Electronic Supplementary Material (ESI) for Physical Chemistry Chemical Physics.

Supporting Information for

Refining DIIS algorithms for Si and GaAs solar cells: Incorporation of weight regularization, conjugate gradient, and reverse automatic differentiation techniques

Zhaosheng Zhang*, Sijia Liu and Yingjie Zhang

College of Chemistry and Materials Science, Hebei University, Baoding, 071002, P. R. China

*misaraty@163.com

Table S1 Implementation of the DIIS algorithm in Julia language

DIIS Solver Function

```
function DIIS_Solver(f, x0, max_iter; tol, diis_max_size=5)
    mixing_factor = 1.0    # Set the initial mixing factor
    x = x0    # Initialize the solution with the initial guess
    fx = f(x)    # Apply the function to the initial guess
    residuals = [fx - x]    # Initialize the list of residuals
    basis = [fx]    # Initialize the list of basis functions

    # Iterate up to the maximum number of iterations
    for n = 1:max_iter
        # Break the loop if the latest residual is below the tolerance
        if norm(residuals[end]) < tol
            break
        end

        # DIIS correction when there is more than one residual
        if length(residuals) > 1
            num_res = length(residuals)
            B = zeros(num_res, num_res)    # Initialize the B matrix

            # Fill the B matrix with dot products of residuals
            for i = 1:num_res
                for j = 1:num_res
                    B[i, j] = dot(residuals[i], residuals[j])
                end
            end

            rhs = zeros(num_res)
            rhs[end] = -1.0

            # Solve for weights using matrix division
            weights = B \ rhs
            weights /= sum(weights)    # Normalize the weights

            # Update the solution vector x
            x = zeros(size(x0))
            for i = 1:num_res
                x += weights[i] * basis[i]
            end
```

```
        else
            # Simple update if only one residual is present
            x += mixing_factor * residuals[end]
        end

        # Compute the new function value and update residuals and basis
        fx = f(x)
        push!(residuals, fx - x)
        push!(basis, fx)

        # Limit the history size of residuals and basis
        if length(residuals) > diis_max_size
            popfirst!(residuals)
            popfirst!(basis)
        end
    end

    # Return the final solution and convergence status
    return (fixpoint = x, converged = norm(residuals[end]) < tol)
end
```

Table S2 Implementation of the Weighted-DIIS algorithm in Julia language

Weighted-DIIS Solver Function

```
function Weighted-DIIS_Solver(f, x0, max_iter; tol, diis_max_size=5, lambda=1e-6)
    x = x0    # Initialize the solution with the initial guess
    fx = f(x)    # Apply the function to the initial guess
    residuals = [fx - x]    # Initialize the list of residuals
    basis = [fx]    # Initialize the list of basis functions

    # Iterate up to the maximum number of iterations
    for n = 1:max_iter
        # Break the loop if the latest residual is below the tolerance
        if norm(residuals[end]) < tol
            break
        end

        # DIIS correction when there is more than one residual
        if length(residuals) > 1
            num_res = length(residuals)
            B = zeros(num_res, num_res)    # Initialize the B matrix

            # Fill the B matrix with dot products of residuals
            for i = 1:num_res
                for j = 1:num_res
                    B[i, j] = dot(residuals[i], residuals[j])
                end
            end

            # Add regularization term to the B matrix
            B += lambda * I

            rhs = zeros(num_res)
            rhs[end] = -1.0

            # Solve for weights using LU decomposition
            weights = B \ rhs
            weights /= sum(weights)    # Normalize the weights

            # Update the solution vector x
            x = zeros(size(x0))
            for i = 1:num_res
```

```
                    x += weights[i] * basis[i]
                end
            else
                # Simple update if only one residual is present
                x += residuals[end]
            end

            # Compute the new function value and update residuals and basis
            fx = f(x)
            push!(residuals, fx - x)
            push!(basis, fx)

            # Limit the history size of residuals and basis
            if length(residuals) > diis_max_size
                popfirst!(residuals)
                popfirst!(basis)
            end
        end

        # Return the final solution and convergence status
        return (fixpoint = x, converged = norm(residuals[end]) < tol)
    end
```

