate of t-statistics with standard normal density overlay.

Uses non-parametric KDE to show smooth empirical density. As N increases, the
empirical density becomes smoother and converges to the standard normal.

# Arguments
- `t_vec::AbstractVector{<:Real}`: vector of t-statistics from simulations.
- `N::Integer`: sample size parameter (total sample is 5N).
- `beta_val::Real`: β₁ value used for data generation.

# Keyword arguments
- `outdir::String`: output directory for plot (default:
    script_dir/../out/plots/ej1).
- `shade::Bool`: whether to shade rejection regions (default: true).
- `alpha::Real`: significance level for two-sided test (default: 0.05).
- `null::Bool`: whether null hypothesis is true (default: true). If false,
    auto-scales axes to KDE.
- `gls_engine::Function`: estimation function used (default: fgls). Name is
    appended to filename if not fgls.
"""
function plot_t_kde(t_vec::AbstractVector{<:Real}, N::Integer, beta_val::Real;
                    outdir::String=joinpath(@__DIR__, "../out/plots/ej1"),
                    shade::Bool=true,
                    alpha::Real=0.05,
                    null::Bool=true,
                    gls_engine::Function=fgls)
    mkpath(outdir)

    # Compute kernel density estimate (non-parametric, smooth empirical density)
    kde_result = kde(t_vec)

    # Determine axis ranges based on whether null hypothesis is true
    if null
        # Fixed axis ranges for visual comparison across different N (under H0)
        xlims_range = (-5.0, 5.0)
        ylims_range = (0.0, 0.4)
    else
        # Automatic ranges based on KDE support (when H0 is false)
        xlims_range = (minimum(kde_result.x), maximum(kde_result.x))
        ylims_range = (0.0, maximum(kde_result.density) * 1.1)
    end
```

```
409
410     elements = Any[]
411
412     # Pre-compute rejection rate; shade rejection tails if requested
413     rejection_title = ""
414     if shade
415         z_crit = quantile(Normal(0, 1), 1 - alpha/2)
416         rejection_rate = mean(abs.(t_vec) .> z_crit)
417         rejection_title = @sprintf("Rechazo: %.1f\\%%", rejection_rate * 100)
418
419         # Left tail: manually close polygon to y=0 so pgfplots fills correctly
420         left_idx = kde_result.x .<= -z_crit
421         if any(left_idx)
422             xs, ys = kde_result.x[left_idx], kde_result.density[left_idx]
423             push!(elements, @pgf PGFPlotsX.Plot(
424                 {fill = "gray!30", fill_opacity = 0.4, draw = "none"},
425                 Coordinates([xs; xs[end]; xs[1]], [ys; 0.0; 0.0])))
426         end
427
428         # Right tail
429         right_idx = kde_result.x .>= z_crit
430         if any(right_idx)
431             xs, ys = kde_result.x[right_idx], kde_result.density[right_idx]
432             push!(elements, @pgf PGFPlotsX.Plot(
433                 {fill = "gray!30", fill_opacity = 0.4, draw = "none"},
434                 Coordinates([xs; xs[end]; xs[1]], [ys; 0.0; 0.0])))
435         end
436     end
437
438     # Empirical KDE (solid black)
439     push!(elements, @pgf PGFPlotsX.Plot(
440         {black, solid, line_width = "1.5pt"},
441         Coordinates(collect(kde_result.x), collect(kde_result.density))))
442
443     # Standard normal overlay (dashed gray, semi-transparent)
444     # When null=false this shows the null distribution for reference
445     x_ref = collect(range(xlims_range[1], xlims_range[2], length=200))
446     y_ref = exp.(-x_ref.^2 ./ 2) ./ sqrt(2π)
447     push!(elements, @pgf PGFPlotsX.Plot(
448         {color = "gray", dashed, line_width = "1.5pt", opacity = 0.7},
449         Coordinates(x_ref, y_ref)))
450
451     # Build axis using PGFPlotsX directly for precise dimensional and font control.
452     # width/height set the total figure size including all labels.
453     ax = @pgf Axis(
454         {
455             width  = "227pt",
```

```
            height = "188pt",
            scale_only_axis = false,
            xmin = xlims_range[1], xmax = xlims_range[2],
            ymin = ylims_range[1], ymax = ylims_range[2],
            xlabel = raw"Estadístico $t$",
            ylabel = "Densidad",
            raw"xlabel style={font=\fontsize{10}{12}\selectfont}",
            raw"ylabel style={font=\fontsize{10}{12}\selectfont}",
            raw"tick label style={font=\fontsize{8}{9.6}\selectfont}",
            title = rejection_title,
            raw"title style={font=\fontsize{10}{12}\selectfont, at={(0,1)},
            ↪  anchor=south west, xshift=0pt}",
        },
        elements...
    )

    tp = TikzPicture(ax)

    # Append engine name to filename if not using fgls
    fname_base = "t_N$(N)_beta$(beta_val)"
    if gls_engine !== fgls
        engine_name = string(nameof(gls_engine))
        fname_base *= "_$(engine_name)"
    end

    pgfsave(joinpath(outdir, fname_base * ".tex"), tp; include_preamble=false)
    pgfsave(joinpath(outdir, fname_base * ".pdf"), tp)
end


# === MONTE CARLO EXPERIMENT ORCHESTRATION ===

"""    run_monte_carlo(B; N_values=..., β1_true_values=..., alpha_values=...,
↪  gls_engine=fgls)

Run complete Monte Carlo experiment across all combinations of N and β₁_true.

For each (N, β₁_true) combination:
- Runs B simulations via `run_exper`.
- Computes empirical statistics (mean, median, std) for β₀ and β₁ estimates.
- Computes rejection rates at specified significance levels.
- Optionally generates t-statistic histograms when β₁_true == β₁_null (null
↪  distribution).

# Arguments
- `B::Integer`: number of Monte Carlo simulations per experiment.

```

```
# Keyword arguments
- `N_values::Vector{<:Integer}`: sample size parameters to test (default: [1, 2, 6, 20, 40, 100]).
- `β1_true_values::Vector{<:Real}`: true β₁ values for DGP (default: [0.8, 0.0, 0.4]).
- `alpha_values::Vector{<:Real}`: significance levels for hypothesis tests (default: [0.01, 0.05]).
- `gls_engine::Function`: estimation function taking (x,y,k) and returning (b0,b1,se_b0,se_b1,Ω) (default: fgls).
- `generate_plots::Bool`: whether to generate t-statistic KDE plots (default: true).
- `outdir::String`: output directory for plots (default: script_dir/../out/plots/ej1).
- `rng::AbstractRNG`: random number generator (default: Random.default_rng()).

# Returns
A named tuple with two elements:
- `param_stats`: Vector of NamedTuples containing (N, β1_true, β0_mean, β0_median, β0_std, β1_mean, β1_median, β1_std).
- `rejection_stats`: Vector of NamedTuples containing (N, β1_true, alpha, rejection_rate).

# Notes
- β₀_true is hardcoded to -3.0.
- β₁_null is hardcoded to 0.8 for all hypothesis tests.
- Plots are only generated when β₁_true == β₁_null (i.e., when testing size, not power).
"""
function run_monte_carlo(
    B::Integer;
    N_values::Vector{<:Integer} = [1, 2, 6, 20, 40, 100],
    β1_true_values::Vector{<:Real} = [0.8, 0.0, 0.4],
    alpha_values::Vector{<:Real} = [0.01, 0.05],
    gls_engine::Function = fgls,
    generate_plots::Bool = true,
    outdir::String = joinpath(@__DIR__, "../out/plots/ej1"),
    rng::AbstractRNG = Random.default_rng()
)
    # Hardcoded parameters as specified
    β0_true = -3.0
    β1_null = 0.8

    # Storage for results
    param_stats = []
    rejection_stats = []

```

```
    @info "Starting Monte Carlo experiment with B=$B simulations"

    for N in N_values
        for β1_true in β1_true_values
            @info "Running experiment: N=$N, β₁_true=$β1_true"

            # Run experiment
            results = run_exper(N, B; β0_true=β0_true, β1_true=β1_true,
                                β1_null=β1_null, gls_engine=gls_engine, rng=rng)

            # Compute parameter statistics
            β0_mean = mean(results.β0)
            β0_median = median(results.β0)
            β0_std = std(results.β0)
            β1_mean = mean(results.β1)
            β1_median = median(results.β1)
            β1_std = std(results.β1)

            push!(param_stats, (
                N = N,
                β1_true = β1_true,
                β0_mean = β0_mean,
                β0_median = β0_median,
                β0_std = β0_std,
                β1_mean = β1_mean,
                β1_median = β1_median,
                β1_std = β1_std
            ))

            # Compute rejection rates for each alpha
            for alpha in alpha_values
                z_crit = quantile(Normal(0, 1), 1 - alpha/2)
                rejection_rate = mean(abs.(results.t) .> z_crit)

                push!(rejection_stats, (
                    N = N,
                    β1_true = β1_true,
                    alpha = alpha,
                    z_crit = z_crit,
                    rejection_rate = rejection_rate
                ))
            end

            # Generate t-statistic plot (optional)
            if generate_plots
                @info "Generating t-statistic plot for N=$N, β₁_true=β1_true"
```

```
                plot_t_kde(results.t, N, β1_true; outdir=outdir, null=(β1_true ≈
                ↪  β1_null), gls_engine=gls_engine)
            end
        end
    end

    @info "Monte Carlo experiment complete"

    return (param_stats = param_stats, rejection_stats = rejection_stats)
end


# === VIEWING RESULTS ===

"""    print_param_stats(stats)

Print parameter statistics in a readable tabular format.
"""
function print_param_stats(stats)
    println("\n" * "="^80)
    println("PARAMETER ESTIMATES STATISTICS")
    println("="^80)
    println(@sprintf("%-6s %-10s %-12s %-12s %-12s %-12s %-12s %-12s",
                     "N", "β₁_true", "β₀_mean", "β₀_median", "β₀_std",
                     "β₁_mean", "β₁_median", "β₁_std"))
    println("-"^80)

    for s in stats
        println(@sprintf("%-6d %-10.2f %-12.4f %-12.4f %-12.4f %-12.4f %-12.4f
        ↪  %-12.4f",
                         s.N, s.β1_true, s.β0_mean, s.β0_median, s.β0_std,
                         s.β1_mean, s.β1_median, s.β1_std))
    end
    println("="^80 * "\n")
end

"""    print_rejection_stats(stats)

Print rejection rate statistics in a readable tabular format.
"""
function print_rejection_stats(stats)
    println("\n" * "="^80)
    println("REJECTION RATES")
    println("="^80)
    println(@sprintf("%-6s %-10s %-8s %-12s %-15s",
                     "N", "β₁_true", "α", "z_crit", "Reject Rate"))
    println("-"^80)
```

```
628
629     for s in stats
630         println(@sprintf("%-6d %-10.2f %-8.3f %-12.4f %-15.4f",
631                          s.N, s.β1_true, s.alpha, s.z_crit, s.rejection_rate))
632     end
633     println("="^80 * "\n")
634 end
635
636 """    write_param_stats_latex(stats; filename="param_estimates.tex")
637
638 Write parameter statistics to a LaTeX booktabs table with hierarchical column
    ↪ structure.
639
640 The table organizes results by β₁_true values (columns) and N values (rows), with
641 each major column subdivided by parameter (β̃ ₀, β̃ ₁) and statistic (mean/std,
    ↪ median).
642
643 # Arguments
644 - `stats`: Vector of NamedTuples from `run_monte_carlo` containing parameter
    ↪ statistics.
645
646 # Keyword arguments
647 - `filename::String`: Output filename (default: "param_estimates.tex").
648 - `outdir::String`: Output directory (default: script_dir/../out/tables/ej1).
649 """
650 function write_param_stats_latex(stats;
651                                  filename::String = "param_estimates.tex",
652                                  outdir::String = joinpath(@__DIR__,
                                     ↪ "../out/tables/ej1"))
653     mkpath(outdir)
654
655     # Extract unique values and sort
656     beta1_values = sort(unique([s.β1_true for s in stats]))
657     N_values = sort(unique([s.N for s in stats]))
658
659     # Organize data: data[N][β1_true] = (β0_mean, β0_median, β0_std, β1_mean,
        ↪ β1_median, β1_std)
660     data = Dict()
661     for s in stats
662         if !haskey(data, s.N)
663             data[s.N] = Dict()
664         end
665         data[s.N][s.β1_true] = (s.β0_mean, s.β0_median, s.β0_std,
666                                 s.β1_mean, s.β1_median, s.β1_std)
667     end
668
669     # Format a number for decimal alignment: split at decimal point
```

```
    # Returns (integer_part, decimal_part) as strings
    function split_decimal(x::Real)
        str = @sprintf("%.4f", x)
        parts = split(str, '.')
        if length(parts) == 2
            return (parts[1], parts[2])
        else
            return (str, "0000")
        end
    end

    # Format number with math mode for negative sign
    function fmt_decimal(x::Real)
        int_part, dec_part = split_decimal(x)
        if startswith(int_part, "-")
            int_part = "\$" * int_part * "\$"
        end
        return (int_part, dec_part)
    end

    # Open file and write table
    open(joinpath(outdir, filename), "w") do io
        # Begin tabular with decimal-aligned columns (r@{.}l for each number)
        # Each β1 value has 4 data columns, each split into r@{.}l = 8 actual
        ↪  columns
        println(io, "\\begin{tabular}{c" *
        ↪  "r@{.}lr@{.}lr@{.}lr@{.}l"^length(beta1_values) * "}")
        println(io, "\\toprule")

        # Top level: β₁ values (each spans 8 columns)
        print(io, "")
        for β1 in beta1_values
            print(io, " & \\multicolumn{8}{c}{\$\\beta_1 = ")
            print(io, β1 == 0.0 ? "0" : string(β1))
            print(io, "\$}")
        end
        println(io, " \\\\")

        # Add cmidrules for top level
        for (i, _) in enumerate(beta1_values)
            col_start = 2 + 8*(i-1)
            col_end = col_start + 7
            print(io, "\\cmidrule(lr){$col_start-$col_end} ")
        end
        println(io)

        # Second level: β̃₀ and β̃₁ (each spans 4 columns)
```

```
        print(io, "")
        for _ in beta1_values
            print(io, " & \\multicolumn{4}{c}{\$\\widetilde{\\beta}_{0,
            ↪ \\mathrm{fgls}}\$}")
            print(io, " & \\multicolumn{4}{c}{\$\\widetilde{\\beta}_{1,
            ↪ \\mathrm{fgls}}\$}")
        end
        println(io, " \\\\")

        # Add cmidrules for second level
        for (i, _) in enumerate(beta1_values)
            col_start = 2 + 8*(i-1)
            col_mid = col_start + 3
            col_end = col_start + 7
            print(io, "\\cmidrule(lr){$col_start-$col_mid} ")
            print(io, "\\cmidrule(lr){$(col_mid+1)-$col_end} ")
        end
        println(io)

