



Código para el Estudio del Problema de los Dos Átomos

A.1 Creación de Mallas

Todas las mallas utilizadas en la simulación del problema de los dos átomos fueron creadas utilizando el software MSMS, considerando átomos de $1.27 \text{ [\AA]}$,

capas de exclusión iónica de 2 [Å] de grosor y una esfera imaginaria de 1.4 [Å] para generar la superficie a simular, del tipo excluída al solvente (sección 2.6.3). Cada malla se genera a partir de un archivo .xyzr. Este tipo de archivo contiene, en cada línea, información de la posición y radio de un átomo; a manera de ejemplo, el archivo .xyzr para la interfaz dieléctrica compartida de dos átomos separados a 3.0 [Å] luce de la forma:

```
1 0.0 0.0 0.0 1.27
2 3.0 0.0 0.0 1.27
```

El software MSMS se llama a través del símbolo del sistema utilizando el comando:

```
msms.exe -if archivo_xyzr -of archivo_salida_malla -p radio -d densidad
-no_header
```

MSMS crea archivos .vert y .face con información sobre la malla, los primeros contienen las coordenadas de los vértices de la malla y sus respectivas normales. Los archivos .face contienen los índices de los vértices que componen cada cara.

Para simular las interfaces dieléctricas se elige una densidad de $15 \left[\frac{\text{vrtices}}{\text{\AA}^2} \right]$, para el caso de las capas de exclusión iónicas, una densidad de $3 \left[\frac{\text{vrtices}}{\text{\AA}^2} \right]$. Se eligen estos valores para generar mallas con números similares de elementos y lo suficientemente finas.

A.2 Parámetros de Simulación

Se definen, antes de realizar la simulación, los parámetros de SLIC y los parámetros del método de resolución GMRES:

```
1 ITER = 20
2 alpha = 0.94
3 beta = -18.72
4 gamma = -0.47
5 MAXITER_GM = 1000
6 TOL = 1e-5
```

`ITER` es el número de iteraciones del método SLIC, `alpha`, `beta` y `gamma` parametrizan el modelo SLIC y `MAXITER_GM` y `TOL` son el número máximo de iteraciones y la tolerancia del método GMRES, respectivamente.

A.3 Configuración 1: Interfaz Dieléctrica y *Stern Layer* Compartidas

Se comienza importando las mallas utilizadas para la interfaz dieléctrica y capa de exclusión iónica:

```
1 grid_dielectric = import_msms_mesh('dielectric.face', 'dielectric.vert')
2 grid_stern = import_msms_mesh('stern.face', 'stern.vert')
```

La función `import_msms_mesh` crea las mallas en el módulo Bempp a partir de archivos `.face` y `.vert`.

Luego, se deben crear los espacios funcionales correspondientes y definir las constantes del problema:

```

1  dirichlet_ion = bempp.api.function_space(grid_dielectric, 'DP', 0)
2  neumann_ion = bempp.api.function_space(grid_dielectric, 'DP', 0)
3  dirichlet_stern = bempp.api.function_space(grid_stern, 'DP', 0)
4  neumann_stern = bempp.api.function_space(grid_stern, 'DP', 0)
5
6  om = 0 #Longitud de Debye
7  ep_i = 1
8  ep_ii = 80
9  ep_iii = 80

```

Se definen los operadores integrales asociados a los sistemas 3.12 y 3.13:

```

1  identity_ion = bempp.api.operators.boundary.sparse.identity(dirichlet_ion,
    ↪  dirichlet_ion, neumann_ion)
2  K_L_I_a_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion,
    ↪  dirichlet_ion, neumann_ion)
3  K_L_I_a_a_adjoint =
    ↪  bempp.api.operators.boundary.laplace.adjoint_double_layer(dirichlet_ion,
    ↪  dirichlet_ion, neumann_ion)
4  V_L_I_a_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion,
    ↪  dirichlet_ion, neumann_ion)
5  K_L_II_a_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion,
    ↪  dirichlet_ion, neumann_ion)
6  V_L_II_a_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion,
    ↪  dirichlet_ion, neumann_ion)