Table S3 Implementation of the CG-Enhanced algorithm in Julia language

CG-Enhanced Solver Function

```
using IterativeSolvers # Import the IterativeSolvers package for the conjugate gradient method

function CG-Enhanced_Solver(f, x0, max_iter; tol, diis_max_size=5, lambda=1e-6)
    x = x0    # Initialize the solution with the initial guess
    fx = f(x)    # Apply the function to the initial guess
    residuals = [fx - x]    # Initialize the list of residuals
    basis = [fx]    # Initialize the list of basis functions

    # Iterate up to the maximum number of iterations
    for n = 1:max_iter
        # Break the loop if the latest residual is below the tolerance
        if norm(residuals[end]) < tol
            break
        end

        # DIIS correction when there is more than one residual
        if length(residuals) > 1
            num_res = length(residuals)
            B = zeros(num_res, num_res)    # Initialize the B matrix

            # Fill the B matrix with dot products of residuals
            for i = 1:num_res
                for j = 1:num_res
                    B[i, j] = dot(residuals[i], residuals[j])
                end
            end

            # Add regularization term to the B matrix
            B += lambda * I

            rhs = zeros(num_res)
            rhs[end] = -1.0

            # Solve for weights using the conjugate gradient method
            weights = cg(B, rhs)
            weights /= sum(weights)    # Normalize the weights

            # Update the solution vector x
```

```
            x = zeros(size(x0))
            for i = 1:num_res
                x += weights[i] * basis[i]
            end
        else
            # Simple update if only one residual is present
            x += residuals[end]
        end

        # Compute the new function value and update residuals and basis
        fx = f(x)
        push!(residuals, fx - x)
        push!(basis, fx)

        # Limit the history size of residuals and basis
        if length(residuals) > diis_max_size
            popfirst!(residuals)
            popfirst!(basis)
        end
    end

    # Return the final solution and convergence status
    return (fixpoint = x, converged = norm(residuals[end]) < tol)
end
```

Table S4 Implementation of the Jacobian-ReverseDiff algorithm in Julia language

Jacobian-ReverseDiff Solver Function

```
using ReverseDiff # Using ReverseDiff for automatic differentiation

function Jacobian_ReverseDiff_Solver(f, x0, max_iter; tol, diis_max_size=5, lambda=1e-6)
    x = x0    # Initialize the solution with the initial guess
    fx = f(x)    # Apply the function to the initial guess
    residuals = [fx - x]    # Initialize the list of residuals
    basis = [fx]    # Initialize the list of basis functions

    # Iterate up to the maximum number of iterations
    for n = 1:max_iter
        # Break the loop if the latest residual is below the tolerance
        if norm(residuals[end]) < tol
            break
        end

        # DIIS correction when more than one residual is available
        if length(residuals) > 1
            num_res = length(residuals)
            B = zeros(num_res, num_res)    # Initialize the B matrix

            # Fill the B matrix with dot products of residuals
            for i = 1:num_res
                for j = 1:num_res
                    B[i, j] = dot(residuals[i], residuals[j])
                end
            end

            B += lambda * I    # Add regularization term

            rhs = zeros(num_res)
            rhs[end] = -1.0

            # Calculate weights using Jacobian matrix from ReverseDiff
            weights = ReverseDiff.jacobian((w) -> B * w - rhs, zeros(num_res)) \ rhs
            weights /= sum(weights)    # Normalize the weights

            # Update the solution vector x
            x = zeros(size(x0))
```

```
            for i = 1:num_res
                x += weights[i] * basis[i]
            end
        else
            # Simple update if only one residual is present
            x += residuals[end]
        end

        # Compute the new function value and update residuals and basis
        fx = f(x)
        push!(residuals, fx - x)
        push!(basis, fx)

        # Limit the history size of residuals and basis
        if length(residuals) > diis_max_size
            popfirst!(residuals)
            popfirst!(basis)
        end
    end

    # Return the final solution and convergence status
    return (fixpoint = x, converged = norm(residuals[end]) < tol)
end
```

Table S5 Implementation of the Gradient-ReverseDiff algorithm in Julia language

Gradient-ReverseDiff Solver Function

```
using ReverseDiff # Using ReverseDiff for automatic differentiation

# Gradient-ReverseDiff Solver Function
function Gradient_ReverseDiff_Solver(f, x0, max_iter; tol, diis_max_size=5, lambda=1e-6)
    x = x0    # Initialize the solution with the initial guess
    fx = f(x)    # Apply the function to the initial guess
    residuals = [fx - x]    # Initialize the list of residuals
    basis = [fx]    # Initialize the list of basis functions

    # Iterate up to the maximum number of iterations
    for n = 1:max_iter
        # Break the loop if the latest residual is below the tolerance
        if norm(residuals[end]) < tol
            break
        end

        # DIIS correction when more than one residual is available
        if length(residuals) > 1
            num_res = length(residuals)
            B = zeros(num_res, num_res)    # Initialize the B matrix

            # Fill the B matrix with dot products of residuals
            for i = 1:num_res
                for j = 1:num_res
                    B[i, j] = dot(residuals[i], residuals[j])
                end
            end

            B += lambda * I    # Add regularization term

            rhs = zeros(num_res)
            rhs[end] = -1.0

            # Calculate weights using gradient function from ReverseDiff
            weights = ReverseDiff.gradient(w -> sum(w .* (B * w - rhs)), zeros(size(rhs)))
            weights /= sum(weights)    # Normalize the weights

            # Update the solution vector x
```

```
            x = zeros(size(x0))
            for i = 1:num_res
                x += weights[i] * basis[i]
            end
        else
            # Simple update if only one residual is present
            x += residuals[end]
        end

        # Compute the new function value and update residuals and basis
        fx = f(x)
        push!(residuals, fx - x)
        push!(basis, fx)

        # Limit the history size of residuals and basis
        if length(residuals) > diis_max_size
            popfirst!(residuals)
            popfirst!(basis)
        end
    end

    # Return the final solution and convergence status
    return (fixpoint = x, converged = norm(residuals[end]) < tol)
end
```