        # Third level: headers for Media (desvío) and Mediana (each spans 2
        ↪ columns)
        print(io, "\$5N\$")
        for _ in beta1_values
            for _ in 1:2  # β̃₀ and β̃₁
                print(io, " & \\multicolumn{2}{c}{Media} &
                ↪ \\multicolumn{2}{c}{Mediana}")
            end
        end
        println(io, " \\\\")

        # Fourth line: (desvío) under Media columns
        print(io, "")
        for _ in beta1_values
            for _ in 1:2  # β̃₀ and β̃₁
                print(io, " & \\multicolumn{2}{c}{(desvío)} &
                ↪ \\multicolumn{2}{c}{}")
            end
        end
        println(io, " \\\\")

        println(io, "\\midrule")

        # Data rows
        for N in N_values
            # First row: means and medians
            print(io, "$(5*N)")
            for β1 in beta1_values
```

```
                if haskey(data[N], β1)
                    β0_mean, β0_median, β0_std, β1_mean, β1_median, β1_std =
                    ↪  data[N][β1]
                    int0m, dec0m = fmt_decimal(β0_mean)
                    int0med, dec0med = fmt_decimal(β0_median)
                    int1m, dec1m = fmt_decimal(β1_mean)
                    int1med, dec1med = fmt_decimal(β1_median)
                    print(io, " & $int0m & $dec0m & $int0med & $dec0med")
                    print(io, " & $int1m & $dec1m & $int1med & $dec1med")
                else
                    print(io, " & \\multicolumn{2}{c}{--} &
                    ↪  \\multicolumn{2}{c}{--}")
                    print(io, " & \\multicolumn{2}{c}{--} &
                    ↪  \\multicolumn{2}{c}{--}")
                end
            end
            println(io, " \\\\")

            # Second row: standard deviations in parentheses
            print(io, "")
            for β1 in beta1_values
                if haskey(data[N], β1)
                    _, _, β0_std, _, _, β1_std = data[N][β1]
                    int0s, dec0s = split_decimal(β0_std)
                    int1s, dec1s = split_decimal(β1_std)
                    print(io, " & ($int0s & $dec0s) & \\multicolumn{2}{c}{}")
                    print(io, " & ($int1s & $dec1s) & \\multicolumn{2}{c}{}")
                else
                    print(io, " & \\multicolumn{2}{c}{} & \\multicolumn{2}{c}{}")
                    print(io, " & \\multicolumn{2}{c}{} & \\multicolumn{2}{c}{}")
                end
            end
            println(io, " \\\\")
        end

        println(io, "\\bottomrule")
        println(io, "\\end{tabular}")
    end

    @info "LaTeX table written to $(joinpath(outdir, filename))"
end

"""    write_rejection_stats_latex(stats; filename="rejection_rates.tex")

Write rejection rate statistics to a LaTeX booktabs table with hierarchical column
↪  structure.

```

```
The table organizes results by β₁_true values (columns) and N values (rows), with
each major column subdivided by significance level (1% and 5%).

# Arguments
- `stats`: Vector of NamedTuples from `run_monte_carlo` containing rejection
  ↪ statistics.

# Keyword arguments
- `filename::String`: Output filename (default: "rejection_rates.tex").
- `outdir::String`: Output directory (default: script_dir/../out/tables/ej1).
"""
function write_rejection_stats_latex(stats;
                                     filename::String = "rejection_rates.tex",
                                     outdir::String = joinpath(@__DIR__,
                                     ↪ "../out/tables/ej1"))
    mkpath(outdir)

    # Extract unique values and sort
    beta1_values = sort(unique([s.β1_true for s in stats]))
    N_values = sort(unique([s.N for s in stats]))
    alpha_values = sort(unique([s.alpha for s in stats]))

    # Organize data: data[N][β1_true][alpha] = rejection_rate
    data = Dict()
    for s in stats
        if !haskey(data, s.N)
            data[s.N] = Dict()
        end
        if !haskey(data[s.N], s.β1_true)
            data[s.N][s.β1_true] = Dict()
        end
        data[s.N][s.β1_true][s.alpha] = s.rejection_rate
    end

    # Format rejection rate as percentage with smart rounding
    # Strips trailing zeros: 100.00 → 100, 97.50 → 97.5, 97.06 → 97.06
    function split_decimal_rate(x::Real, include_pct::Bool=true)
        pct = x * 100.0
        str = @sprintf("%.2f", pct)
        # Strip trailing zeros and decimal point if integer
        str = replace(str, r"\.?0+$" ⇒ "")
        # Split at decimal point (if still present)
        parts = split(str, '.')
        if length(parts) == 2
            # Has decimal part - include the dot in the decimal part
            if include_pct
                return (parts[1], "." * parts[2] * "\\%")
```

```
            else
                return (parts[1], "." * parts[2])
            end
        else
            # No decimal point (integer) - empty decimal part
            if include_pct
                return (str, "\\%")
            else
                return (str, "")
            end
        end
    end

    # Open file and write table
    open(joinpath(outdir, filename), "w") do io
        # Each β1 value has 2 significance levels, each split into r@{}l = 4 actual
        ↪  columns
        println(io, "\\begin{tabular}{c" * "r@{}lr@{}l"^length(beta1_values) * "}")
        println(io, "\\toprule")

        # Top level: β₁ values (each spans 4 columns)
        print(io, "")
        for β1 in beta1_values
            print(io, " & \\multicolumn{4}{c}{\$\\beta_1 = ")
            print(io, β1 == 0.0 ? "0" : string(β1))
            print(io, "\$}")
        end
        println(io, " \\\\")

        # Add cmidrules for top level
        for (i, _) in enumerate(beta1_values)
            col_start = 2 + 4*(i-1)
            col_end = col_start + 3
            print(io, "\\cmidrule(lr){$col_start-$col_end} ")
        end
        println(io)

        # Second level: significance levels (each spans 2 columns)
        print(io, "\$5N\$")
        for _ in beta1_values
            print(io, " & \\multicolumn{2}{c}{\$1\\%\$}")
            print(io, " & \\multicolumn{2}{c}{\$5\\%\$}")
        end
        println(io, " \\\\")

        println(io, "\\midrule")

```

```
        # Data rows
        for (row_idx, N) in enumerate(N_values)
            print(io, "$(5*N)")
            is_first_row = (row_idx == 1)

            for β1 in beta1_values
                for α in alpha_values
                    if haskey(data[N], β1) && haskey(data[N][β1], α)
                        rate = data[N][β1][α]
                        int_part, dec_part = split_decimal_rate(rate, is_first_row)
                        print(io, " & $int_part & $dec_part")
                    else
                        print(io, " & \\multicolumn{2}{c}{--}")
                    end
                end
            end
            println(io, " \\\\")
        end

        println(io, "\\bottomrule")
        println(io, "\\end{tabular}")
    end

    @info "LaTeX table written to $(joinpath(outdir, filename))"
end

"""    write_param_stats_comparison_latex(stats_fgls, stats_gls;
↪  filename="param_estimates_comparison.tex")

Write parameter statistics comparison between FGLS and GLS to a LaTeX booktabs
↪  table.

The table organizes results by estimation method (columns) and N values (rows),
↪  with
each major column subdivided by parameter (β̃₀, β̃₁) and statistic (mean/std,
↪  median).
Only includes results for β₁_true = 0.8.

# Arguments
- `stats_fgls`: Vector of NamedTuples from `run_monte_carlo` with FGLS results.
- `stats_gls`: Vector of NamedTuples from `run_monte_carlo` with GLS results.

# Keyword arguments
- `filename::String`: Output filename (default: "param_estimates_comparison.tex").
- `outdir::String`: Output directory (default: script_dir/../out/tables/ej1).
"""
function write_param_stats_comparison_latex(stats_fgls, stats_gls;
```

```
                                        filename::String =
                                        ↪  "param_estimates_comparison.tex",
                                        outdir::String = joinpath(@__DIR__,
                                        ↪  "../out/tables/ej1"))
    mkpath(outdir)

    # Filter for β₁_true = 0.8 and extract N values
    stats_fgls_filtered = filter(s -> s.β1_true ≈ 0.8, stats_fgls)
    stats_gls_filtered = filter(s -> s.β1_true ≈ 0.8, stats_gls)
    N_values = sort(unique([s.N for s in stats_fgls_filtered]))

    # Organize data: data[method][N] = (β0_mean, β0_median, β0_std, β1_mean,
    ↪  β1_median, β1_std)
    data = Dict("FGLS" => Dict(), "GLS" => Dict())

    for s in stats_fgls_filtered
        data["FGLS"][s.N] = (s.β0_mean, s.β0_median, s.β0_std, s.β1_mean,
        ↪  s.β1_median, s.β1_std)
    end
    for s in stats_gls_filtered
        data["GLS"][s.N] = (s.β0_mean, s.β0_median, s.β0_std, s.β1_mean,
        ↪  s.β1_median, s.β1_std)
    end

    methods = ["FGLS", "GLS"]

    # Format a number for decimal alignment: split at decimal point
    # Returns (integer_part, decimal_part) as strings
    function split_decimal(x::Real)
        str = @sprintf("%.4f", x)
        parts = split(str, '.')
        if length(parts) == 2
            return (parts[1], parts[2])
        else
            return (str, "0000")
        end
    end

    # Format number with math mode for negative sign
    function fmt_decimal(x::Real)
        int_part, dec_part = split_decimal(x)
        if startswith(int_part, "-")
            int_part = "\$" * int_part * "\$"
        end
        return (int_part, dec_part)
    end

```

```
    # Open file and write table
    open(joinpath(outdir, filename), "w") do io
        # Begin tabular with decimal-aligned columns (r@{.}l for each number)
        # Each method has 4 data columns, each split into r@{.}l = 8 actual columns
        println(io, "\\begin{tabular}{c" *
        ↪  "r@{.}lr@{.}lr@{.}lr@{.}l"^length(methods) * "}")
        println(io, "\\toprule")

        # Top level: Method names (each spans 8 columns)
        print(io, "")
        for method in methods
            print(io, " & \\multicolumn{8}{c}{$method}")
        end
        println(io, " \\\\")

        # Add cmidrules for top level
        for (i, _) in enumerate(methods)
            col_start = 2 + 8*(i-1)
            col_end = col_start + 7
            print(io, "\\cmidrule(lr){$col_start-$col_end} ")
        end
        println(io)

        # Second level: β̃₀ and β̃₁ (each spans 4 columns)
        print(io, "")
        for _ in methods
            print(io, " & \\multicolumn{4}{c}{\$\\widetilde{\\beta}_0\$}")
            print(io, " & \\multicolumn{4}{c}{\$\\widetilde{\\beta}_1\$}")
        end
        println(io, " \\\\")

        # Add cmidrules for second level
        for (i, _) in enumerate(methods)
            col_start = 2 + 8*(i-1)
            col_mid = col_start + 3
            col_end = col_start + 7
            print(io, "\\cmidrule(lr){$col_start-$col_mid} ")
            print(io, "\\cmidrule(lr){$(col_mid+1)-$col_end} ")
        end
        println(io)

        # Third level: headers for Media (desvío) and Mediana (each spans 2
        ↪  columns)
        print(io, "\$5N\$")
        for _ in methods
            for _ in 1:2  # β̃₀ and β̃₁
```

```
            print(io, " & \\multicolumn{2}{c}{Media} & \\multicolumn{2}{c}{Mediana}")
        end
    end
    println(io, " \\\\")

    # Fourth line: (desvío) under Media columns
    print(io, "")
    for _ in methods
        for _ in 1:2  # β̃₀ and β̃₁
            print(io, " & \\multicolumn{2}{c}{(desvío)} & \\multicolumn{2}{c}{}")
        end
    end
    println(io, " \\\\")

    println(io, "\\midrule")

    # Data rows
    for N in N_values
        # First row: means and medians
        print(io, "$(5*N)")
        for method in methods
            if haskey(data[method], N)
                β0_mean, β0_median, β0_std, β1_mean, β1_median, β1_std = data[method][N]
                int0m, dec0m = fmt_decimal(β0_mean)
                int0med, dec0med = fmt_decimal(β0_median)
                int1m, dec1m = fmt_decimal(β1_mean)
                int1med, dec1med = fmt_decimal(β1_median)
                print(io, " & $int0m & $dec0m & $int0med & $dec0med")
                print(io, " & $int1m & $dec1m & $int1med & $dec1med")
            else
                print(io, " & \\multicolumn{2}{c}{--} & \\multicolumn{2}{c}{--}")
                print(io, " & \\multicolumn{2}{c}{--} & \\multicolumn{2}{c}{--}")
            end
        end
        println(io, " \\\\")

        # Second row: standard deviations in parentheses
        print(io, "")
        for method in methods
            if haskey(data[method], N)
                _, _, β0_std, _, _, β1_std = data[method][N]
                int0s, dec0s = split_decimal(β0_std)
```

```
                    int1s, dec1s = split_decimal(β1_std)
                    print(io, " & ($int0s & $dec0s) & \\multicolumn{2}{c}{}")
                    print(io, " & ($int1s & $dec1s) & \\multicolumn{2}{c}{}")
                else
                    print(io, " & \\multicolumn{2}{c}{} & \\multicolumn{2}{c}{}")
                    print(io, " & \\multicolumn{2}{c}{} & \\multicolumn{2}{c}{}")
                end
            end
            println(io, " \\\\")
        end

        println(io, "\\bottomrule")
        println(io, "\\end{tabular}")
    end

    @info "LaTeX table written to $(joinpath(outdir, filename))"
end

"""    write_rejection_stats_comparison_latex(stats_fgls, stats_gls;
↪  filename="rejection_rates_comparison.tex")

Write rejection rate statistics comparison between FGLS and GLS to a LaTeX booktabs
↪  table.

The table organizes results by estimation method (columns) and N values (rows),
↪  with
each major column subdivided by significance level (1% and 5%).
Only includes results for β₁_true = 0.8.

# Arguments
- `stats_fgls`: Vector of NamedTuples from `run_monte_carlo` with FGLS results.
- `stats_gls`: Vector of NamedTuples from `run_monte_carlo` with GLS results.