```

```

7 K_L_II_a_b = bempp.api.operators.boundary.laplace.double_layer(dirichlet_stern,
    ↪ dirichlet_ion, neumann_ion)
8 V_L_II_a_b = bempp.api.operators.boundary.laplace.single_layer(neumann_stern,
    ↪ dirichlet_ion, neumann_ion)
9 K_L_II_b_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion,
    ↪ dirichlet_stern, neumann_stern)
10 V_L_II_b_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion,
    ↪ dirichlet_stern, neumann_stern)
11 identity_stern = bempp.api.operators.boundary.sparse.identity(dirichlet_stern,
    ↪ dirichlet_stern, neumann_stern)
12 K_L_II_b_b = bempp.api.operators.boundary.laplace.double_layer(dirichlet_stern,
    ↪ dirichlet_stern, neumann_stern)
13 V_L_II_b_b = bempp.api.operators.boundary.laplace.single_layer(neumann_stern,
    ↪ dirichlet_stern, neumann_stern)
14 K_Y_III_b_b =
    ↪ bempp.api.operators.boundary.modified_helmholtz.double_layer(dirichlet_stern,
    ↪ dirichlet_stern, neumann_stern, om)
15 V_Y_III_b_b =
    ↪ bempp.api.operators.boundary.modified_helmholtz.single_layer(neumann_stern,
    ↪ dirichlet_stern, neumann_stern, om)

```

Posteriormente, se define un arreglo de **Numpy** que contiene las posiciones y cargas de los átomos, según el caso a simular; y se crea la variable **qtotal**:

```

1 atoms = np.array([[0,0,0,q1],[x,0,0,q2]])
2 qtotal=np.sum(atoms[:,3])

```

Se mapea la función analítica del primer término del lado derecho del sistema a la malla y luego se concatena el vector resultante a arreglos de ceros para crear el lado derecho en la variable `rhs`:

```

1 @bempp.api.real_callable
2 def rhs_function(x, n, domain_index, result):
3     result[:] = np.sum(atoms[:,3]/(4 * np.pi * ep_i * impl1(x-atoms[:,3])))
4
5 rhs_1 = bempp.api.GridFunction(dirichlet_ion, fun=rhs_function)
6 rhs_2 = np.zeros(neumann_ion.global_dof_count)
7 rhs_3 = np.zeros(dirichlet_stern.global_dof_count)
8 rhs_4 = np.zeros(neumann_stern.global_dof_count)
9 rhs = np.concatenate([rhs_1.coefficients, rhs_2, rhs_3, rhs_4])

```

`impl1` es una implementación de la función `linalg.norm` de `numpy` en la librería `numba`²⁰. En este punto, el código diverge dependiendo de si se está resolviendo el sistema con la formulación de Poisson-Boltzmann o de SLIC.

A.3.1 Formulación PB

Para crear el lado izquierdo del sistema 3.12 se genera un operador bloqueado y se pobla usando los operadores integrales definidos anteriormente:

```

1 blocked = bempp.api.BlockedOperator(4, 4)
2
3 blocked[0,0] = 0.5 * identity_ion + K_L_I_a_a
4 blocked[0,1] = - V_L_I_a_a

```

```

5
6 blocked[1,0] = 0.5 * identity_ion - K_L_II_a_a
7 blocked[1,1] = (ep_i/ep_ii) * V_L_II_a_a
8 blocked[1,2] = K_L_II_a_b
9 blocked[1,3] = - V_L_II_a_b
10
11 blocked[2,0] = - K_L_II_b_a
12 blocked[2,1] = (ep_i/ep_ii) * V_L_II_b_a
13 blocked[2,2] = 0.5 * identity_stern + K_L_II_b_b
14 blocked[2,3] = - V_L_II_b_b
15
16 blocked[3,2] = 0.5 * identity_stern - K_Y_III_b_b
17 blocked[3,3] = (ep_ii/ep_iii) * V_Y_III_b_b
18
19 blocked = blocked.strong_form()

```

A.3.1.1 Nota Sobre Formas Fuertes y Débiles

El método `strong_form()` entrega una discretización directa del operador entre dos espacios funcionales, multiplicando por la matriz de masas entre ellos:

$$\mathbf{A}_{i,j, \text{disc}} = \mathbf{M}^{-1} \mathbf{A}_{i,j} \quad (\text{A.1})$$

Este método es llamado forma fuerte o *strong form*.