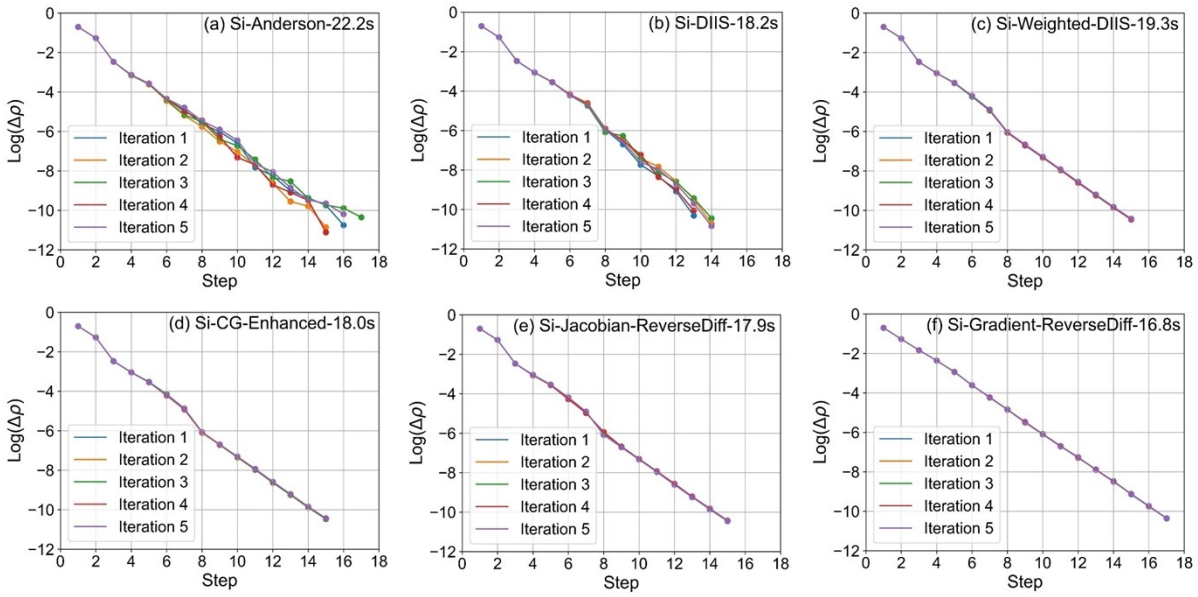

Fig. S1 Electronic minimization algorithm performance in SCF iterations for Si under Ecut15 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

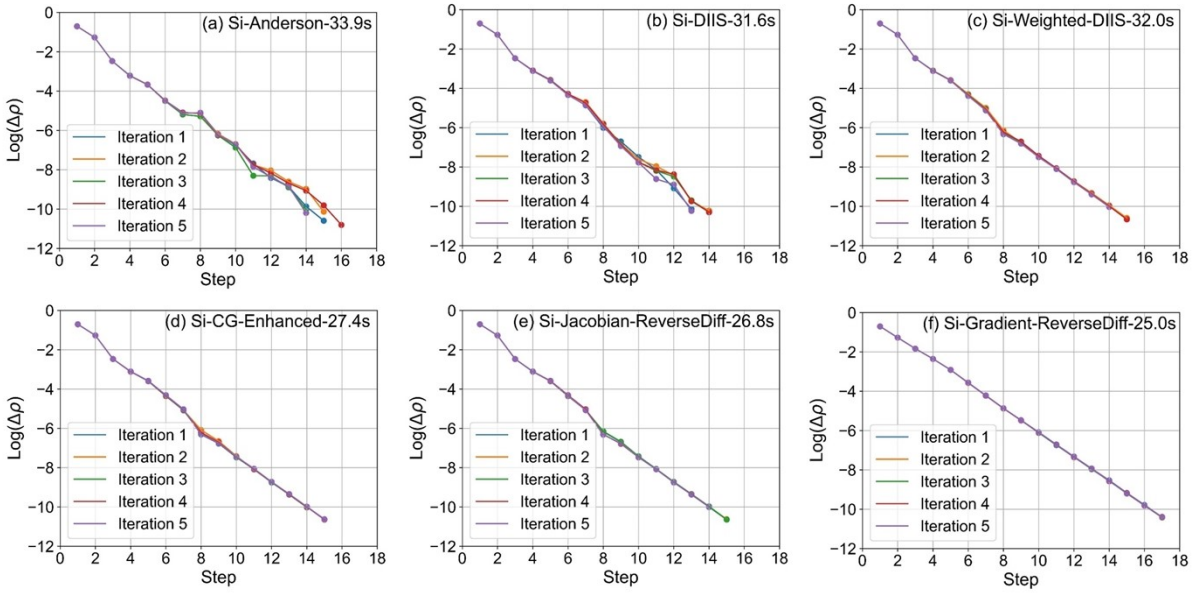

Fig. S2 Electronic minimization algorithm performance in SCF iterations for Si under Ecut25 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