# Keyword arguments
- `filename::String`: Output filename (default: "rejection_rates_comparison.tex").
- `outdir::String`: Output directory (default: script_dir/../out/tables/ej1).
"""
function write_rejection_stats_comparison_latex(stats_fgls, stats_gls;
                                                filename::String =
                                                ↪  "rejection_rates_comparison.tex",
                                                outdir::String = joinpath(@__DIR__,
                                                ↪  "../out/tables/ej1"))
    mkpath(outdir)

    # Filter for β₁_true = 0.8
    stats_fgls_filtered = filter(s -> s.β1_true ≈ 0.8, stats_fgls)
    stats_gls_filtered = filter(s -> s.β1_true ≈ 0.8, stats_gls)
```

```
1104
1105    N_values = sort(unique([s.N for s in stats_fgls_filtered]))
1106    alpha_values = sort(unique([s.alpha for s in stats_fgls_filtered]))
1107
1108    # Organize data: data[method][N][alpha] = rejection_rate
1109    data = Dict("FGLS" => Dict(), "GLS" => Dict())
1110
1111    for s in stats_fgls_filtered
1112        if !haskey(data["FGLS"], s.N)
1113            data["FGLS"][s.N] = Dict()
1114        end
1115        data["FGLS"][s.N][s.alpha] = s.rejection_rate
1116    end
1117
1118    for s in stats_gls_filtered
1119        if !haskey(data["GLS"], s.N)
1120            data["GLS"][s.N] = Dict()
1121        end
1122        data["GLS"][s.N][s.alpha] = s.rejection_rate
1123    end
1124
1125    methods = ["FGLS", "GLS"]
1126
1127    # Format rejection rate as percentage with smart rounding
1128    # Strips trailing zeros: 100.00 → 100, 97.50 → 97.5, 97.06 → 97.06
1129    function split_decimal_rate(x::Real, include_pct::Bool=true)
1130        pct = x * 100.0
1131        str = @sprintf("%.2f", pct)
1132        # Strip trailing zeros and decimal point if integer
1133        str = replace(str, r"\.?0+$" => "")
1134        # Split at decimal point (if still present)
1135        parts = split(str, '.')
1136        if length(parts) == 2
1137            # Has decimal part - include the dot in the decimal part
1138            if include_pct
1139                return (parts[1], "." * parts[2] * "\\%")
1140            else
1141                return (parts[1], "." * parts[2])
1142            end
1143        else
1144            # No decimal point (integer) - empty decimal part
1145            if include_pct
1146                return (str, "\\%")
1147            else
1148                return (str, "")
1149            end
1150        end
```

```
    end

    # Open file and write table
    open(joinpath(outdir, filename), "w") do io
        # Each method has 2 significance levels, each split into r@{}l = 4 actual
        ↪ columns
        println(io, "\\begin{tabular}{c" * "r@{}lr@{}l"^length(methods) * "}")
        println(io, "\\toprule")

        # Top level: Method names (each spans 4 columns)
        print(io, "")
        for method in methods
            print(io, " & \\multicolumn{4}{c}{$method}")
        end
        println(io, " \\\\")

        # Add cmidrules for top level
        for (i, _) in enumerate(methods)
            col_start = 2 + 4*(i-1)
            col_end = col_start + 3
            print(io, "\\cmidrule(lr){$col_start-$col_end} ")
        end
        println(io)

        # Second level: significance levels (each spans 2 columns)
        print(io, "\$5N\$")
        for _ in methods
            print(io, " & \\multicolumn{2}{c}{\$1\\%\$}")
            print(io, " & \\multicolumn{2}{c}{\$5\\%\$}")
        end
        println(io, " \\\\")

        println(io, "\\midrule")

        # Data rows
        for (row_idx, N) in enumerate(N_values)
            print(io, "$(5*N)")
            is_first_row = (row_idx == 1)

            for method in methods
                for α in alpha_values
                    if haskey(data[method], N) && haskey(data[method][N], α)
                        rate = data[method][N][α]
                        int_part, dec_part = split_decimal_rate(rate, is_first_row)
                        print(io, " & $int_part & $dec_part")
                    else
                        print(io, " & \\multicolumn{2}{c}{--}")
```

```
                    end
                end
            end
            println(io, " \\\\")
        end

        println(io, "\\bottomrule")
        println(io, "\\end{tabular}")
    end

    @info "LaTeX table written to $(joinpath(outdir, filename))"
end


# === REJECTION RATE BY SAMPLE SIZE PLOTS ===

"""
    plot_rejection_rates_by_n(B;
                              N_grid=1:1:100,
                              β1_values=[0.0, 0.4, 0.8],
                              alpha_values=[0.01, 0.05],
                              gls_engine=fgls,
                              outdir=joinpath(@__DIR__, "../out/plots/ej1"))

Generate plots showing rejection rates as a function of sample size for GLS t-test.

Creates one plot per significance level, with separate lines for each β₁ value.
Each plot shows dots at data points connected by lines, plus a horizontal reference
line at the nominal significance level.

# Arguments
- `B::Integer`: Number of Monte Carlo replications per (N, β₁) combination.

# Keyword arguments
- `N_grid`: Sample sizes to test (default: 1:1:100), where n = 5N.
- `β1_values::Vector{<:Real}`: True β₁ values to test (default: [0.0, 0.4, 0.8]).
- `alpha_values::Vector{<:Real}`: Significance levels for plots (default: [0.01,
  ↪ 0.05]).
- `gls_engine::Function`: estimation function (default: fgls).
- `outdir::String`: Output directory for plots.
"""
function plot_rejection_rates_by_n(
    B::Integer;
    N_grid = 1:1:100,
    β1_values::Vector{<:Real} = [0.0, 0.4, 0.8],
    alpha_values::Vector{<:Real} = [0.01, 0.05],
    gls_engine::Function = fgls,
```

```
    outdir::String = joinpath(@__DIR__, "../out/plots/ej1")
)
    # Run Monte Carlo with specified N-grid
    @info "Running Monte Carlo for rejection rate plots (N = $(N_grid[1]):$(step(N_grid)):$(N_grid[end]))"
    N_vals = collect(N_grid)
    results = run_monte_carlo(
        B;
        N_values = N_vals,
        β1_true_values = β1_values,
        alpha_values = alpha_values,
        gls_engine = gls_engine,
        generate_plots = false
    )
    rejection_stats = results.rejection_stats

    # Extract rejection rates for a given α level
    function extract_rates_by_beta(stats, N_vals, α_target)
        betas = sort(unique([s.β1_true for s in stats]))
        rates = Dict()
        for β in betas
            rates[β] = Float64[]
            for N in N_vals
                idx = findfirst(s -> s.N == N && s.β1_true ≈ β && s.alpha ≈ α_target, stats)
                if idx !== nothing
                    push!(rates[β], stats[idx].rejection_rate)
                else
                    push!(rates[β], NaN)
                end
            end
        end
        return rates
    end

    # Plot styling
    beta_labels = Dict(0.8 ⇒ "β₁ = 0.8 (H₀ verdadera)",
                       0.4 ⇒ "β₁ = 0.4",
                       0.0 ⇒ "β₁ = 0.0")
    beta_colors = Dict(0.8 ⇒ :blue, 0.4 ⇒ :red, 0.0 ⇒ :green)
    beta_markers = Dict(0.8 ⇒ :circle, 0.4 ⇒ :square, 0.0 ⇒ :diamond)

    mkpath(outdir)

    # Create one plot for each significance level
    for α_level in alpha_values
        @info "Creating rejection rate plot for α = $α_level"
```

```
1288
1289        rates_by_beta = extract_rates_by_beta(rejection_stats, N_vals, α_level)
1290
1291        # Convert N to n (n = 5N) for x-axis
1292        n_vals = 5 .* N_vals
1293
1294        # Create plot
1295        p = plot(
1296            xlabel = L"Tamaño de muestra ($n$)",
1297            ylabel = "Tasa de rechazo",
1298            legend = :topright,
1299            size = (600, 400),
1300            ylims = (0, 1),
1301            grid = true
1302        )
1303
1304        # Plot each β value as dots connected by lines
1305        for β in sort(collect(keys(rates_by_beta)), rev=true)  # Sort descending so
            ↪  0.8 appears first
1306            plot!(p, n_vals, rates_by_beta[β],
1307                  label = beta_labels[β],
1308                  color = beta_colors[β],
1309                  marker = beta_markers[β],
1310                  markersize = 4,
1311                  linewidth = 2,
1312                  linestyle = :solid)
1313        end
1314
1315        # Add horizontal reference line at nominal significance level
1316        hline!(p, [α_level],
1317               linestyle = :dash,
1318               color = :gray,
1319               alpha = 0.5,
1320               linewidth = 1.5,
1321               label = "")
1322
1323        # Save plot with α in filename
1324        α_str = replace(string(α_level), "." ⇒ "")
1325        fname_base = "rejection_rate_by_n_alpha$(α_str)"
1326        savefig(p, joinpath(outdir, fname_base * ".tex"))
1327        savefig(p, joinpath(outdir, fname_base * ".pdf"))
1328
1329        @info "Plot saved: $(fname_base).pdf"
1330    end
1331
1332    @info "All rejection rate plots saved to $outdir"
1333 end
```

```
# === HEATMAP: REJECTION RATE AS FUNCTION OF β₁ AND N ===

"""
    plot_rejection_rate_heatmap(B;
                                β1_grid=0.70:0.01:0.90,
                                N_grid=1:2:100,
                                test_alpha=0.01,
                                gls_engine=fgls,
                                resolution=1.0,
                                outdir=joinpath(@__DIR__, "../out/plots/ej1"))

Generate heatmap showing empirical rejection rates as a function of β₁ and sample size.

Creates a heatmap with n (sample size) on x-axis, β₁ on y-axis, and color encoding
the empirical rejection rate at each grid point. The human visual system perceives the
implied iso-rejection contours from the color gradient directly.

# Arguments
- `B::Integer`: Number of Monte Carlo replications per (N, β₁) combination.

# Keyword arguments
- `β1_grid`: Grid of β₁ values to test (default: 0.70:0.01:0.90).
- `N_grid`: Grid of N values to test, where n = 5N (default: 1:2:100).
- `test_alpha::Real`: Significance level for the test (default: 0.01).
- `gls_engine::Function`: estimation function (default: fgls).
- `resolution::Real`: Resolution multiplier - higher values create finer grids (default: 1.0).
- `outdir::String`: Output directory for plots.
"""
function plot_rejection_rate_heatmap(
    B::Integer;
    β1_grid = 0.70:0.01:0.90,
    N_grid = 1:2:100,
    test_alpha::Real = 0.01,
    gls_engine::Function = fgls,
    resolution::Real = 1.0,
    outdir::String = joinpath(@__DIR__, "../out/plots/ej1")
)
    # Apply resolution scaling to grids
    if resolution != 1.0
        β1_min, β1_max = extrema(β1_grid)
        β1_step = step(β1_grid) / resolution
        β1_grid = β1_min:β1_step:β1_max
```

```
        N_min, N_max = extrema(N_grid)
        N_step = max(1, round(Int, step(N_grid) / resolution))
        N_grid = N_min:N_step:N_max
    end

    # Ensure N ≥ 1 (so n = 5N ≥ 5)
    N_min = first(N_grid)
    if N_min < 1
        error("N must be at least 1 (n = 5N must be ≥ 5)")
    end

    # Convert ranges to vectors
    β1_vals = collect(β1_grid)
    N_vals = collect(N_grid)

    @info "Running Monte Carlo for contour plot"
    @info "  β₁ grid: $(length(β1_vals)) values from $(β1_vals[1]) to
    ↪  $(β1_vals[end])"
    @info "  N grid: $(length(N_vals)) values from $(N_vals[1]) to $(N_vals[end])"
    @info "  Total experiments: $(length(β1_vals) × length(N_vals))"

    # Run Monte Carlo on the grid
    results = run_monte_carlo(
        B;
        N_values = N_vals,
        β1_true_values = β1_vals,
        alpha_values = [test_alpha],
        gls_engine = gls_engine,
        generate_plots = false
    )
    rejection_stats = results.rejection_stats

    # Build rejection rate matrix: rows = N, cols = β₁
    rejection_matrix = Matrix{Float64}(undef, length(N_vals), length(β1_vals))

    for (i, N) in enumerate(N_vals)
        for (j, β1) in enumerate(β1_vals)
            idx = findfirst(s -> s.N == N && s.β1_true ≈ β1 && s.alpha ≈
            ↪  test_alpha, rejection_stats)
            if idx !== nothing
                rejection_matrix[i, j] = rejection_stats[idx].rejection_rate
            else
                rejection_matrix[i, j] = NaN
            end
        end
    end
```

```
    # Convert N to n for plotting
    n_vals = 5 .* N_vals

    # Heatmap: color encodes empirical rejection rate at each (n, β₁) grid point
    # Compute x-axis ticks so the minimum n value is always labelled
    x_tick_step = max(10, ceil(Int, (last(n_vals) - first(n_vals)) / 5 / 10) * 10)
    x_ticks = sort(unique(vcat(first(n_vals),
    ↪  collect(first(n_vals)+x_tick_step:x_tick_step:last(n_vals)))))

    p = heatmap(
        n_vals, β1_vals, rejection_matrix',
        xlabel = L"Tamaño de muestra ($5N$)",
        ylabel = L"Valor verdadero de $\beta_1$",
        title = "",
        colorbar_title = "",
        color = cgrad([:black, :white]),
        size = (700, 500),
        clims = (0.0, 1.0),
        grid = false,
        framestyle = :box,
        xticks = x_ticks
    )

    # Mark the null hypothesis value β₁ = 0.8 for reference
    hline!(p, [0.8],
           linestyle = :dash,
           color = :white,
           linewidth = 0.8,
           label = "",
           alpha = 0.9)

    mkpath(outdir)

    α_str = replace(string(test_alpha), "." ⇒ "")
    fname_base = "rejection_rate_heatmap_alpha$(α_str)"
    savefig(p, joinpath(outdir, fname_base * ".tex"))
    savefig(p, joinpath(outdir, fname_base * ".pdf"))

    @info "Heatmap saved: $(fname_base).pdf"
    @info "Plot saved to $outdir"
end


# === EXECUTION ===

# Generate high resolution heatmap (used in Figure ej-1-rejection-rate-heatmap)
```

```
plot_rejection_rate_heatmap(10000, resolution = 20.0)

# Run with FGLS (default) - generates t-statistic plots for FGLS
results_fgls = run_monte_carlo(5000, )

# Run with GLS (true omega) - generates t-statistic plots for GLS
results_gls = run_monte_carlo(5000, gls_engine=oracle_gls, generate_plots=true)

# Generate comparison tables (FGLS vs GLS for β₁_true = 0.8)
write_param_stats_comparison_latex(results_fgls.param_stats,
    ↪ results_gls.param_stats)
write_rejection_stats_comparison_latex(results_fgls.rejection_stats,
    ↪ results_gls.rejection_stats)

# Generate main FGLS tables
param_stats = results_fgls.param_stats
rejection_stats = results_fgls.rejection_stats
write_param_stats_latex(param_stats)
write_rejection_stats_latex(rejection_stats)
```

A.2. Ejercicio 2. El contenido del archivo `ej2.jl` es el siguiente.