Una forma débil o *weak form* entrega una forma variacional de la discretización del operador entre ambos espacios, es decir, un producto interno entre el oper-

ador y las funciones base de los espacios funcionales:

$$A_{i,j, \text{disc}} = \int A \psi(y) \overline{\phi(y)} dy \quad (\text{A.2})$$

Donde ψ y ϕ son las funciones base de los espacios.

Luego de crear el lado izquierdo, se resuelve el sistema usando la función `gmres` de la librería `scipy`. La función retorna un arreglo que contiene toda la solución, por lo que hay que extraer las soluciones de cada espacio funcional.

```

1 x, info_1 = gmres(blocked, rhs, maxiter=MAXITER_GM, tol = TOL)
2 solution_neumann_stern = bempp.api.GridFunction(neumann_stern, coefficients =
  ↪ x[(dirichlet_ion.global_dof_count+neumann_ion.global_dof_count +
  ↪ dirichlet_stern.global_dof_count):])
3 solution_neumann_ion = bempp.api.GridFunction(neumann_ion, coefficients =
  ↪ x[dirichlet_ion.global_dof_count:(dirichlet_ion.global_dof_count +
  ↪ neumann_ion.global_dof_count)])
4 solution_dirichlet_ion = bempp.api.GridFunction(dirichlet_ion, coefficients =
  ↪ x[:dirichlet_ion.global_dof_count])
5 solution_dirichlet_stern = bempp.api.GridFunction(dirichlet_stern, coefficients =
  ↪ x[(dirichlet_ion.global_dof_count +
  ↪ neumann_ion.global_dof_count):(dirichlet_ion.global_dof_count +
  ↪ neumann_ion.global_dof_count + dirichlet_stern.global_dof_count)])

```

Finalmente, se calcula la energía de solvatación del sistema:

```

1 S0_q = bempp.api.operators.potential.laplace.single_layer(neumann_ion,
    ↪ atoms[:, :3].transpose())
2 D0_q = bempp.api.operators.potential.laplace.double_layer(dirichlet_ion,
    ↪ atoms[:, :3].transpose())
3 phi_q = S0_q * solution_neumann_ion - D0_q * solution_dirichlet_ion
4 en_sol = 2 * np.pi * 332.064 * np.sum(atoms[:, 3] * phi_q).real

```

A.3.2 Formulación SLIC

En este método se debe iterar sobre las soluciones de los sistemas 3.13 y 3.20. Además de mapear el lado derecho del sistema 3.13 a la malla, también es necesario llevar a ella el primer término de la ecuación de campo eléctrico (3.19); esto se hace de la forma:

```

1 @bempp.api.real_callable
2 def en_function(x, n, domain_index, result):
3     result[:] = np.sum(np.dot(n, (np.transpose(x-atoms[:, :3]) / (4 * np.pi *
    ↪ impl1(x-atoms[:, :3])**3)) * (atoms[:, 3])))
4 en_1 = bempp.api.GridFunction(dirichlet_ion, fun=en_function)

```

Se crea, además, una función f que recibe como argumentos la distribución σ y el primer término de la ecuación de campo eléctrico y retorna un arreglo con los valores de $\frac{f}{1+f}$; se define la variable $d1$, y los valores para la primera iteración de σ y $d2$:

```

1 def f(sigma, fun):
2     Ksigma = K_L_I_a_a_adjoint * sigma
3     En = fun - Ksigma
4     mu = - alpha * np.tanh(- gamma)
5     h = alpha * np.tanh(beta * En.coefficients - gamma) + mu
6     f = ep_i/(ep_ii - ep_i) - h
7     f_div = f/(1+f)
8     return f_div
9
10 d2 = 1
11 d1 = - qtotal/ep_ii
12 sigma = bempp.api.GridFunction(dirichlet_ion, coefficients = np.zeros(
    dirichlet_ion.global_dof_count))