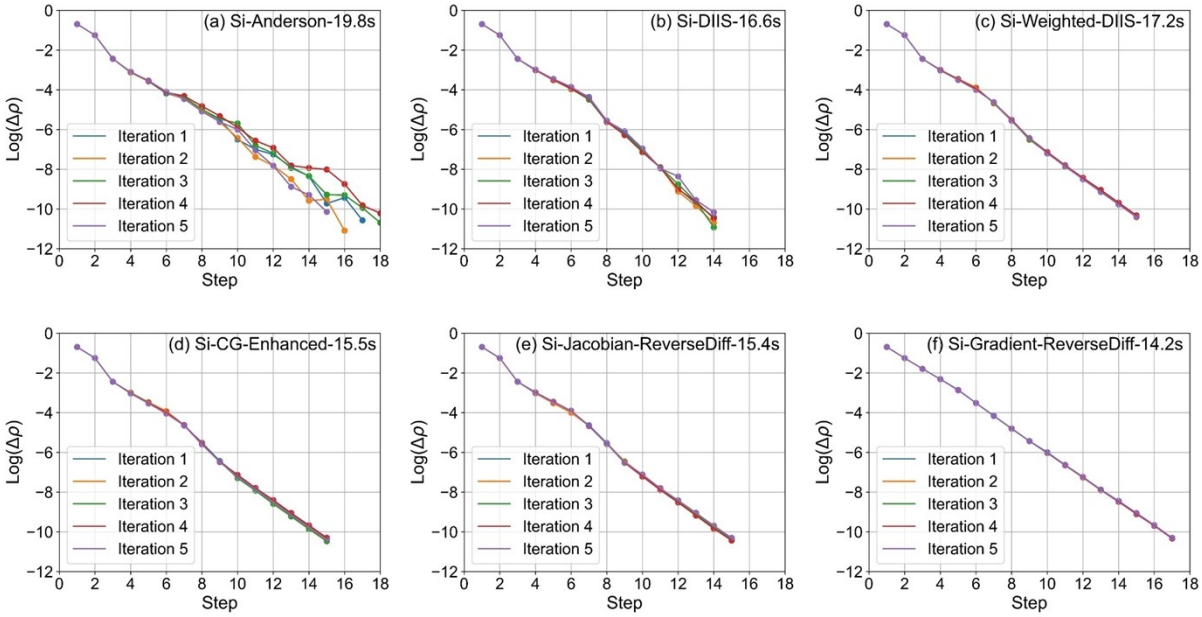

Fig. S3 Electronic minimization algorithm performance in SCF iterations for Si under K555 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

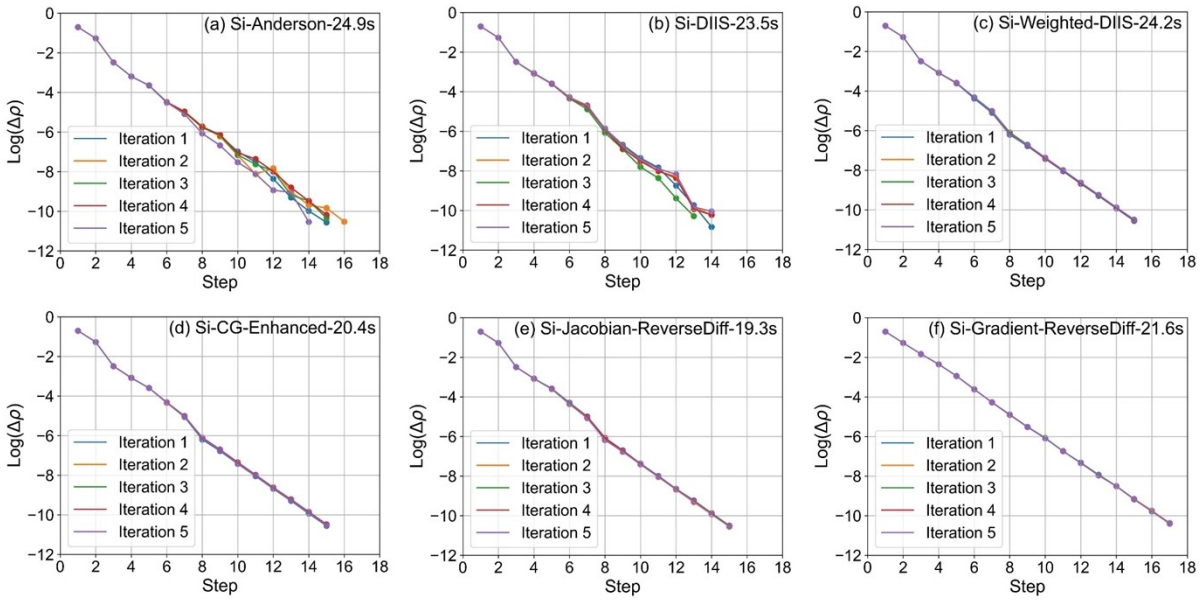

Fig. S4 Electronic minimization algorithm performance in SCF iterations for Si under 777 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