```
# ================================================================
# TRABAJO FINAL ECONOMETRIA
# Ejercicio 2: Test de Heterocedasticidad de White
#
# Autor: Felipe Tappata
# Fecha: Febrero 2026
# ================================================================

# Load dependencies
using Random, Statistics
using LinearAlgebra
using Printf, LaTeXStrings
using Distributions
using StaticArrays
using Plots, KernelDensity
import PGFPlotsX

# Set plotting backend
function __init_plots__()
    pgfplotsx()
    default(tex_output_standalone=false)
end
__init_plots__()

# Set seed
```

```julia
const SEED = 8869
Random.seed!(SEED)



# === DATA GENERATION ===

# x₁ base pattern: 18 evenly-spaced points in [-1,1] plus endpoints -1.1 and 1.1
const X1_BASE = vcat(-1.1, range(-1.0, 1.0, length=18), 1.1)

"""
    draw_sample(n::Integer, design::Integer;
                beta::AbstractVector{<:Real} = [1.0, 1.0, 1.0],
                rng::AbstractRNG = Random.default_rng())

Generate one sample from the heteroskedastic model:
    yᵢ = β₀ + β₁x₁ᵢ + β₂x₂ᵢ + √νᵢ uᵢ

# Designs:
- Design 0: uᵢ ~ N(0,1), νᵢ = 1 (homoskedastic normal)
- Design 1: uᵢ ~ N(0,1), νᵢ = exp(0.25x₁ᵢ + 0.25x₂ᵢ) (heteroskedastic normal)
- Design 2: uᵢ ~ t₅, νᵢ = exp(0.25x₁ᵢ + 0.25x₂ᵢ) (heteroskedastic non-normal)

# Arguments
- `n::Integer`: sample size, must be a multiple of 20.
- `design::Integer`: design number (0, 1, or 2).

# Keyword arguments
- `beta::AbstractVector{<:Real}`: coefficient vector [β₀, β₁, β₂] (default: [1.0, 1.0, 1.0]).
- `rng::AbstractRNG`: random number generator (default: Random.default_rng()).

# Returns
- `X::Matrix{Float64}`: design matrix with intercept column (n × 3).
- `y::Vector{Float64}`: dependent variable (length n).

# Notes
x₁: 18 evenly-spaced points in [-1,1] plus endpoints -1.1 and 1.1 (deterministic); repeats m = n/20 times.
x₂: 20 fresh random draws from standard normal per repetition (n draws total).
"""
function draw_sample(
    n::Integer,
    design::Integer;
    beta::AbstractVector{<:Real} = [1.0, 1.0, 1.0],
    rng::AbstractRNG = Random.default_rng(),
)
```

```
    # Validate inputs
    @assert n % 20 == 0 "Sample size n must be a multiple of 20"
    @assert design ∈ (0, 1, 2) "Design must be 0, 1, or 2"
    @assert length(beta) == 3 "beta must have 3 elements: [β₀, β₁, β₂]"

    n = Int(n)
    m = div(n, 20)  # number of repetitions

    # Allocate output
    X = Matrix{Float64}(undef, n, 3)
    y = Vector{Float64}(undef, n)

    # Fill design matrix
    X[:, 1] .= 1.0  # intercept

    # Fill x₁ and x₂ columns: x₁ repeats deterministic pattern, x₂ draws fresh
    ↪ N(0,1) each rep
    @inbounds for rep in 1:m
        idx_start = (rep - 1) * 20 + 1
        idx_end = rep * 20
        X[idx_start:idx_end, 2] .= X1_BASE
        rand!(rng, Normal(0, 1), view(X, idx_start:idx_end, 3))
    end

    # Generate y based on design
    β₀, β₁, β₂ = Float64.(beta)

    @inbounds for i in 1:n
        x1i = X[i, 2]
        x2i = X[i, 3]

        # Compute ν (variance scaling)
        if design == 0
            ν = 1.0
        else  # design == 1 or 2
            ν = exp(0.25 * x1i + 0.25 * x2i)
        end

        # Generate error term u
        if design == 2
            # t₅ distribution (non-normal)
            u = rand(rng, TDist(5))
        else  # design == 0 or 1
            # Standard normal
            u = randn(rng)
        end

```

```
        # Compute y
        y[i] = β₀ + β₁ * x1i + β₂ * x2i + sqrt(ν) * u
    end

    return X, y
end

"""
    draw_sample!(X, y, n, design;
                 beta::AbstractVector{<:Real} = [1.0, 1.0, 1.0],
                 rng::AbstractRNG = Random.default_rng())

In-place version of `draw_sample`. Mutates pre-allocated arrays `X` and `y`.
More memory-efficient for Monte Carlo simulations with repeated sampling.

# Arguments
- `X::AbstractMatrix{Float64}`: pre-allocated design matrix (n × 3).
- `y::AbstractVector{Float64}`: pre-allocated dependent variable array (length n).
- `n::Integer`: sample size, must be a multiple of 20.
- `design::Integer`: design number (0, 1, or 2).

# Keyword arguments
- `beta::AbstractVector{<:Real}`: coefficient vector [β₀, β₁, β₂] (default: [1.0,
  ↪ 1.0, 1.0]).
- `rng::AbstractRNG`: random number generator (default: Random.default_rng()).
"""
function draw_sample!(
    X::AbstractMatrix{Float64},
    y::AbstractVector{Float64},
    n::Integer,
    design::Integer;
    beta::AbstractVector{<:Real} = [1.0, 1.0, 1.0],
    rng::AbstractRNG = Random.default_rng(),
)
    # Validate inputs
    @assert n % 20 == 0 "Sample size n must be a multiple of 20"
    @assert design ∈ (0, 1, 2) "Design must be 0, 1, or 2"
    @assert length(beta) == 3 "beta must have 3 elements: [β₀, β₁, β₂]"
    @assert size(X) == (n, 3) "X must be n × 3"
    @assert length(y) == n "y must have length n"

    n = Int(n)
    m = div(n, 20)  # number of repetitions

    # Fill design matrix
    X[:, 1] .= 1.0  # intercept

```

```
    # Fill x₁ and x₂ columns: x₁ repeats deterministic pattern, x₂ draws fresh N(0,1) each rep
    @inbounds for rep in 1:m
        idx_start = (rep - 1) * 20 + 1
        idx_end = rep * 20
        X[idx_start:idx_end, 2] .= X1_BASE
        rand!(rng, Normal(0, 1), view(X, idx_start:idx_end, 3))
    end

    # Generate y based on design
    β₀, β₁, β₂ = Float64.(beta)

    @inbounds for i in 1:n
        x1i = X[i, 2]
        x2i = X[i, 3]

        # Compute ν (variance scaling)
        if design == 0
            ν = 1.0
        else  # design == 1 or 2
            ν = exp(0.25 * x1i + 0.25 * x2i)
        end

        # Generate error term u
        if design == 2
            # t₅ distribution (non-normal)
            u = rand(rng, TDist(5))
        else  # design == 0 or 1
            # Standard normal
            u = randn(rng)
        end

        # Compute y
        y[i] = β₀ + β₁ * x1i + β₂ * x2i + sqrt(ν) * u
    end

    return nothing
end


# === OLS ESTIMATION ===

"""
    ols_fit_k3(X, y)

Fit OLS regression y = Xβ + u using StaticArrays (stack-allocated) for k=3.
Computes X'X and X'y via scalar accumulation, then
```

```
stores in stack-allocated SMatrix/SVector for zero heap allocations.

# Arguments
- `X::AbstractMatrix{<:Real}`: design matrix (n × 3), including intercept column.
- `y::AbstractVector{<:Real}`: dependent variable (length n).

# Returns
- `β̂::Vector{Float64}`: OLS coefficient estimates (length 3).
- `ê::Vector{Float64}`: OLS residuals (length n).
"""
function ols_fit_k3(X::AbstractMatrix{<:Real}, y::AbstractVector{<:Real})
    n = length(y)
    @assert size(X, 1) == n
    @assert size(X, 2) == 3 "ols_fit_k3 optimized for k=3 (intercept + 2
    ↪  regressors)"

    # Compute X'X and X'y via scalar accumulation (no intermediate allocations)
    # X'X is symmetric, so we only compute upper triangle
    S11 = 0.0; S12 = 0.0; S13 = 0.0
    S22 = 0.0; S23 = 0.0; S33 = 0.0
    Sy1 = 0.0; Sy2 = 0.0; Sy3 = 0.0

    @inbounds for i in 1:n
        x1 = Float64(X[i, 1])
        x2 = Float64(X[i, 2])
        x3 = Float64(X[i, 3])
        yi = Float64(y[i])

        # Accumulate X'X (symmetric)
        S11 += x1 * x1
        S12 += x1 * x2
        S13 += x1 * x3
        S22 += x2 * x2
        S23 += x2 * x3
        S33 += x3 * x3

        # Accumulate X'y
        Sy1 += x1 * yi
        Sy2 += x2 * yi
        Sy3 += x3 * yi
    end

    # Build STACK-ALLOCATED 3×3 matrix using SMatrix
    XtX = @SMatrix [S11 S12 S13;
                    S12 S22 S23;
                    S13 S23 S33]

```

```
    Xty = @SVector [Sy1, Sy2, Sy3]

    # Solve via Cholesky factorization (StaticArrays has optimized linear algebra)
    C = cholesky(XtX)
    β̂_static = C \ Xty

    # Convert back to regular Vector for consistency
    β̂ = Vector{Float64}(β̂_static)

    # Compute residuals manually (must allocate this since n varies)
    ê = Vector{Float64}(undef, n)
    @inbounds for i in 1:n
        ê[i] = Float64(y[i]) - (β̂[1] * Float64(X[i, 1]) +
                                β̂[2] * Float64(X[i, 2]) +
                                β̂[3] * Float64(X[i, 3]))
    end

    return β̂, ê
end

"""
    ols_fit_k6(X, y)

Optimized OLS for k=6 regressors using StaticArrays (stack-allocated).

This is specifically designed for White test auxiliary regression:
    ê² ~ 1 + x₁ + x₂ + x₁² + x₂² + x₁·x₂

Uses scalar accumulation to build X'X and X'y, then stores them in
stack-allocated SMatrix/SVector for zero heap allocations.

# Arguments
- `X::AbstractMatrix{<:Real}`: design matrix (n × 6).
- `y::AbstractVector{<:Real}`: dependent variable (length n).

# Returns
- `β̂::Vector{Float64}`: OLS coefficient estimates (length 6).
- `ê::Vector{Float64}`: OLS residuals (length n).
"""
function ols_fit_k6(X::AbstractMatrix{<:Real}, y::AbstractVector{<:Real})
    n = length(y)
    @assert size(X, 1) == n
    @assert size(X, 2) == 6 "ols_fit_k6 optimized for k=6 (auxiliary regression)"

    # Compute X'X and X'y via scalar accumulation
    # X'X is symmetric 6×6, so we compute 21 unique elements
    S11 = 0.0; S12 = 0.0; S13 = 0.0; S14 = 0.0; S15 = 0.0; S16 = 0.0
```

```
            S22 = 0.0; S23 = 0.0; S24 = 0.0; S25 = 0.0; S26 = 0.0
                       S33 = 0.0; S34 = 0.0; S35 = 0.0; S36 = 0.0
                                  S44 = 0.0; S45 = 0.0; S46 = 0.0
                                             S55 = 0.0; S56 = 0.0
                                                        S66 = 0.0

    Sy1 = 0.0; Sy2 = 0.0; Sy3 = 0.0; Sy4 = 0.0; Sy5 = 0.0; Sy6 = 0.0

    @inbounds for i in 1:n
        x1 = Float64(X[i, 1])
        x2 = Float64(X[i, 2])
        x3 = Float64(X[i, 3])
        x4 = Float64(X[i, 4])
        x5 = Float64(X[i, 5])
        x6 = Float64(X[i, 6])
        yi = Float64(y[i])

        # Accumulate X'X (upper triangle only)
        S11 += x1 * x1; S12 += x1 * x2; S13 += x1 * x3; S14 += x1 * x4; S15 += x1 *
        ↪  x5; S16 += x1 * x6
                        S22 += x2 * x2; S23 += x2 * x3; S24 += x2 * x4; S25 += x2 *
                        ↪  x5; S26 += x2 * x6
                                        S33 += x3 * x3; S34 += x3 * x4; S35 += x3 *
                                        ↪  x5; S36 += x3 * x6
                                                        S44 += x4 * x4; S45 += x4 *
                                                        ↪  x5; S46 += x4 * x6
                                                                        S55 += x5 *
                                                                        ↪  x5; S56
                                                                        ↪  += x5 *
                                                                        ↪  x6

                                                                                    ↪  S66
                                                                                    ↪  +=
                                                                                    ↪  x6
                                                                                    ↪  *
                                                                                    ↪  x6

        # Accumulate X'y
        Sy1 += x1 * yi
        Sy2 += x2 * yi
        Sy3 += x3 * yi
        Sy4 += x4 * yi
        Sy5 += x5 * yi
        Sy6 += x6 * yi
    end

    # Build STACK-ALLOCATED 6×6 matrix using SMatrix (zero heap allocations!)
```

```
    XtX = @SMatrix [S11 S12 S13 S14 S15 S16;
                    S12 S22 S23 S24 S25 S26;
                    S13 S23 S33 S34 S35 S36;
                    S14 S24 S34 S44 S45 S46;
                    S15 S25 S35 S45 S55 S56;
                    S16 S26 S36 S46 S56 S66]

    Xty = @SVector [Sy1, Sy2, Sy3, Sy4, Sy5, Sy6]

    # Solve using Cholesky (StaticArrays has optimized linear algebra)
    C = cholesky(XtX)
    β̂_static = C \ Xty

    # Convert back to regular Vector for consistency
    β̂ = Vector{Float64}(β̂_static)

    # Compute residuals manually
    ê = Vector{Float64}(undef, n)
    @inbounds for i in 1:n
        ê[i] = Float64(y[i]) - (β̂[1] * Float64(X[i, 1]) +
                                β̂[2] * Float64(X[i, 2]) +
                                β̂[3] * Float64(X[i, 3]) +
                                β̂[4] * Float64(X[i, 4]) +
                                β̂[5] * Float64(X[i, 5]) +
                                β̂[6] * Float64(X[i, 6]))
    end

    return β̂, ê
end


# === WHITE'S HETEROSKEDASTICITY TEST ===

"""
    white_test(X, y; α=0.05)
Perform White's test for heteroskedasticity.

Tests H₀: Homoskedasticity vs H₁: Heteroskedasticity.

# Procedure:
1. Fit OLS: y = Xβ + u, obtain residuals ê
2. Construct auxiliary regressors Z = [1, x₁, x₂, x₁², x₂², x₁x₂]
3. Fit auxiliary regression: ê² = Zγ + v
4. Compute test statistic: W = n·R²_aux
5. Compare to χ²(p) where p = number of slope coefficients in auxiliary regression

# Arguments
```

```
- `X::AbstractMatrix{<:Real}`: design matrix (n × 3), with intercept column.
- `y::AbstractVector{<:Real}`: dependent variable (length n).