```

Luego, se itera sobre la resolución del sistema principal (3.13) y la del sistema 3.20:

```

1 for i in range(ITER):
2     blocked = bempp.api.BlockedOperator(4, 4)
3     if i != 0:
4         d2 = solution_neumann_stern.integrate()[0]
5         f_it = f(sigma, en_1)
6         f_fun = bempp.api.GridFunction(neumann_ion, coefficients = f_it)
7         f_op = bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun,
            ↪ neumann_ion, neumann_ion, neumann_ion)
8         blocked[1,1] = V_L_II_a_a * f_op
9         blocked[2,1] = V_L_II_b_a * f_op
10    else:

```

```

11         blocked[1,1] = V_L_II_a_a
12         blocked[2,1] = V_L_II_b_a
13
14         blocked[0,0] = 0.5 * identity_ion + K_L_I_a_a
15         blocked[0,1] = - V_L_I_a_a
16
17         blocked[1,0] = 0.5 * identity_ion - K_L_II_a_a
18         blocked[1,2] = K_L_II_a_b
19         blocked[1,3] = - V_L_II_a_b
20
21         blocked[2,0] = - K_L_II_b_a
22         blocked[2,2] = 0.5 * identity_stern + K_L_II_b_b
23         blocked[2,3] = - V_L_II_b_b
24
25         blocked[3,2] = 0.5 * identity_stern - K_Y_III_b_b
26         blocked[3,3] = V_Y_III_b_b * (d1/d2)
27
28         blocked = blocked.strong_form()
29
30         x, info_1 = gmres(blocked, rhs, maxiter=MAXITER_GM, tol = TOL)
31
32         solution_neumann_stern = bempp.api.GridFunction(neumann_stern, coefficients =
33         ↪ x[(dirichlet_ion.global_dof_count + neumann_ion.global_dof_count +
34         ↪ dirichlet_stern.global_dof_count):])
35
36         solution_neumann_ion = bempp.api.GridFunction(neumann_ion, coefficients =
37         ↪ x[dirichlet_ion.global_dof_count:(dirichlet_ion.global_dof_count +
38         ↪ neumann_ion.global_dof_count)])

```

```

35
36     solution_dirichlet_ion = bempp.api.GridFunction(dirichlet_ion, coefficients =
    ↪ x[:dirichlet_ion.global_dof_count])
37
38     solution_dirichlet_stern = bempp.api.GridFunction(dirichlet_stern, coefficients
    ↪ = x[(dirichlet_ion.global_dof_count +
    ↪ neumann_ion.global_dof_count):(dirichlet_ion.global_dof_count +
    ↪ neumann_ion.global_dof_count + dirichlet_stern.global_dof_count)])
39
40     rhs_sigma = V_L_I_a_a * solution_neumann_ion - (- 0.5 * identity_ion + K_L_I_a_a)
    ↪ * solution_dirichlet_ion
41
42     sigma, info_2 = bempp.api.linalg.gmres(V_L_I_a_a, rhs_sigma, tol=TOL, maxiter =
    ↪ MAXITER_GM)

```

La energía de solvatación del sistema se calcula de la misma manera que en [A.3.1](#).

A.3.3 Formulación SLIC con la Mejora Propuesta

Para esta formulación, se debe mapear la distribución σ_{d1} a la superficie de la capa de exclusión iónica y crear un operador integral que contenga los valores de la distribución:

```

1 @bempp.api.real_callable
2 def d1_function(x, n, domain_index, result):
3     result[:] = np.sum(atoms[:,3]/(impl1(x-atoms[:,3])))

```

```

4
5 d1_fun = bempp.api.GridFunction(neumann_stern, fun = d1_function)
6 if abs(qtotal) < 1e-8:
7     d1_mat = -(1/ep_ii) * d1_fun.coefficients
8 else:
9     d1_mat = -(qtotal/ep_ii) * d1_fun.coefficients/np.mean(d1_fun.coefficients)
10 d1_gridfun = bempp.api.GridFunction(neumann_stern, coefficients = d1_mat)
11 d1_op = bempp.api.assembly.boundary_operator.MultiplicationOperator(d1_gridfun,
    ↪ neumann_stern, neumann_stern, neumann_stern)