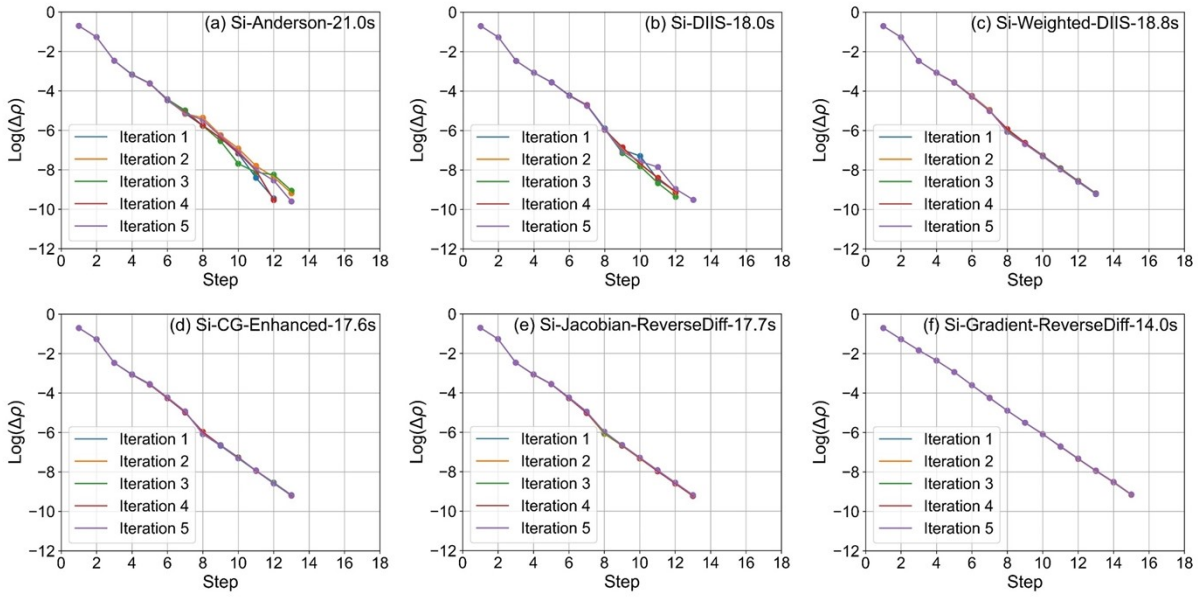

Fig. S5 Electronic minimization algorithm performance in SCF iterations for Si under Tol-9 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

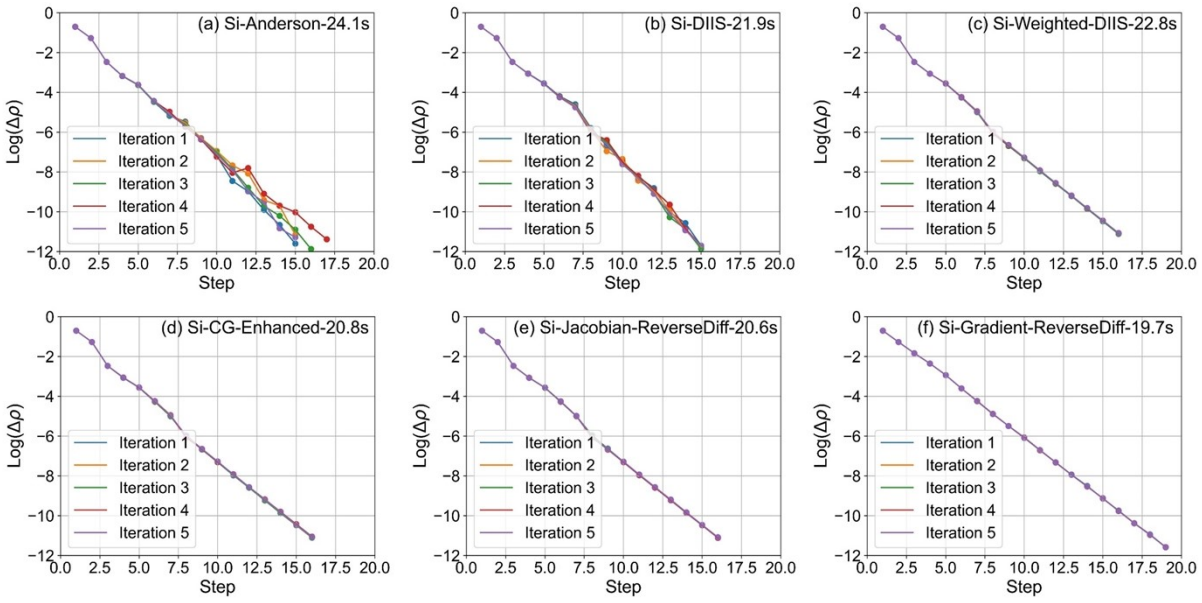

Fig. S6 Electronic minimization algorithm performance in SCF iterations for Si under Tol-11 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