# Keyword arguments
- `α::Real`: significance level for test (default: 0.05).

# Returns
- `LM::Float64`: test statistic (n·R²).
- `pval::Float64`: p-value from χ² distribution.
- `reject::Bool`: whether to reject H₀ at level α.
"""
function white_test(
    X::AbstractMatrix{<:Real},
    y::AbstractVector{<:Real};
    α::Real = 0.05,
)
    n = length(y)
    @assert size(X, 1) == n
    @assert size(X, 2) == 3 "X must have 3 columns: [1, x₁, x₂]"

    # Step 1: Fit main regression
    β̂ , ê = ols_fit_k3(X, y)
    ê² = ê.^2

    # Step 2: Construct auxiliary design matrix Z
    # Z = [1, x₁, x₂, x₁², x₂², x₁·x₂]
    x1 = X[:, 2]
    x2 = X[:, 3]

    Z = Matrix{Float64}(undef, n, 6)
    Z[:, 1] .= 1.0        # intercept
    Z[:, 2] .= x1         # x₁
    Z[:, 3] .= x2         # x₂
    Z[:, 4] .= x1.^2      # x₁²
    Z[:, 5] .= x2.^2      # x₂²
    Z[:, 6] .= x1 .* x2   # x₁·x₂

    # Step 3: Fit auxiliary regression on ê² (use optimized k=6 solver)
    γ̂ , v̂  = ols_fit_k6(Z, ê²)

    # Step 4: Compute R² from auxiliary regression
    ê²_fitted = Z * γ̂
    SSR = sum(v̂ .^2)
    SST = sum((ê² .- mean(ê²)).^2)
    R² = 1.0 - SSR / SST

    # Step 5: Compute test statistic
```

```
    LM = Float64(n) * R²

    # Degrees of freedom: number of slope coefficients in Z (exclude intercept)
    df = 5

    # Compute p-value from χ² distribution
    chisq_dist = Chisq(df)
    pval = 1.0 - cdf(chisq_dist, LM)

    # Rejection decision
    reject = pval < α

    return LM, pval, reject
end


"""
    white_test!(X, y, Z, ê², aux; α=0.05)

Perform White's test for heteroskedasticity using preallocated buffers.

This version reuses memory buffers across multiple calls, eliminating allocations
in Monte Carlo loops. Algorithm and results are identical to `white_test`.

Tests H₀: Homoskedasticity vs H₁: Heteroskedasticity.

# Procedure:
1. Fit OLS: y = Xβ + u, obtain residuals ê
2. Construct auxiliary regressors Z = [1, x₁, x₂, x₁², x₂², x₁x₂] (in-place)
3. Fit auxiliary regression: ê² = Zγ + v
4. Compute test statistic: W = n·R²_aux
5. Compare to χ²(p) where p = number of slope coefficients in auxiliary regression

# Arguments
- `X::AbstractMatrix{<:Real}`: design matrix (n × 3), with intercept column.
- `y::AbstractVector{<:Real}`: dependent variable (length n).
- `Z::AbstractMatrix{Float64}`: preallocated buffer for auxiliary design (n × 6).
- `ê²::AbstractVector{Float64}`: preallocated buffer for squared residuals (n).
- `aux::AbstractVector{Float64}`: preallocated buffer for auxiliary calculations
  ↪ (n).

# Keyword arguments
- `α::Real`: significance level for test (default: 0.05).

# Returns
- `LM::Float64`: test statistic (n·R²).
- `pval::Float64`: p-value from χ² distribution.
```

```
- `reject::Bool`: whether to reject H₀ at level α.
"""
function white_test!(
    X::AbstractMatrix{<:Real},
    y::AbstractVector{<:Real},
    Z::AbstractMatrix{Float64},
    ê²::AbstractVector{Float64},
    aux::AbstractVector{Float64};
    α::Real = 0.05,
)
    n = length(y)
    @assert size(X, 1) == n
    @assert size(X, 2) == 3 "X must have 3 columns: [1, x₁, x₂]"
    @assert size(Z) == (n, 6) "Z must be n × 6"
    @assert length(ê²) == n "ê² must have length n"
    @assert length(aux) == n "aux must have length n"

    # Step 1: Fit main regression
    β̂ , ê = ols_fit_k3(X, y)

    # Step 2: Compute squared residuals (in-place)
    @inbounds for i in 1:n
        ê²[i] = ê[i]^2
    end

    # Step 3: Construct auxiliary design matrix Z (in-place)
    # Z = [1, x₁, x₂, x₁², x₂², x₁·x₂]
    @inbounds for i in 1:n
        x1 = X[i, 2]
        x2 = X[i, 3]
        Z[i, 1] = 1.0
        Z[i, 2] = x1
        Z[i, 3] = x2
        Z[i, 4] = x1 * x1
        Z[i, 5] = x2 * x2
        Z[i, 6] = x1 * x2
    end

    # Step 4: Fit auxiliary regression on ê²
    γ̂ , v̂  = ols_fit_k6(Z, ê²)

    # Step 5: Compute R² from auxiliary regression
    # Compute fitted values using aux buffer
    @inbounds for i in 1:n
        aux[i] = (γ̂[1] * Z[i, 1] + γ̂[2] * Z[i, 2] + γ̂[3] * Z[i, 3] +
                  γ̂[4] * Z[i, 4] + γ̂[5] * Z[i, 5] + γ̂[6] * Z[i, 6])
    end
```

```
524
525     SSR = sum(v̂ .^2)
526     SST = sum((ê² .- mean(ê²)).^2)
527     R² = 1.0 - SSR / SST
528
529     # Step 6: Compute test statistic
530     LM = Float64(n) * R²
531
532     # Degrees of freedom: number of slope coefficients in Z (exclude intercept)
533     df = 5
534
535     # Compute p-value from χ² distribution
536     chisq_dist = Chisq(df)
537     pval = 1.0 - cdf(chisq_dist, LM)
538
539     # Rejection decision
540     reject = pval < α
541
542     return LM, pval, reject
543 end
544
545
546 # === MONTE CARLO SIMULATIONS ===
547
548 """
549     run_exper(n, design, B;
550               β=[1.0, 1.0, 1.0],
551               α_levels=(0.01, 0.05, 0.10),
552               rng=Random.default_rng())
553
554 Run B Monte Carlo simulations of White's heteroskedasticity test.
555
556 For each simulation:
557 1. Generate data from model with specified design
558 2. Fit OLS and perform White test
559 3. Record test statistic W, p-value, and rejection decisions
560
561 # Arguments
562 - `n::Integer`: sample size (must be multiple of 20).
563 - `design::Integer`: design number (0, 1, or 2).
564 - `B::Integer`: number of Monte Carlo replications (default: 5000).
565
566 # Keyword arguments
567 - `β::AbstractVector{<:Real}`: true coefficient vector (default: [1.0, 1.0, 1.0]).
568 - `α_levels::NTuple{N,<:Real}`: significance levels for rejection rate calculation.
569 - `rng::AbstractRNG`: random number generator.
570
```

```
# Returns
- `NamedTuple`: contains n, design, B, LM (test statistics), pvals, and rejection indicators.

# Notes
- Design 0 (homoskedastic) measures test SIZE (type I error rate)
- Designs 1, 2 (heteroskedastic) measure test POWER
"""
function run_exper(
    n::Integer,
    design::Integer,
    B::Integer=5000;
    β::AbstractVector{<:Real} = [1.0, 1.0, 1.0],
    α_levels::NTuple{N,<:Real} = (0.01, 0.05, 0.10),
    rng::AbstractRNG = Random.default_rng(),
) where N
    # Preallocate output vectors
    LM_vec = Vector{Float64}(undef, B)
    pval_vec = Vector{Float64}(undef, B)
    reject_mat = Matrix{Bool}(undef, B, N)  # B × N matrix for N significance levels

    # Preallocate sample arrays (reused across iterations)
    X = Matrix{Float64}(undef, n, 3)
    y = Vector{Float64}(undef, n)

    # Preallocate white_test! buffers (reused across iterations)
    Z = Matrix{Float64}(undef, n, 6)
    ê² = Vector{Float64}(undef, n)
    aux = Vector{Float64}(undef, n)

    @inbounds for b in 1:B
        # Generate sample
        draw_sample!(X, y, n, design; beta=β, rng=rng)

        # Perform White test
        LM, pval, _ = white_test!(X, y, Z, ê², aux; α=α_levels[1])

        # Store results
        LM_vec[b] = LM
        pval_vec[b] = pval

        # Compute rejection decisions at all α levels
        for (i, α) in enumerate(α_levels)
            reject_mat[b, i] = pval < α
        end
    end
```

```
616
617     return (
618         n = Int(n),
619         design = Int(design),
620         B = Int(B),
621         β = Float64.(β),
622         α_levels = α_levels,
623         LM = LM_vec,
624         pvals = pval_vec,
625         rejections = reject_mat,
626     )
627 end
628
629 """
630     plot_LM_kde(LM_vec, n, design; outdir=..., shade=true, alpha=0.05)
631
632 Plot kernel density estimate of LM statistics with χ²(5) density overlay.
633
634 Uses non-parametric KDE to show smooth empirical density. Under H₀, the LM
    ↪ statistic
635 should follow χ²(5) asymptotically. As n increases, the empirical density should
636 converge to the chi-squared distribution.
637
638 # Arguments
639 - `LM_vec::AbstractVector{<:Real}`: vector of LM statistics from simulations.
640 - `n::Integer`: sample size.
641 - `design::Integer`: design number (0, 1, or 2).
642
643 # Keyword arguments
644 - `outdir::String`: output directory for plot (default:
    ↪ script_dir/../out/plots/ej2).
645 - `shade::Bool`: whether to shade rejection region (default: true).
646 - `alpha::Real`: significance level for test (default: 0.05).
647 - `null::Bool`: whether null hypothesis is true (default: true). If false,
    ↪ auto-scales axes to KDE.
648 """
649 function plot_LM_kde(LM_vec::AbstractVector{<:Real}, n::Integer, design::Integer;
650                      outdir::String=joinpath(@__DIR__, "../out/plots/ej2"),
651                      shade::Bool=true,
652                      alpha::Real=0.05,
653                      null::Bool=true)
654     mkpath(outdir)
655
656     # Degrees of freedom for White test
657     df = 5
658
659     # Compute kernel density estimate (non-parametric, smooth empirical density)
```

```
    kde_result = kde(LM_vec)

    # Determine axis ranges based on whether null hypothesis is true
    if null
        # Fixed axis ranges for visual comparison across different n (under H0)
        xlims_range = (0.0, 25.0)
        ylims_range = (0.0, 0.2)
    else
        # Automatic ranges based on KDE support (when H0 is false)
        xlims_range = (minimum(kde_result.x), maximum(kde_result.x))
        ylims_range = (0.0, maximum(kde_result.density) * 1.1)
    end

    elements = Any[]

    # Pre-compute rejection rate; shade rejection region if requested
    rejection_title = ""
    if shade
        # Critical value for right-tailed test at level alpha (χ²(5) critical
        ↪  value)
        chi_crit = quantile(Chisq(df), 1 - alpha)
        rejection_rate = mean(LM_vec .> chi_crit)
        rejection_title = @sprintf("Rechazo: %.1f\\%%", rejection_rate * 100)

        # Right tail: manually close polygon to y=0 so pgfplots fills correctly
        right_idx = kde_result.x .>= chi_crit
        if any(right_idx)
            xs, ys = kde_result.x[right_idx], kde_result.density[right_idx]
            push!(elements, @pgf PGFPlotsX.Plot(
                {fill = "gray!30", fill_opacity = 0.4, draw = "none"},
                PGFPlotsX.Coordinates([xs; xs[end]; xs[1]], [ys; 0.0; 0.0])))
        end
    end

    # Empirical KDE (solid black)
    push!(elements, @pgf PGFPlotsX.Plot(
        {black, solid, line_width = "1.5pt"},
        PGFPlotsX.Coordinates(collect(kde_result.x), collect(kde_result.density))))

    # Chi-squared(5) overlay (dashed gray, semi-transparent)
    # When null=false this shows the null distribution for reference
    x_ref = collect(range(xlims_range[1], xlims_range[2], length=200))
    chisq_dist = Chisq(df)
    y_ref = pdf.(chisq_dist, x_ref)
    push!(elements, @pgf PGFPlotsX.Plot(
        {color = "gray", dashed, line_width = "1.5pt", opacity = 0.7},
        PGFPlotsX.Coordinates(x_ref, y_ref)))
```

```
706
707     # Build axis using PGFPlotsX directly for precise dimensional and font control.
708     # width/height set the total figure size including all labels.
709     ax = @pgf PGFPlotsX.Axis(
710         {
711             width  = "227pt",
712             height = "188pt",
713             scale_only_axis = false,
714             xmin = xlims_range[1], xmax = xlims_range[2],
715             ymin = ylims_range[1], ymax = ylims_range[2],
716             xlabel = raw"Estadístico $LM$",
717             ylabel = "Densidad",
718             raw"xlabel style={font=\fontsize{10}{12}\selectfont}",
719             raw"ylabel style={font=\fontsize{10}{12}\selectfont}",
720             raw"tick label style={font=\fontsize{8}{9.6}\selectfont}",
721             title = rejection_title,
722             raw"title style={font=\fontsize{10}{12}\selectfont, at={(0,1)},
                ↪  anchor=south west, xshift=0pt}",
723         },
724         elements...
725     )
726
727     tp = PGFPlotsX.TikzPicture(ax)
728
729     fname_base = "LM_n$(n)_design$(design)"
730     PGFPlotsX.pgfsave(joinpath(outdir, fname_base * ".tex"), tp;
        ↪  include_preamble=false)
731     PGFPlotsX.pgfsave(joinpath(outdir, fname_base * ".pdf"), tp)
732 end
733
734 """
735     print_white_stats(stats)
736
737 Print White test rejection rate statistics in a readable tabular format.
738 """
739 function print_white_stats(stats)
740     println("\n" * "="^80)
741     println("WHITE TEST REJECTION RATES")
742     println("="^80)
743     println(@sprintf("%-6s %-10s %-8s %-15s",
744                      "n", "Design", "α", "Reject Rate"))
745     println("-"^80)
746
747     for s in stats
748         println(@sprintf("%-6d %-10d %-8.3f %-15.4f",
749                          s.n, s.design, s.alpha, s.rejection_rate))
750     end
```

```
    println("="^80 * "\n")
end

"""
    print_white_summary(rejection_stats::Vector)

Print summary table of White test results across all experiments.
"""
function print_white_summary(rejection_stats::Vector)
    # Extract unique values
    n_values = sort(unique([s.n for s in rejection_stats]))
    design_values = sort(unique([s.design for s in rejection_stats]))

    # Organize data by n, design, alpha
    data = Dict()
    for s in rejection_stats
        if !haskey(data, s.n)
            data[s.n] = Dict()
        end
        if !haskey(data[s.n], s.design)
            data[s.n][s.design] = Dict()
        end
        data[s.n][s.design][s.alpha] = s.rejection_rate
    end

    println("\n" * "="^80)
    println("WHITE TEST SUMMARY (Empirical Rejection Rates at α = 5%)")
    println("="^80)
    println(@sprintf("%-10s", "n"), " | ", join([@sprintf("Design %-4d", d) for d
    ↪  in design_values], " | "))
    println("-"^80)

    for n in n_values
        row_str = @sprintf("%-10d", n)
        for design in design_values
            if haskey(data[n], design) && haskey(data[n][design], 0.05)
                rate = data[n][design][0.05]
                row_str *= " | " * @sprintf("%-10.4f", rate)
            else
                row_str *= " | " * @sprintf("%-10s", "N/A")
            end
        end
        println(row_str)
    end
    println("="^80)
    println("Note: Design 0 measures SIZE, Designs 1-2 measure POWER")
    println("="^80)
```

```
end

"""
    write_rejection_rates_latex(rejection_stats;
                                filename="rejection_rates.tex",
                                outdir=joinpath(@__DIR__, "../out/tables"))

Write White test rejection rates to a LaTeX booktabs table with hierarchical column
↪ structure.