```

La iteración principal varía al incluir la distribución de carga:

```

1 for i in range(ITER):
2     blocked = bempp.api.BlockedOperator(4, 4)
3     if i != 0:
4         d2 = solution_neumann_stern.integrate()[0]
5         f_it = f(sigma,en_1)
6         f_fun = bempp.api.GridFunction(neumann_ion, coefficients = f_it)
7         f_op = bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun,
            ↪ neumann_ion, neumann_ion, neumann_ion)
8         blocked[1,1] = V_L_II_a_a * f_op
9         blocked[2,1] = V_L_II_b_a * f_op
10        blocked[3,3] = (1/d2) * V_Y_III_b_b * d1_op
11    else:
12        blocked[1,1] = V_L_II_a_a
13        blocked[2,1] = V_L_II_b_a
14        blocked[3,3] = (1/d2) * V_Y_III_b_b

```

```

15
16     blocked[0,0] = 0.5 * identity_ion + K_L_I_a_a
17     blocked[0,1] = - V_L_I_a_a
18
19     blocked[1,0] = 0.5 * identity_ion - K_L_II_a_a
20     blocked[1,2] = K_L_II_a_b
21     blocked[1,3] = - V_L_II_a_b
22
23     blocked[2,0] = - K_L_II_b_a
24     blocked[2,2] = 0.5 * identity_stern + K_L_II_b_b
25     blocked[2,3] = - V_L_II_b_b
26
27     blocked[3,2] = 0.5 * identity_stern - K_Y_III_b_b
28
29     blocked = blocked.strong_form()
30
31     x, info_1 = gmres(blocked, rhs, maxiter=MAXITER_GM, tol = TOL)
32
33     solution_neumann_stern = bempp.api.GridFunction(neumann_stern, coefficients =
34     ↪ x[(dirichlet_ion.global_dof_count + neumann_ion.global_dof_count +
35     ↪ dirichlet_stern.global_dof_count):])
36
37     solution_neumann_ion = bempp.api.GridFunction(neumann_ion, coefficients =
38     ↪ x[dirichlet_ion.global_dof_count:(dirichlet_ion.global_dof_count +
39     ↪ neumann_ion.global_dof_count)])
40
41     solution_dirichlet_ion = bempp.api.GridFunction(dirichlet_ion, coefficients =
42     ↪ x[:dirichlet_ion.global_dof_count])

```

```

38
39     solution_dirichlet_stern = bempp.api.GridFunction(dirichlet_stern, coefficients
↪     = x[(dirichlet_ion.global_dof_count +
↪     neumann_ion.global_dof_count):(dirichlet_ion.global_dof_count +
↪     neumann_ion.global_dof_count + dirichlet_stern.global_dof_count)])
40
41     rhs_sigma = V_L_I_a_a * solution_neumann_ion - (- 0.5 * identity_ion + K_L_I_a_a)
↪     * solution_dirichlet_ion
42
43     sigma, info_2 = bempp.api.linalg.gmres(V_L_I_a_a, rhs_sigma, tol=TOL, maxiter =
↪     MAXITER_GM)

```

Nuevamente, se utiliza el código de la sección [A.3.1](#) para calcular la energía de solvatación del sistema.