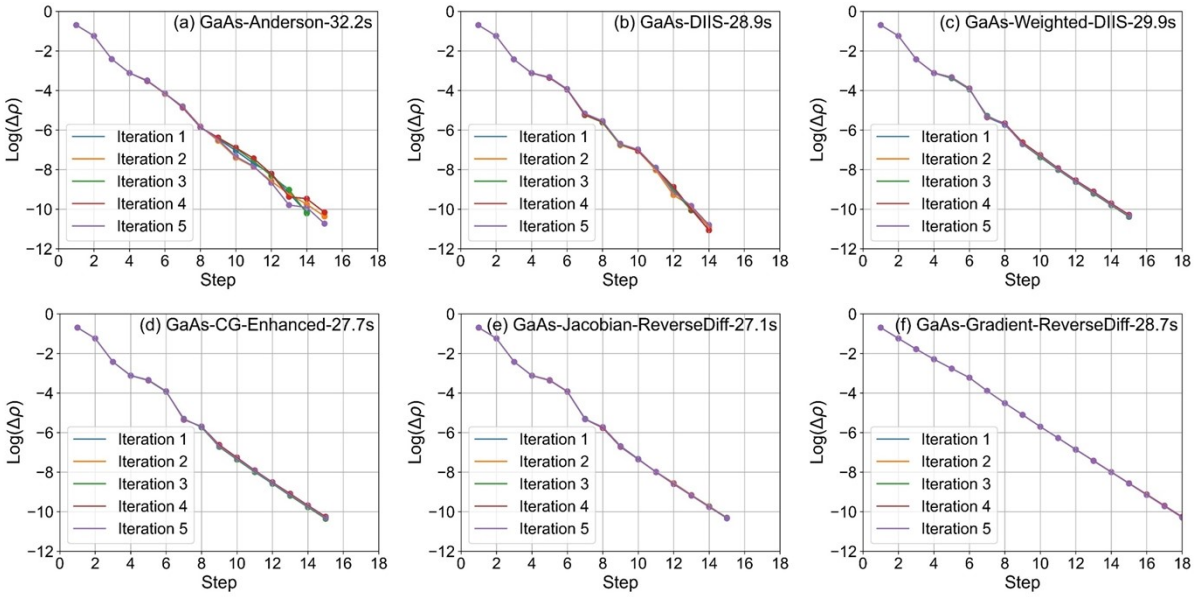

Fig. S7 Electronic minimization algorithm performance in SCF iterations for GaAs under Ecut15 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

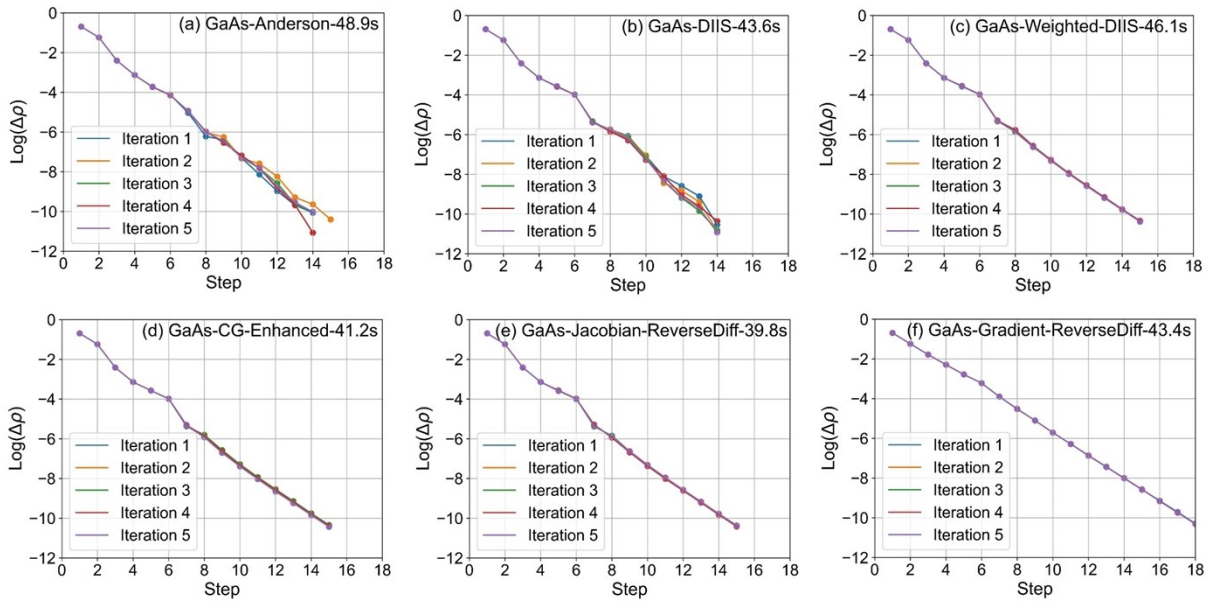

Fig. S8 Electronic minimization algorithm performance in SCF iterations for GaAs under Ecut25 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