The table organizes results by design (columns) and n values (rows), with each
↪ major
column subdivided by significance level (1%, 5%, 10%).

# Arguments
- `rejection_stats::Vector`: Vector of rejection statistics from `run_monte_carlo`.

# Keyword arguments
- `filename::String`: Output filename (default: "rejection_rates.tex").
- `outdir::String`: Output directory (default: script_dir/../out/tables/ej2).
"""
function write_rejection_rates_latex(rejection_stats::Vector;
                                     filename::String = "rejection_rates.tex",
                                     outdir::String = joinpath(@__DIR__,
                                     ↪ "../out/tables/ej2"))
    mkpath(outdir)

    # Extract unique values and sort
    n_values = sort(unique([s.n for s in rejection_stats]))
    design_values = sort(unique([s.design for s in rejection_stats]))
    α_levels = sort(unique([s.alpha for s in rejection_stats]))

    # Organize data: data[n][design][α] = rejection_rate
    data = Dict()
    for s in rejection_stats
        if !haskey(data, s.n)
            data[s.n] = Dict()
        end
        if !haskey(data[s.n], s.design)
            data[s.n][s.design] = Dict()
        end
        data[s.n][s.design][s.alpha] = s.rejection_rate
    end

    # Format rejection rate as percentage with smart rounding
    # Strips trailing zeros: 100.00 → 100, 97.50 → 97.5, 97.06 → 97.06
    function split_decimal_rate(x::Real, include_pct::Bool=true)
```

```
        pct = x * 100.0
        str = @sprintf("%.2f", pct)
        # Strip trailing zeros and decimal point if integer
        str = replace(str, r"\.?0+$" => "")
        # Split at decimal point (if still present)
        parts = split(str, '.')
        if length(parts) == 2
            # Has decimal part - include the dot in the decimal part
            if include_pct
                return (parts[1], "." * parts[2] * "\\%")
            else
                return (parts[1], "." * parts[2])
            end
        else
            # No decimal point (integer) - empty decimal part
            if include_pct
                return (str, "\\%")
            else
                return (str, "")
            end
        end
    end

    # Open file and write table
    open(joinpath(outdir, filename), "w") do io
        # Each design has 3 significance levels, each split into r@{}l = 6 actual columns per design
        println(io, "\\begin{tabular}{c" * "r@{}lr@{}lr@{}l"^length(design_values) * "}")
        println(io, "\\toprule")

        # Top level: Design values (each spans 6 columns)
        print(io, "")
        for design in design_values
            print(io, " & \\multicolumn{6}{c}{Diseño $design}")
        end
        println(io, " \\\\")

        # Add cmidrules for top level
        for (i, _) in enumerate(design_values)
            col_start = 2 + 6*(i-1)
            col_end = col_start + 5
            print(io, "\\cmidrule(lr){$col_start-$col_end} ")
        end
        println(io)

        # Second level: significance levels (each spans 2 columns)
```

```
        print(io, "\$n\$")
        for _ in design_values
            print(io, " & \\multicolumn{2}{c}{\$1\\%\$}")
            print(io, " & \\multicolumn{2}{c}{\$5\\%\$}")
            print(io, " & \\multicolumn{2}{c}{\$10\\%\$}")
        end
        println(io, " \\\\")

        println(io, "\\midrule")

        # Data rows
        for (row_idx, n) in enumerate(n_values)
            print(io, "$n")
            is_first_row = (row_idx == 1)

            for design in design_values
                for α in α_levels
                    if haskey(data[n], design) && haskey(data[n][design], α)
                        rate = data[n][design][α]
                        int_part, dec_part = split_decimal_rate(rate, is_first_row)
                        print(io, " & $int_part & $dec_part")
                    else
                        print(io, " & \\multicolumn{2}{c}{--}")
                    end
                end
            end
            println(io, " \\\\")
        end

        println(io, "\\bottomrule")
        println(io, "\\end{tabular}")
    end

    @info "LaTeX table written to $(joinpath(outdir, filename))"
end

"""
    run_monte_carlo(B;
                    n_values=[20, 60, 100, 200, 400, 600],
                    design_values=[0, 1, 2],
                    β=[1.0, 1.0, 1.0],
                    α_levels=(0.01, 0.05, 0.10),
                    generate_plots=true,
                    rng=Random.default_rng())

Run White test Monte Carlo simulations for all combinations of sample sizes and
↪ designs.
```

```
This is the "outer loop" that sweeps all parameter combinations. For each (n, design)
combination, calls `run_exper` to perform B Monte Carlo replications. Optionally
generates KDE plots of LM statistics.

# Arguments
- `B::Integer`: Monte Carlo replications per combination.

# Keyword arguments
- `n_values::Vector{<:Integer}`: sample sizes to test.
- `design_values::Vector{<:Integer}`: designs to test (0, 1, 2).
- `β::AbstractVector{<:Real}`: true coefficient vector.
- `α_levels::NTuple{N,<:Real}`: significance levels.
- `generate_plots::Bool`: whether to generate LM-statistic KDE plots (default: true).
- `rng::AbstractRNG`: random number generator.

# Returns
- `Vector`: Rejection rate statistics as vector of NamedTuples containing (n, design, alpha, rejection_rate).
"""
function run_monte_carlo(
    B::Integer;
    n_values::Vector{<:Integer} = [20, 60, 100, 200, 400, 600],
    design_values::Vector{<:Integer} = [0, 1, 2],
    β::AbstractVector{<:Real} = [1.0, 1.0, 1.0],
    α_levels::NTuple{N,<:Real} = (0.01, 0.05, 0.10),
    generate_plots::Bool = true,
    rng::AbstractRNG = Random.default_rng(),
) where N

    # Storage for statistics
    rejection_stats = Vector{@NamedTuple{n::Int, design::Int, alpha::Float64, rejection_rate::Float64}}()

    @info "Starting White Test Monte Carlo with B=$B simulations"

    for n in n_values
        for design in design_values
            @info "Running experiment: n=$n, design=$design"

            # Run experiment
            result = run_exper(n, design, B; β=β, α_levels=α_levels, rng=rng)

            # Compute rejection rates for each alpha
            for (i, alpha) in enumerate(α_levels)
```

```
                rejection_rate = mean(result.rejections[:, i])

                push!(rejection_stats, (
                    n = n,
                    design = design,
                    alpha = alpha,
                    rejection_rate = rejection_rate
                ))
            end

            # Generate LM-statistic plot (optional)
            if generate_plots
                @info "Generating LM-statistic plot for n=$n, design=$design"
                plot_LM_kde(result.LM, n, design;
                           outdir=joinpath(@__DIR__, "../out/plots/ej2"),
                           null=(design == 0))
            end
        end
    end

    @info "Monte Carlo experiment complete"

    return rejection_stats
end


# === REJECTION RATE BY SAMPLE SIZE PLOTS ===

"""
    plot_rejection_rates_by_n(B;
                              n_grid=20:20:600,
                              design_values=[0, 1, 2],
                              α_levels=(0.01, 0.05, 0.10),
                              outdir=joinpath(@__DIR__, "../out/plots/ej2"))

Generate plots showing rejection rates as a function of sample size for White test.

Creates one plot per significance level, with separate lines for each design.
Each plot shows dots at data points connected by lines, plus a horizontal reference
line at the nominal significance level.

# Arguments
- `B::Integer`: Number of Monte Carlo replications per (n, design) combination.

# Keyword arguments
- `n_grid`: Sample sizes to test (default: 20:20:600).
- `design_values::Vector{<:Integer}`: Designs to include (default: [0, 1, 2]).
```

```
- `α_levels::Tuple`: Significance levels for plots (default: (0.01, 0.05, 0.10)).
- `outdir::String`: Output directory for plots.
"""
function plot_rejection_rates_by_n(
    B::Integer;
    n_grid = 20:20:600,
    design_values::Vector{<:Integer} = [0, 1, 2],
    α_levels::Tuple = (0.01, 0.05, 0.10),
    outdir::String = joinpath(@__DIR__, "../out/plots/ej2")
)
    # Run Monte Carlo with specified n-grid
    @info "Running Monte Carlo for rejection rate plots (n = $(n_grid[1]):$(step(n_grid)):$(n_grid[end]))"
    n_vals = collect(n_grid)
    mc_stats_plot = run_monte_carlo(
        B;
        n_values = n_vals,
        design_values = design_values,
        β = [1.0, 1.0, 1.0],
        α_levels = α_levels,
        generate_plots = false
    )

    # Extract rejection rates for a given α level
    function extract_rates_by_design(stats, n_vals, α_target)
        designs = sort(unique([s.design for s in stats]))
        rates = Dict()
        for d in designs
            rates[d] = Float64[]
            for n in n_vals
                idx = findfirst(s -> s.n == n && s.design == d && s.alpha ≈ α_target, stats)
                if idx !== nothing
                    push!(rates[d], stats[idx].rejection_rate)
                else
                    push!(rates[d], NaN)
                end
            end
        end
        return rates
    end

    # Plot styling - all black with different markers
    design_labels = Dict(0 ⇒ L"Diseño $0$",
                         1 ⇒ L"Diseño $1$",
                         2 ⇒ L"Diseño $2$")
    design_markers = Dict(0 ⇒ :circle, 1 ⇒ :dtriangle, 2 ⇒ :square)
```

```
    # Vertical offsets for labels to avoid overlap with lines
    design_label_offsets = Dict(0 => 0.05, 1 => 0.10, 2 => 0.07)

    mkpath(outdir)

    # Create one plot for each significance level
    for α_level in α_levels
        @info "Creating rejection rate plot for α = $α_level"

        rates_by_design = extract_rates_by_design(mc_stats_plot, n_vals, α_level)

        # Create plot without legend
        p = plot(
            xlabel = L"Tamaño de muestra ($n$)",
            ylabel = "Tasa de rechazo",
            legend = false,
            size = (600, 400),
            ylims = (0, 1),
            yticks = [0, α_level, 0.25, 0.5, 0.75, 1.0],
            grid = true,
            framestyle = :box
        )

        # Plot each design as dots connected by lines
        for design in sort(collect(keys(rates_by_design)))
            plot!(p, n_vals, rates_by_design[design],
                  label = "",
                  color = :black,
                  marker = design_markers[design],
                  markersize = 4,
                  linewidth = 1.5,
                  linestyle = :solid)
        end

        # Add text labels near the middle of each line (around 50%)
        for design in sort(collect(keys(rates_by_design)))
            # Get the rates to determine label position
            rates = rates_by_design[design]
            # Find position around 50% along the x-axis
            label_idx = max(1, round(Int, 0.5 * length(n_vals)))
            label_x = n_vals[label_idx]
            label_y = rates[label_idx]

            # Use design-specific offset to avoid overlap
            y_offset = design_label_offsets[design]
            annotate!(p, label_x, label_y + y_offset,
                      text(design_labels[design], 9, :center))
```

```
        end

        # Save plot with α in filename
        α_str = replace(string(α_level), "." ⇒ "")
        fname_base = "rejection_rate_by_n_alpha$(α_str)"
        savefig(p, joinpath(outdir, fname_base * ".tex"))
        savefig(p, joinpath(outdir, fname_base * ".pdf"))

        @info "Plot saved: $(fname_base).pdf"
    end

    @info "All rejection rate plots saved to $outdir"
end


# === EXECUTION ===

# Run Monte Carlo with plot generation for LM density plots (Figure ej-2-LM-design0)
mc_stats = run_monte_carlo(
    5000;
    n_values       = [20, 60, 100, 200, 400, 600],
    generate_plots = true,
    design_values  = [0, 1, 2],
    β              = [1.0, 1.0, 1.0],
    α_levels       = (0.01, 0.05, 0.10),
)

print_white_summary(mc_stats)
write_rejection_rates_latex(mc_stats)

# Generate rejection rate by n plots (Figures rejection-rates-by-n-alpha-*)
plot_rejection_rates_by_n(100000)
```

El contenido del archivo `ej3.jl` es el siguiente.

```
# ============================================================
# TRABAJO FINAL ECONOMETRIA
# Ejercicio 2 (parte II): Estimación robusta de la varianza de OLS
#
# Autor: Felipe Tappata
# Fecha: Febrero 2026
# ============================================================

# Load dependencies
using Random, Statistics
using LinearAlgebra
```

```julia
using Printf, LaTeXStrings
using Distributions
using StaticArrays
using Plots, KernelDensity
import PGFPlotsX

# Set plotting backend
function __init_plots__()
    pgfplotsx()
    default(tex_output_standalone=false)
end
__init_plots__()

# Set seed
const SEED = 8869
Random.seed!(SEED)

# Fixed regression design: 20 values of x₁ (repeated across simulations)
const X1_BASE = vcat(-1.1, range(-1.0, 1.0, length=18), 1.1)


# === DATA GENERATION ===

"""
    draw_sample_e!(X, y, e, ν, n, design; beta, rng)

In-place data generation that extends the model of `draw_sample!` to additionally
record the composite errors and variance scalings needed for sandwich estimation.

Fills `X` and `y` identically to `draw_sample!`, and also fills:
- `e[i]  = √νᵢ · uᵢ`  — composite error (what OLS residuals estimate)
- `ν[i]  = νᵢ`        — conditional variance of eᵢ given xᵢ (diagonal of Ω)

Restricted to designs 1 and 2 (heteroskedastic).

# Arguments
- `X::AbstractMatrix{Float64}`: pre-allocated design matrix (n × 3).
- `y::AbstractVector{Float64}`: pre-allocated dependent variable (length n).
- `e::AbstractVector{Float64}`: pre-allocated composite error buffer (length n).
- `ν::AbstractVector{Float64}`: pre-allocated variance scaling buffer (length n).
- `n::Integer`: sample size, must be a multiple of 20.
- `design::Integer`: design number (1 or 2).