A.4 Configuración 2: Interfaces Dieléctricas Separadas y *Stern Layer* Compartida

De igual forma que para el caso descrito en la sección [A.3](#), se deben importar las mallas, crear los espacios funcionales en bempp, definir las constantes para el problema y los operadores integrales a utilizar:

```

1  grid_dielectric_1 = import_msms_mesh('dielectric_1.face', 'dielectric_1.vert')
2  grid_dielectric_2 = import_msms_mesh('dielectric_2.face', 'dielectric_2.vert')
3  grid_stern = import_msms_mesh('stern.face', 'stern.vert')
4

```

```

5  dirichlet_ion_1 = bempp.api.function_space(grid_dielectric_1, 'DP', 0)
6  neumann_ion_1 = bempp.api.function_space(grid_dielectric_1, 'DP', 0)
7
8  dirichlet_ion_2 = bempp.api.function_space(grid_dielectric_2, 'DP', 0)
9  neumann_ion_2 = bempp.api.function_space(grid_dielectric_2, 'DP', 0)
10
11 dirichlet_stern = bempp.api.function_space(grid_stern, 'DP', 0)
12 neumann_stern = bempp.api.function_space(grid_stern, 'DP', 0)

```

```

1  om = 0
2  ep_iv = 80
3  ep_iii = 80
4  ep_ii = 1
5  ep_i = 1

```

```

1  identity_ion_1 = bempp.api.operators.boundary.sparse.identity(dirichlet_ion_1,
    ↪ dirichlet_ion_1, neumann_ion_1)
2  K_L_I_a_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_1,
    ↪ dirichlet_ion_1, neumann_ion_1)
3  K_L_I_a_a_adjoint =
    ↪ bempp.api.operators.boundary.laplace.adjoint_double_layer(dirichlet_ion_1,
    ↪ dirichlet_ion_1, neumann_ion_1)
4  G_L_I_a_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_1,
    ↪ dirichlet_ion_1, neumann_ion_1)

```

```

5 K_L_III_a_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_1,
    ↪ dirichlet_ion_1, neumann_ion_1)
6 G_L_III_a_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_1,
    ↪ dirichlet_ion_1, neumann_ion_1)
7 K_L_III_a_c = bempp.api.operators.boundary.laplace.double_layer(dirichlet_stern,
    ↪ dirichlet_ion_1, neumann_ion_1)
8 G_L_III_a_c = bempp.api.operators.boundary.laplace.single_layer(neumann_stern,
    ↪ dirichlet_ion_1, neumann_ion_1)
9 K_L_III_a_b = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_2,
    ↪ dirichlet_ion_1, neumann_ion_1)
10 G_L_III_a_b = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_2,
    ↪ dirichlet_ion_1, neumann_ion_1)
11 identity_ion_2 = bempp.api.operators.boundary.sparse.identity(dirichlet_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
12 K_L_II_b_b = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
13 K_L_II_b_b_adjoint =
    ↪ bempp.api.operators.boundary.laplace.adjoint_double_layer(dirichlet_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
14 G_L_II_b_b = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
15 K_L_III_b_c = bempp.api.operators.boundary.laplace.double_layer(dirichlet_stern,
    ↪ dirichlet_ion_2, neumann_ion_2)
16 G_L_III_b_c = bempp.api.operators.boundary.laplace.single_layer(neumann_stern,
    ↪ dirichlet_ion_2, neumann_ion_2)
17 K_L_III_b_b = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)

```

```

18 G_L_III_b_b = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
19 K_L_III_b_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_1,
    ↪ dirichlet_ion_2, neumann_ion_2)
20 G_L_III_b_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_1,
    ↪ dirichlet_ion_2, neumann_ion_2)
21 K_L_III_c_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_1,
    ↪ dirichlet_stern, neumann_stern)
22 G_L_III_c_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_1,
    ↪ dirichlet_stern, neumann_stern)
23 identity_stern = bempp.api.operators.boundary.sparse.identity(dirichlet_stern,
    ↪ dirichlet_stern, neumann_stern)
24 K_L_III_c_c = bempp.api.operators.boundary.laplace.double_layer(dirichlet_stern,
    ↪ dirichlet_stern, neumann_stern)
25 G_L_III_c_c = bempp.api.operators.boundary.laplace.single_layer(neumann_stern,
    ↪ dirichlet_stern, neumann_stern)
26 K_L_III_c_b = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_2,
    ↪ dirichlet_stern, neumann_stern)
27 G_L_III_c_b = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_2,
    ↪ dirichlet_stern, neumann_stern)
28 K_Y_IV_c_c =
    ↪ bempp.api.operators.boundary.modified_helmholtz.double_layer(dirichlet_stern,
    ↪ dirichlet_stern, neumann_stern, om)
29 G_Y_IV_c_c =
    ↪ bempp.api.operators.boundary.modified_helmholtz.single_layer(neumann_stern,
    ↪ dirichlet_stern, neumann_stern, om)