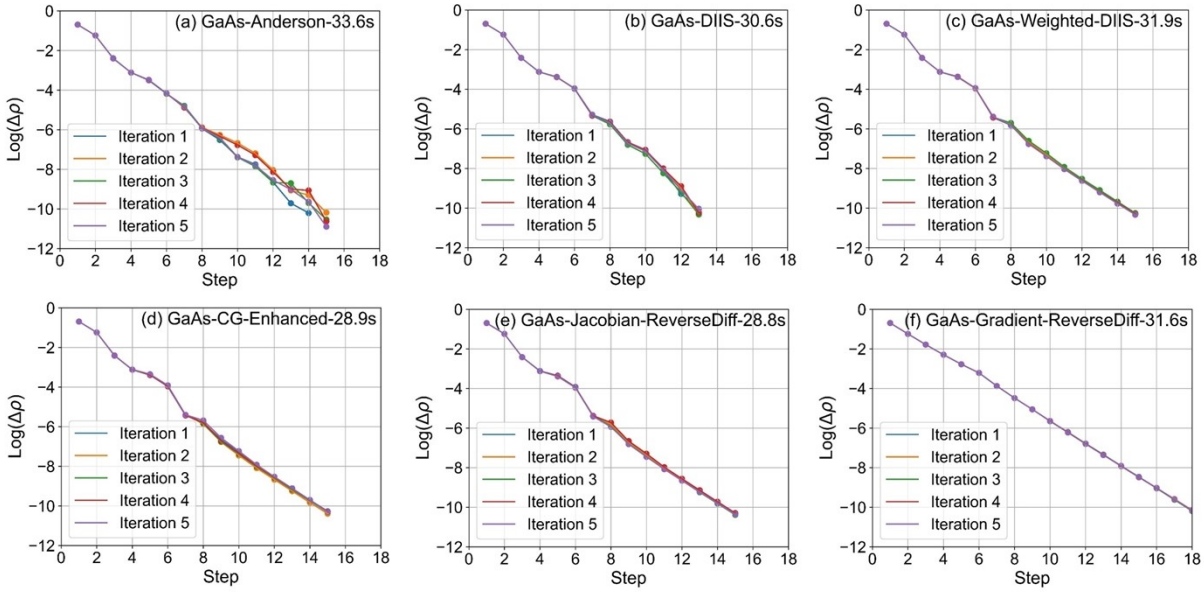

Fig. S9 Electronic minimization algorithm performance in SCF iterations for GaAs under K555 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

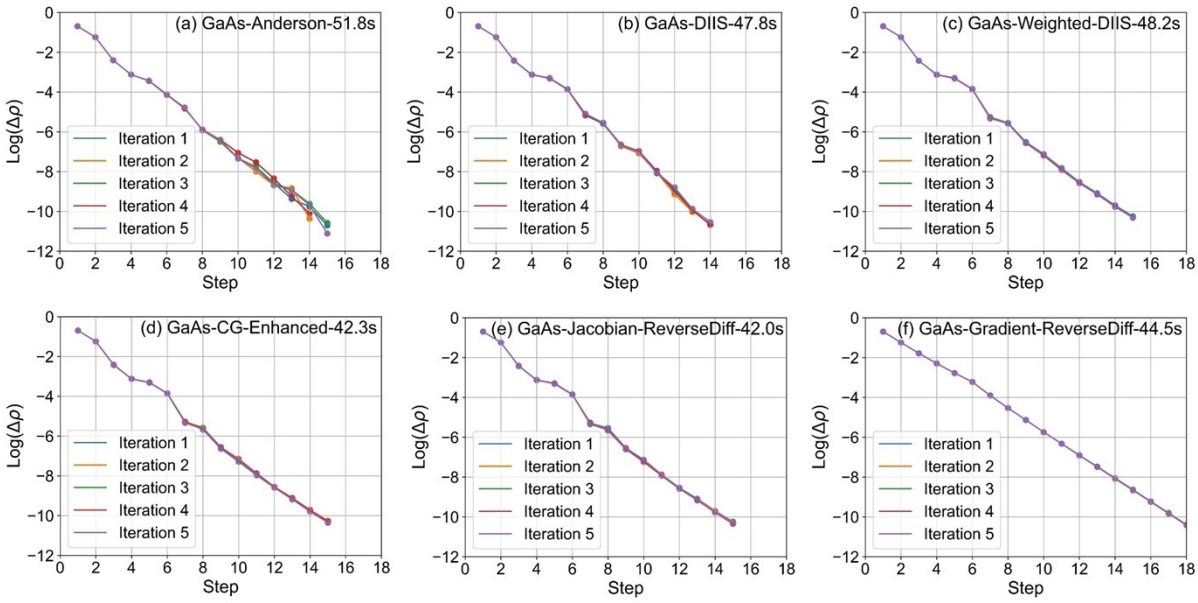

Fig. S10 Electronic minimization algorithm performance in SCF iterations for GaAs under K777 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