# Keyword arguments
- `beta::AbstractVector{<:Real}`: coefficient vector [β₀, β₁, β₂] (default: [1.0,
  ↪ 1.0, 1.0]).
- `rng::AbstractRNG`: random number generator.
```

```
"""
function draw_sample_e!(
    X::AbstractMatrix{Float64},
    y::AbstractVector{Float64},
    e::AbstractVector{Float64},
    ν::AbstractVector{Float64},
    n::Integer,
    design::Integer;
    beta::AbstractVector{<:Real} = [1.0, 1.0, 1.0],
    rng::AbstractRNG = Random.default_rng(),
)
    @assert n % 20 == 0        "Sample size n must be a multiple of 20"
    @assert design ∈ (1, 2)    "draw_sample_e! supports designs 1 and 2 only"
    @assert length(beta) == 3  "beta must have 3 elements: [β₀, β₁, β₂]"
    @assert size(X) == (n, 3)  "X must be n × 3"
    @assert length(y) == n     "y must have length n"
    @assert length(e) == n     "e must have length n"
    @assert length(ν) == n     "ν must have length n"

    n = Int(n)
    m = div(n, 20)

    X[:, 1] .= 1.0
    @inbounds for rep in 1:m
        idx_start = (rep - 1) * 20 + 1
        idx_end   = rep * 20
        X[idx_start:idx_end, 2] .= X1_BASE
        rand!(rng, Normal(0, 1), view(X, idx_start:idx_end, 3))
    end

    β₀, β₁, β₂ = Float64.(beta)

    @inbounds for i in 1:n
        x1i = X[i, 2]
        x2i = X[i, 3]
        νi  = exp(0.25 * x1i + 0.25 * x2i)
        ui  = design == 2 ? rand(rng, TDist(5)) : randn(rng)
        ei  = sqrt(νi) * ui
        ν[i] = νi
        e[i] = ei
        y[i] = β₀ + β₁ * x1i + β₂ * x2i + ei
    end

    return nothing
end
```

```julia
# === OLS ESTIMATION ===

"""
    ols_fit_k3(X, y)

Fit OLS regression y = Xβ + u using StaticArrays (stack-allocated) for k=3.
Computes X'X and X'y via scalar accumulation, then
stores in stack-allocated SMatrix/SVector for zero heap allocations.

# Arguments
- `X::AbstractMatrix{<:Real}`: design matrix (n × 3), including intercept column.
- `y::AbstractVector{<:Real}`: dependent variable (length n).

# Returns
- `β̂::Vector{Float64}`: OLS coefficient estimates (length 3).
- `ê::Vector{Float64}`: OLS residuals (length n).
"""
function ols_fit_k3(X::AbstractMatrix{<:Real}, y::AbstractVector{<:Real})
    n = length(y)
    @assert size(X, 1) == n
    @assert size(X, 2) == 3 "ols_fit_k3 optimized for k=3 (intercept + 2 regressors)"

    # Compute X'X and X'y via scalar accumulation (no intermediate allocations)
    # X'X is symmetric, so we only compute upper triangle
    S11 = 0.0; S12 = 0.0; S13 = 0.0
    S22 = 0.0; S23 = 0.0; S33 = 0.0
    Sy1 = 0.0; Sy2 = 0.0; Sy3 = 0.0

    @inbounds for i in 1:n
        x1 = Float64(X[i, 1])
        x2 = Float64(X[i, 2])
        x3 = Float64(X[i, 3])
        yi = Float64(y[i])

        # Accumulate X'X (symmetric)
        S11 += x1 * x1
        S12 += x1 * x2
        S13 += x1 * x3
        S22 += x2 * x2
        S23 += x2 * x3
        S33 += x3 * x3

        # Accumulate X'y
        Sy1 += x1 * yi
        Sy2 += x2 * yi
        Sy3 += x3 * yi
```

```
    end

    # Build STACK-ALLOCATED 3×3 matrix using SMatrix
    XtX = @SMatrix [S11 S12 S13;
                    S12 S22 S23;
                    S13 S23 S33]

    Xty = @SVector [Sy1, Sy2, Sy3]

    # Solve via Cholesky factorization (StaticArrays has optimized linear algebra)
    C = cholesky(XtX)
    β̂_static = C \ Xty

    # Convert back to regular Vector for consistency
    β̂ = Vector{Float64}(β̂_static)

    # Compute residuals manually (must allocate this since n varies)
    ê = Vector{Float64}(undef, n)
    @inbounds for i in 1:n
        ê[i] = Float64(y[i]) - (β̂[1] * Float64(X[i, 1]) +
                                β̂[2] * Float64(X[i, 2]) +
                                β̂[3] * Float64(X[i, 3]))
    end

    return β̂, ê
end


# === SANDWICH VARIANCE ESTIMATION ===

"""
    vcov_diags(X, ê, e, v, design)

Compute the diagonal elements of four variance-covariance estimates for the OLS
estimator β̂ = (X'X)⁻¹X'y under heteroskedasticity.

All four share the sandwich form  V = (X'X)⁻¹ · M · (X'X)⁻¹, differing only
in the meat matrix M = X'ΩX for different choices of Ω:

| Estimator   | Ω diagonal      | Description                                 |
|-------------|-----------------|---------------------------------------------|
| HC0         | ê²ᵢ             | White (1980); ê estimates the composite e   |
| HC1         | [n/(n-k)] · ê²ᵢ | HC0 with degrees-of-freedom correction      |
| V_ideal     | e²ᵢ             | Infeasible: uses true composite errors      |
| V_true      | Var(eᵢ|xᵢ)      | Oracle: uses true conditional variances     |

HC1 is computed at negligible extra cost by scaling the HC0 meat.
```

```
198
199 For V_true, the true conditional variance depends on the error distribution:
200 - Design 1 (Normal): Var(eᵢ|xᵢ) = νᵢ · Var(N(0,1)) = νᵢ
201 - Design 2 (t₅):     Var(eᵢ|xᵢ) = νᵢ · Var(t₅) = νᵢ · 5/3
202
203 # Arguments
204 - `X::AbstractMatrix{<:Real}`: design matrix (n × 3).
205 - `ê::AbstractVector{<:Real}`: OLS residuals (length n); each ê_i estimates eᵢ =
    ↪ √νᵢ uᵢ.
206 - `e::AbstractVector{<:Real}`: true composite errors eᵢ = √νᵢ uᵢ (length n).
207 - `ν::AbstractVector{<:Real}`: variance scaling factors νᵢ = exp(0.25·x₁ᵢ +
    ↪ 0.25·x₂ᵢ) (length n).
208 - `design::Integer`: design number (1 or 2) to determine error distribution
    ↪ variance.
209
210 # Returns
211 Four `SVector{3,Float64}`: `(d_hc0, d_hc1, d_ideal, d_true)`, each holding the
212 three diagonal entries Var(β̂₀), Var(β̂₁), Var(β̂₂) for that estimator.
213 """
214 function vcov_diags(
215     X::AbstractMatrix{<:Real},
216     ê::AbstractVector{<:Real},
217     e::AbstractVector{<:Real},
218     ν::AbstractVector{<:Real},
219     design::Integer,
220 )
221     n = length(ê)
222     k = 3
223     @assert size(X) == (n, k)  "X must be n × 3"
224     @assert length(e) == n     "e must have length n"
225     @assert length(ν) == n     "ν must have length n"
226     @assert design ∈ (1, 2)    "design must be 1 or 2"
227
228     # Variance of the standardized error distribution
229     # Design 1 (Normal): Var(u) = 1
230     # Design 2 (t₅):     Var(u) = 5/(5-2) = 5/3
231     σ²_u = design == 1 ? 1.0 : 5.0/3.0
232
233     # --- (X'X): symmetric 3×3, accumulate upper triangle ---
234     XtX_11 = 0.0; XtX_12 = 0.0; XtX_13 = 0.0
235                   XtX_22 = 0.0; XtX_23 = 0.0
236                                 XtX_33 = 0.0
237
238     # --- Three meat matrices: also 3×3 symmetric ---
239     # m0: weight = ê²ᵢ  (base for HC0 and HC1)
240     m0_11 = 0.0; m0_12 = 0.0; m0_13 = 0.0
241                  m0_22 = 0.0; m0_23 = 0.0
```

```
                            m0_33 = 0.0

    # mi: weight = e²ᵢ  (V_ideal)
    mi_11 = 0.0; mi_12 = 0.0; mi_13 = 0.0
                 mi_22 = 0.0; mi_23 = 0.0
                              mi_33 = 0.0

    # mt: weight = νᵢ   (V_true)
    mt_11 = 0.0; mt_12 = 0.0; mt_13 = 0.0
                 mt_22 = 0.0; mt_23 = 0.0
                              mt_33 = 0.0

    @inbounds for i in 1:n
        x1 = X[i, 1]; x2 = X[i, 2]; x3 = X[i, 3]
        w0 = ê[i]^2
        wi = e[i]^2
        wt = ν[i] * σ²_u  # True variance: Var(eᵢ|xᵢ) = νᵢ · Var(uᵢ)

        XtX_11 += x1*x1; XtX_12 += x1*x2; XtX_13 += x1*x3
                         XtX_22 += x2*x2; XtX_23 += x2*x3
                                          XtX_33 += x3*x3

        m0_11 += w0*x1*x1; m0_12 += w0*x1*x2; m0_13 += w0*x1*x3
                           m0_22 += w0*x2*x2; m0_23 += w0*x2*x3
                                              m0_33 += w0*x3*x3

        mi_11 += wi*x1*x1; mi_12 += wi*x1*x2; mi_13 += wi*x1*x3
                           mi_22 += wi*x2*x2; mi_23 += wi*x2*x3
                                              mi_33 += wi*x3*x3

        mt_11 += wt*x1*x1; mt_12 += wt*x1*x2; mt_13 += wt*x1*x3
                           mt_22 += wt*x2*x2; mt_23 += wt*x2*x3
                                              mt_33 += wt*x3*x3
    end

    XtX = @SMatrix [XtX_11 XtX_12 XtX_13;
                    XtX_12 XtX_22 XtX_23;
                    XtX_13 XtX_23 XtX_33]

    M_hc0   = @SMatrix [m0_11 m0_12 m0_13;
                        m0_12 m0_22 m0_23;
                        m0_13 m0_23 m0_33]

    M_ideal = @SMatrix [mi_11 mi_12 mi_13;
                        mi_12 mi_22 mi_23;
                        mi_13 mi_23 mi_33]

```

```
    M_true  = @SMatrix [mt_11 mt_12 mt_13;
                        mt_12 mt_22 mt_23;
                        mt_13 mt_23 mt_33]

    # Invert (X'X): StaticArrays computes this analytically for 3×3
    A = inv(XtX)

    hc1_scale = Float64(n) / Float64(n - k)

    V_hc0   = A * M_hc0   * A
    V_hc1   = hc1_scale * V_hc0     # scalar  × SMatrix, zero extra matrix multiplies
    V_ideal = A * M_ideal * A
    V_true  = A * M_true  * A

    d_hc0   = SVector(V_hc0[1,1],  V_hc0[2,2],  V_hc0[3,3])
    d_hc1   = SVector(V_hc1[1,1],  V_hc1[2,2],  V_hc1[3,3])
    d_ideal = SVector(V_ideal[1,1], V_ideal[2,2], V_ideal[3,3])
    d_true  = SVector(V_true[1,1],  V_true[2,2],  V_true[3,3])

    return d_hc0, d_hc1, d_ideal, d_true
end


# === MONTE CARLO ===

"""
    run_exper(n, design, B; β, rng)

B Monte Carlo replications to measure finite-sample relative bias in robust variance
estimation for OLS under heteroskedasticity.

For each replication:
1. Draw a sample via `draw_sample_e!` (fills X, y, true composite errors e, and variances ν).
2. Compute OLS estimator β̂ and residuals ê (ê estimates the composite error eᵢ = √νᵢ uᵢ).
3. Compute diagonal entries of HC0, HC1, V_ideal, and V_true via `vcov_diags`.
4. Accumulate relative bias: (estimated - true) / true for each coefficient j ∈ {0,1,2}.

# Arguments
- `n::Integer`: sample size (multiple of 20).
- `design::Integer`: design number (1 or 2).
- `B::Integer`: number of Monte Carlo replications (default: 5000).

```

```
# Keyword arguments
- `β::AbstractVector{<:Real}`: true coefficient vector [β₀, β₁, β₂] (default: [1.0, 1.0, 1.0]).
- `rng::AbstractRNG`: random number generator.

# Returns
NamedTuple with:
- `n`, `design`, `B`: experiment parameters.
- `rel_bias_hc0`, `rel_bias_hc1`, `rel_bias_ideal`: `SVector{3}` of relative biases (b₀, b₁, b₂) for each estimator, where bⱼ = average of (V̂ⱼ - Vⱼ)/Vⱼ over B replications.
- `total_bias_hc0`, `total_bias_hc1`, `total_bias_ideal`: |b₀| + |b₁| + |b₂| (total relative bias).
"""
function run_exper(
    n::Integer,
    design::Integer,
    B::Integer = 5000;
    β::AbstractVector{<:Real} = [1.0, 1.0, 1.0],
    rng::AbstractRNG = Random.default_rng(),
)
    @assert design ∈ (1, 2) "run_exper (ej3) requires design ∈ {1, 2}"

    # Preallocate sample and helper buffers (reused across replications)
    X = Matrix{Float64}(undef, n, 3)
    y = Vector{Float64}(undef, n)
    e = Vector{Float64}(undef, n)
    ν = Vector{Float64}(undef, n)

    # Accumulators: sum over B of (V̂ⱼ - Vⱼ)/Vⱼ for each estimator and coefficient
    acc_hc0   = MVector{3, Float64}(0.0, 0.0, 0.0)
    acc_hc1   = MVector{3, Float64}(0.0, 0.0, 0.0)
    acc_ideal = MVector{3, Float64}(0.0, 0.0, 0.0)

    @inbounds for b in 1:B
        draw_sample_e!(X, y, e, ν, n, design; beta=β, rng=rng)
        _, ê = ols_fit_k3(X, y)
        d_hc0, d_hc1, d_ideal, d_true = vcov_diags(X, ê, e, ν, design)

        @inbounds for j in 1:3
            vt = d_true[j]
            acc_hc0[j]   += (d_hc0[j]   - vt) / vt
            acc_hc1[j]   += (d_hc1[j]   - vt) / vt
            acc_ideal[j] += (d_ideal[j]  - vt) / vt
        end
    end

```

```
    Binv = 1.0 / Float64(B)
    rb_hc0   = SVector{3}(acc_hc0)   * Binv
    rb_hc1   = SVector{3}(acc_hc1)   * Binv
    rb_ideal = SVector{3}(acc_ideal)  * Binv

    return (
        n               = Int(n),
        design          = Int(design),
        B               = Int(B),
        rel_bias_hc0    = rb_hc0,
        rel_bias_hc1    = rb_hc1,
        rel_bias_ideal  = rb_ideal,
        total_bias_hc0   = sum(abs, rb_hc0),
        total_bias_hc1   = sum(abs, rb_hc1),
        total_bias_ideal = sum(abs, rb_ideal),
    )
end

"""
    run_monte_carlo(B; n_values, design_values, β, rng)

Outer loop: run `run_exper` for all combinations of sample sizes and designs.