```

Se crean arreglos que contienen información sobre los átomos y se define la variable que contiene la carga total del sistema:

```
1 atoms = np.array([[0,0,0,q1],[x,0,0,q2]])
2 atoms_1 = np.array([atoms[0]])
3 atoms_2 = np.array([atoms[1]])
4 qtotal=np.sum(atoms[:,3])
```

Se mapean los elementos del lado derecho de los sistemas a sus respectivas mallas:

```
1 @bempp.api.real_callable
2 def rhs_function_1(x, n, domain_index, result):
3     result[:] = atoms_1[:,3]/(4 * np.pi * ep_i * impl1(x-atoms_1[:,3]))
4
5 @bempp.api.real_callable
6 def rhs_function_2(x, n, domain_index, result):
7     result[:] = atoms_2[:,3]/(4 * np.pi * ep_ii * impl1(x-atoms_2[:,3]))
8
9 rhs_1 = bempp.api.GridFunction(dirichlet_ion_1, fun=rhs_function_1)
10 rhs_2 = np.zeros(neumann_ion_1.global_dof_count)
11 rhs_3 = bempp.api.GridFunction(dirichlet_ion_2, fun=rhs_function_2)
12 rhs_4 = np.zeros(neumann_ion_2.global_dof_count)
13 rhs_5 = np.zeros(dirichlet_stern.global_dof_count)
14 rhs_6 = np.zeros(neumann_stern.global_dof_count)
15 rhs =
↪ np.concatenate([rhs_1.coefficients,rhs_2,rhs_3.coefficients,rhs_4,rhs_5,rhs_6])
```

A.4.1 Formulación PB

Se genera y pobla el operador bloqueado, se resuelve el sistema utilizando `gmres`:

```
1 blocked = bempp.api.BlockedOperator(6, 6)
2
3 blocked[0,0] = 0.5 * identity_ion_1 + K_L_I_a_a
4 blocked[0,1] = - G_L_I_a_a
5
6 blocked[1,0] = 0.5 * identity_ion_1 - K_L_III_a_a
7 blocked[1,1] = (ep_i/ep_iii) * G_L_III_a_a
8 blocked[1,2] = - K_L_III_a_b
9 blocked[1,3] = (ep_ii/ep_iii) * G_L_III_a_b
10 blocked[1,4] = K_L_III_a_c
11 blocked[1,5] = - G_L_III_a_c
12
13 blocked[2,2] = 0.5 * identity_ion_2 + K_L_II_b_b
14 blocked[2,3] = - G_L_II_b_b
15
16 blocked[3,0] = - K_L_III_b_a
17 blocked[3,1] = (ep_i/ep_iii) * G_L_III_b_a
18 blocked[3,2] = 0.5 * identity_ion_2 - K_L_III_b_b
19 blocked[3,3] = (ep_ii/ep_iii) * G_L_III_b_b
20 blocked[3,4] = K_L_III_b_c
21 blocked[3,5] = - G_L_III_b_c
22
23 blocked[4,0] = - K_L_III_c_a
```

```

24 blocked[4,1] = (ep_i/ep_iii) * G_L_III_c_a
25 blocked[4,2] = - K_L_III_c_b
26 blocked[4,3] = (ep_ii/ep_iii) * G_L_III_c_b
27 blocked[4,4] = 0.5 * identity_stern + K_L_III_c_c
28 blocked[4,5] = - G_L_III_c_c
29
30 blocked[5,4] = 0.5 * identity_stern - K_Y_IV_c_c
31 blocked[5,5] = (ep_iii/ep_iv) * G_Y_IV_c_c
32
33 blocked = blocked.strong_form()
34
35 x, info_1 = gmres(blocked, rhs, maxiter=MAXITER_GM, tol = TOL)