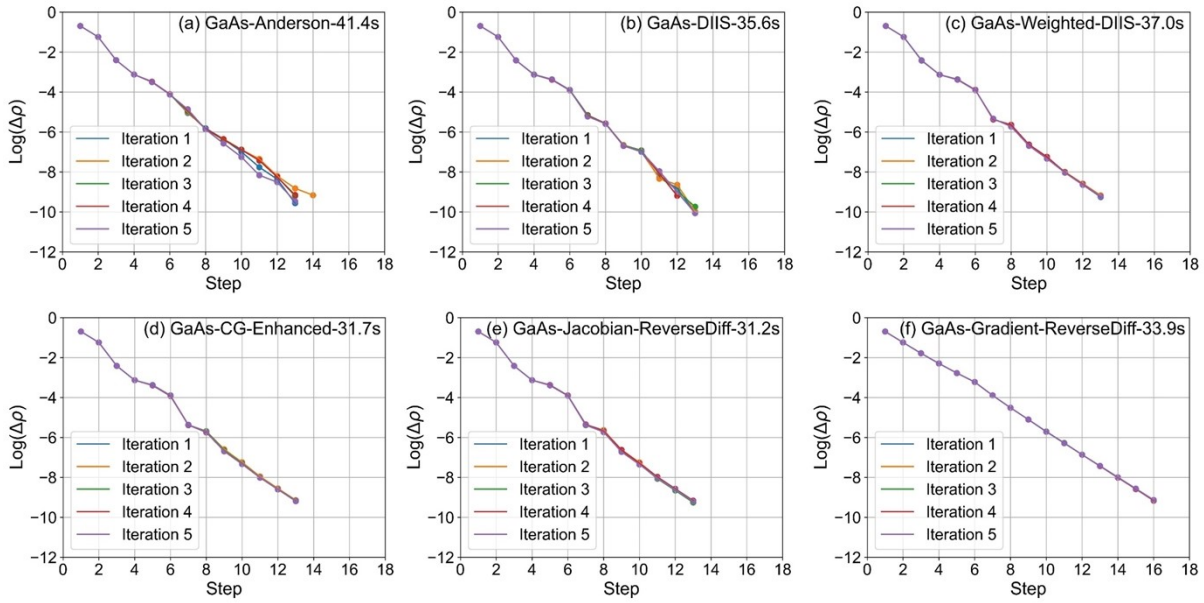

Fig. S11 Electronic minimization algorithm performance in SCF iterations for GaAs under Tol-9 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.

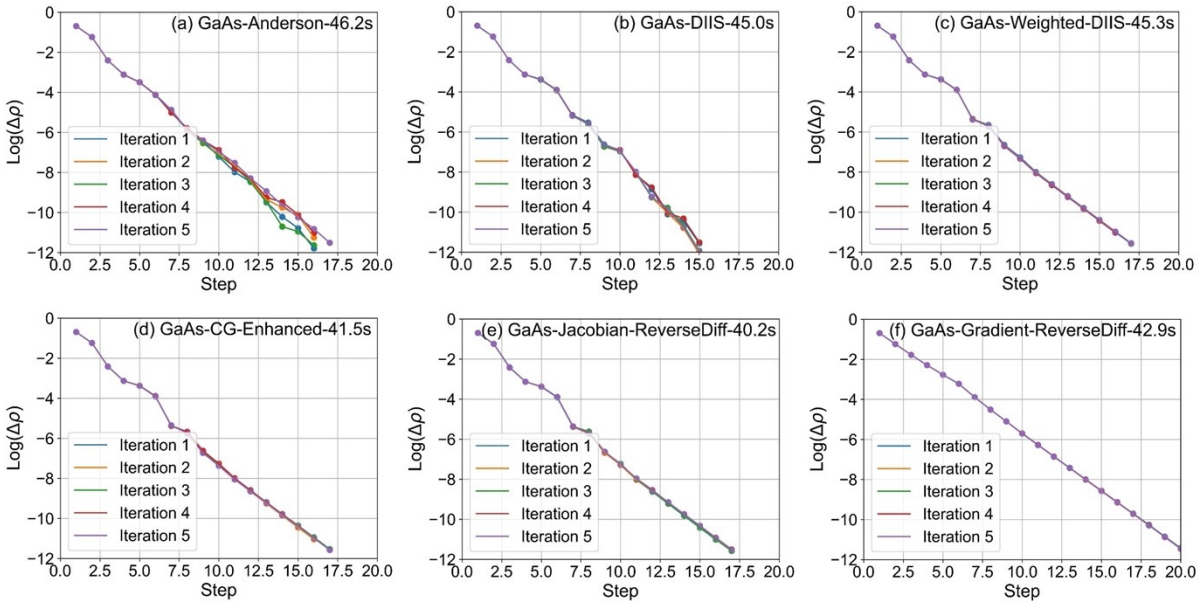

Fig. S12 Electronic minimization algorithm performance in SCF iterations for GaAs under Tol-11 condition specified in Table 1. The algorithms include (a) Anderson, (b) DIIS, (c) Weighted-DIIS, (d) CG-Enhanced, (e) Jacobian-ReverseDiff, and (f) Gradient-ReverseDiff, offering insights into their relative effectiveness and efficiency in the SCF process.