# Arguments
- `B::Integer`: Monte Carlo replications per (n, design) combination.

# Keyword arguments
- `n_values::Vector{<:Integer}`: sample sizes (default: [20, 60, 100, 200, 400,
  ↪ 600]).
- `design_values::Vector{<:Integer}`: designs to test (default: [1, 2]).
- `β::AbstractVector{<:Real}`: true coefficient vector.
- `rng::AbstractRNG`: random number generator.

# Returns
Vector of NamedTuples, one per (n, design) pair, as returned by `run_exper`.
"""
function run_monte_carlo(
    B::Integer;
    n_values::Vector{<:Integer}      = [20, 60, 100, 200, 400, 600],
    design_values::Vector{<:Integer} = [1, 2],
    β::AbstractVector{<:Real}        = [1.0, 1.0, 1.0],
    rng::AbstractRNG                 = Random.default_rng(),
)
    results = Vector{Any}()

    @info "Starting variance bias Monte Carlo: B=$B"

```

```
    for n in n_values
        for design in design_values
            @info "  n=$n, design=$design"
            push!(results, run_exper(n, design, B; β=β, rng=rng))
        end
    end

    @info "Monte Carlo complete"
    return results
end


# === VIEWING RESULTS ===

"""
    print_varbias_results(mc_results)

Print relative bias statistics to the terminal in a readable tabular format.
"""
function print_varbias_results(mc_results)
    println("\n" * "="^90)
    println("VARIANCE BIAS (relative bias of robust variance estimators)")
    println("="^90)
    println(@sprintf("%-6s %-8s %-8s %10s %10s %10s %12s",
                     "n", "Design", "Est", "b₀", "b₁", "b₂", "‖b‖₁"))
    println("-"^90)

    for r in mc_results
        for (label, rb, tb) in [
            ("HC0",   r.rel_bias_hc0,   r.total_bias_hc0),
            ("HC1",   r.rel_bias_hc1,   r.total_bias_hc1),
            ("Ideal", r.rel_bias_ideal, r.total_bias_ideal),
        ]
            println(@sprintf("%-6d %-8d %-8s %10.4f %10.4f %10.4f %12.4f",
                             r.n, r.design, label,
                             rb[1], rb[2], rb[3], tb))
        end
        println("-"^90)
    end
    println("="^90 * "\n")
end

"""
    write_varbias_latex(mc_results;
                        estimators=[:ideal, :hc0],
                        filename="varbias_ideal_hc0.tex",
                        outdir=joinpath(@__DIR__, "../out/tables/ej2"))
```

```
468
469 Write relative bias results to a booktabs LaTeX table.
470
471 Column structure (three-level hierarchy): Design → Estimator → (b₀, b₁, b₂, ‖b‖₁).
472 Numbers are decimal-aligned using r@{.}l column pairs.
473
474 Call once per comparison pair:
475
476     write_varbias_latex(mc_results; estimators=[:hc0, :ideal],
        ↪  filename="varbias_hc0_ideal.tex")
477     write_varbias_latex(mc_results; estimators=[:hc0, :hc1],
        ↪  filename="varbias_hc0_hc1.tex")
478
479 # Arguments
480 - `mc_results`: vector of NamedTuples from `run_monte_carlo`.
481
482 # Keyword arguments
483 - `estimators::Vector{Symbol}`: exactly two estimators to compare; valid values
484   are `:ideal`, `:hc0`, `:hc1`.
485 - `filename::String`: output filename.
486 - `outdir::String`: output directory (default: script_dir/../out/tables/ej2).
487 """
488 function write_varbias_latex(
489     mc_results;
490     estimators::Vector{Symbol} = [:hc0, :ideal],
491     filename::String           = "varbias_hc0_ideal.tex",
492     outdir::String             = joinpath(@__DIR__, "../out/tables/ej2"),
493 )
494     @assert length(estimators) == 2 "Exactly two estimators must be specified"
495     @assert all(e ∈ (:ideal, :hc0, :hc1) for e in estimators) "Each estimator must
        ↪  be :ideal, :hc0, or :hc1"
496
497     mkpath(outdir)
498
499     # Access the relative bias SVector and total bias scalar for a given estimator
500     function get_biases(r, est::Symbol)
501         est == :ideal && return (r.rel_bias_ideal, r.total_bias_ideal)
502         est == :hc0   && return (r.rel_bias_hc0,   r.total_bias_hc0)
503         est == :hc1   && return (r.rel_bias_hc1,   r.total_bias_hc1)
504     end
505
506     est_labels = Dict(
507         :ideal ⇒ raw"$\widehat{V}^{\mathrm{ideal}}$",
508         :hc0   ⇒ raw"$\widehat{V}^{\mathrm{HC0}}$",
509         :hc1   ⇒ raw"$\widehat{V}^{\mathrm{HC1}}$",
510     )
511
```

```
    design_values = sort(unique([r.design for r in mc_results]))
    n_values      = sort(unique([r.n     for r in mc_results]))

    # Build lookup: data[design][n] = NamedTuple
    data = Dict()
    for r in mc_results
        haskey(data, r.design) || (data[r.design] = Dict())
        data[r.design][r.n] = r
    end

    # Format a bias value for r@{.}1 decimal alignment.
    # Wraps negative integer part in math mode for the minus sign.
    function split_bias(x::Real)
        str      = @sprintf("%.3f", x)
        parts    = split(str, '.')
        int_part = parts[1]
        dec_part = length(parts) == 2 ? parts[2] : "000"
        startswith(int_part, "-") && (int_part = "\$" * int_part * "\$")
        return (int_part, dec_part)
    end

    # Column layout: one r@{.}1 pair per (stat × estimator × design)
    n_stats         = 4   # b₀, b₁, b₂, ‖b‖₁
    cols_per_est    = n_stats * 2
    cols_per_design = length(estimators) * cols_per_est
    col_spec        = "c" * ("r@{.}1" ^ (n_stats * length(estimators))) ^
    ↪  length(design_values)

    open(joinpath(outdir, filename), "w") do io
        println(io, "\\begin{tabular}{$col_spec}")
        println(io, "\\toprule")

        # Level 1: Design
        print(io, "")
        for d in design_values
            print(io, " & \\multicolumn{$(cols_per_design)}{c}{Dise\\~no $d}")
        end
        println(io, " \\\\")

        for (i, _) in enumerate(design_values)
            col_start = 2 + cols_per_design * (i - 1)
            col_end   = col_start + cols_per_design - 1
            print(io, "\\cmidrule(lr){$col_start-$col_end} ")
        end
        println(io)

        # Level 2: Estimator
```

```
        print(io, "")
        for _ in design_values
            for est in estimators
                print(io, " &
                ↪  \\multicolumn{$(cols_per_est)}{c}{$(est_labels[est])}")
            end
        end
        println(io, " \\\\")

        for (i, _) in enumerate(design_values)
            for (j, _) in enumerate(estimators)
                col_start = 2 + cols_per_design * (i - 1) + cols_per_est * (j - 1)
                col_end   = col_start + cols_per_est - 1
                print(io, "\\cmidrule(lr){$col_start-$col_end} ")
            end
        end
        println(io)

        # Level 3: Statistic headers
        print(io, "\$n\$")
        for _ in design_values
            for _ in estimators
                print(io, " & \\multicolumn{2}{c}{\$b_0\$}")
                print(io, " & \\multicolumn{2}{c}{\$b_1\$}")
                print(io, " & \\multicolumn{2}{c}{\$b_2\$}")
                print(io, " & \\multicolumn{2}{c}{\$\\|\\boldsymbol{b}\\|_1\$}")
            end
        end
        println(io, " \\\\")

        println(io, "\\midrule")

        # Data rows
        for n in n_values
            print(io, "$n")
            for d in design_values
                for est in estimators
                    if haskey(data, d) && haskey(data[d], n)
                        rb, tb = get_biases(data[d][n], est)
                        for val in (rb[1], rb[2], rb[3], tb)
                            int_p, dec_p = split_bias(val)
                            print(io, " & $int_p & $dec_p")
                        end
                    else
                        for _ in 1:n_stats
                            print(io, " & \\multicolumn{2}{c}{--}")
                        end
```

```
                    end
                end
            end
            println(io, " \\\\")
        end

        println(io, "\\bottomrule")
        println(io, "\\end{tabular}")
    end

    @info "LaTeX table written to $(joinpath(outdir, filename))"
end


# === PLOTTING ===

"""
    plot_varbias_by_n(B;
                      n_grid=20:20:600,
                      design=1,
                      metric=:b0,
                      estimators=[:hc0, :hc1, :ideal],
                      beta=[1.0, 1.0, 1.0],
                      outdir=joinpath(@__DIR__, "../out/plots/ej2"))

Generate plot showing relative bias as a function of sample size.

Creates one plot showing the specified metric (b₀, b₁, b₂, or total bias) for the
given design, with separate lines for each estimator specified.

# Arguments
- `B::Integer`: Number of Monte Carlo replications per n.

# Keyword arguments
- `n_grid`: Sample sizes to test (default: 20:20:600).
- `design::Integer`: Design number (1 or 2).
- `metric::Symbol`: Which bias metric to plot; one of `:b0`, `:b1`, `:b2`, `:total`
  (default: `:b0`).
- `estimators::Vector{Symbol}`: Estimators to include; subset of `[:hc0, :hc1,
  :ideal]` (default: all three).
- `beta::AbstractVector{<:Real}`: True coefficient vector.
- `outdir::String`: Output directory for plots.
"""
function plot_varbias_by_n(
    B::Integer;
    n_grid = 20:20:600,
    design::Integer = 1,
```

```
    metric::Symbol = :b0,
    estimators::Vector{Symbol} = [:hc0, :hc1, :ideal],
    beta::AbstractVector{<:Real} = [1.0, 1.0, 1.0],
    outdir::String = joinpath(@__DIR__, "../out/plots/ej2"),
)
    @assert design ∈ (1, 2) "Design must be 1 or 2"
    @assert metric ∈ (:b0, :b1, :b2, :total) "Metric must be :b0, :b1, :b2, or
    ↪  :total"
    @assert all(e ∈ (:hc0, :hc1, :ideal) for e in estimators) "Each estimator must
    ↪  be :hc0, :hc1, or :ideal"
    @assert length(estimators) >= 1 "At least one estimator must be specified"

    # Run Monte Carlo with specified n-grid
    @info "Running Monte Carlo for varbias plot: design=$design, metric=$metric,
    ↪  n=$(n_grid[1]):$(step(n_grid)):$(n_grid[end])"
    n_vals = collect(n_grid)
    mc_res = run_monte_carlo(
        B;
        n_values = n_vals,
        design_values = [design],
        β = beta,
    )

    # Extract bias values for the specified metric
    function get_bias_value(r, est::Symbol, metric::Symbol)
        rb, tb = if est == :hc0
            (r.rel_bias_hc0, r.total_bias_hc0)
        elseif est == :hc1
            (r.rel_bias_hc1, r.total_bias_hc1)
        else  # :ideal
            (r.rel_bias_ideal, r.total_bias_ideal)
        end

        metric == :b0    && return rb[1]
        metric == :b1    && return rb[2]
        metric == :b2    && return rb[3]
        metric == :total && return tb
    end

    # Build data: bias_data[estimator] = vector of bias values across n_vals
    bias_data = Dict{Symbol, Vector{Float64}}()
    for est in estimators
        bias_data[est] = Float64[]
        for n in n_vals
            idx = findfirst(r -> r.n == n && r.design == design, mc_res)
            push!(bias_data[est], get_bias_value(mc_res[idx], est, metric))
        end
```

```
693     end
694
695     # Plot styling - monochrome with circles, different line styles
696     est_labels = Dict(
697         :hc0   => L"\widehat{\mathbf{V}}_{\widehat{\boldbeta}}^{\mathrm{HC} 0}",
698         :hc1   => L"\widehat{\mathbf{V}}_{\widehat{\boldbeta}}^{\mathrm{HC} 1}",
699         :ideal => L"\widehat{\mathbf{V}}_{\widehat{\boldbeta}}^{\mathrm{ideal}}",
700     )
701     est_linestyles = Dict(:hc0 => :dot, :hc1 => :dash, :ideal => :solid)
702
703     # Metric label for y-axis
704     metric_labels = Dict(
705         :b0    => L"$b_0$",
706         :b1    => L"$b_1$",
707         :b2    => L"$b_2$",
708         :total => L"$||\boldsymbol{b}||_1$",
709     )
710
711     mkpath(outdir)
712
713     # Legend position: top right for total bias, bottom right otherwise
714     legend_pos = metric == :total ? :topright : :bottomright
715
716     # Create plot
717     p = plot(
718         xlabel = L"Tamaño de muestra ($n$)",
719         ylabel = "Sesgo relativo",
720         legend = legend_pos,
721         size = (600, 400),
722         grid = true,
723         framestyle = :box,
724         guidefontsize = 12,
725         tickfontsize = 10,
726         legendfontsize = 12,
727         background_color_legend = nothing,
728         foreground_color_legend = nothing,
729     )
730
731     # Plot each estimator as dots connected by lines
732     for est in estimators
733         plot!(p, n_vals, bias_data[est],
734               label = est_labels[est],
735               color = :black,
736               marker = :circle,
737               markersize = 1.5,
738               linewidth = 1.5,
739               linestyle = est_linestyles[est])
```

```
    end

    # Add horizontal line at zero for reference
    hline!(p, [0.0],
           linestyle = :dash,
           color = :gray,
           alpha = 0.5,
           linewidth = 1.5,
           label = "")

    # Save plot
    metric_str = string(metric)
    fname_base = "varbias_$(metric_str)_design$(design)"
    savefig(p, joinpath(outdir, fname_base * ".tex"))
    savefig(p, joinpath(outdir, fname_base * ".pdf"))

    @info "Plot saved: $(fname_base).pdf"
end


# === EXECUTION ===

mc_results = run_monte_carlo(
    5000;
    n_values      = [20, 60, 100, 200, 400, 600],
    design_values = [1, 2],
    β             = [1.0, 1.0, 1.0],
)

print_varbias_results(mc_results)

write_varbias_latex(mc_results;
                    estimators = [:hc0, :ideal],
                    filename   = "varbias_hc0_ideal.tex")

write_varbias_latex(mc_results;
                    estimators = [:hc0, :hc1],
                    filename   = "varbias_hc0_hc1.tex")

# Generate variance bias plots for all combinations (Figures ej2-varbias-*)
for design in [1, 2]
    for metric in [:b0, :b1, :b2, :total]
        plot_varbias_by_n(100000; design=design, metric=metric)
    end
end

```