```

Luego, se extraen las soluciones para cada espacio funcional:

```

1 solution_dirichlet_ion_1 = bempp.api.GridFunction(dirichlet_ion_1, coefficients =
↪ x[:dirichlet_ion_1.global_dof_count])
2
3 solution_neumann_ion_1 = bempp.api.GridFunction(neumann_ion_1, coefficients =
↪ x[dirichlet_ion_1.global_dof_count:(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count)])
4
5 solution_dirichlet_ion_2 = bempp.api.GridFunction(dirichlet_ion_2, coefficients =
↪ x[(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_ion_2.global_dof_count)])
6

```

```

7 solution_neumann_ion_2 = bempp.api.GridFunction(neumann_ion_2, coefficients =
    ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
    ↪ dirichlet_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
    ↪ neumann_ion_1.global_dof_count + dirichlet_ion_2.global_dof_count +
    ↪ neumann_ion_2.global_dof_count)])
8
9 solution_dirichlet_stern = bempp.api.GridFunction(dirichlet_stern, coefficients =
    ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
    ↪ dirichlet_ion_2.global_dof_count +
    ↪ neumann_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
    ↪ neumann_ion_1.global_dof_count + dirichlet_ion_2.global_dof_count +
    ↪ neumann_ion_2.global_dof_count + dirichlet_stern.global_dof_count)])
10
11 solution_neumann_stern = bempp.api.GridFunction(neumann_stern, coefficients =
    ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
    ↪ dirichlet_ion_2.global_dof_count + neumann_ion_2.global_dof_count +
    ↪ dirichlet_stern.global_dof_count):])

```

Y se calcula la energía de solvatación del sistema:

```

1 S0_q_1 = bempp.api.operators.potential.laplace.single_layer(neumann_ion_1,
    ↪ atoms_1[:, :3].transpose())
2 D0_q_1 = bempp.api.operators.potential.laplace.double_layer(dirichlet_ion_1,
    ↪ atoms_1[:, :3].transpose())
3 phi_q_1 = S0_q_1 * solution_neumann_ion_1 - D0_q_1 * solution_dirichlet_ion_1
4 en_sol_1 = 2 * np.pi * 332.064 * np.sum(atoms_1[:, :3] * phi_q_1).real
5

```

```

6 S0_q_2 = bempp.api.operators.potential.laplace.single_layer(neumann_ion_2,
↳ atoms_2[:, :3].transpose())
7 D0_q_2 = bempp.api.operators.potential.laplace.double_layer(dirichlet_ion_2,
↳ atoms_2[:, :3].transpose())
8 phi_q_2 = S0_q_2 * solution_neumann_ion_2 - D0_q_2 * solution_dirichlet_ion_2
9 en_sol_2 = 2 * np.pi * 332.064 * np.sum(atoms_2[:, :3] * phi_q_2).real
10
11 en_sol = en_sol_1 + en_sol_2

```

A.4.2 Formulación SLIC

Se llevan a las mallas respectivas los primeros términos de las ecuaciones de campo eléctrico (3.30):

```

1 @bempp.api.real_callable
2 def en_function_1(x, n, domain_index, result):
3     result[:] = np.sum(np.dot(n, (np.transpose(x-atoms_1[:, :3]) / (4 * np.pi *
↳ impl1(x-atoms_1[:, :3])**3)) * (atoms_1[:, :3])))
4
5 @bempp.api.real_callable
6 def en_function_2(x, n, domain_index, result):
7     result[:] = np.sum(np.dot(n, (np.transpose(x-atoms_2[:, :3]) / (4 * np.pi *
↳ impl1(x-atoms_2[:, :3])**3)) * (atoms_2[:, :3])))
8
9 en_1 = bempp.api.GridFunction(dirichlet_ion_1, fun=en_function_1)
10 en_2 = bempp.api.GridFunction(dirichlet_ion_2, fun=en_function_2)

```

Se crea una función f que recibe como argumentos una distribución de carga σ , el primer término de la ecuación campo eléctrico (fun), un operador de doble capa adjunto (K) y las constantes dieléctricas interiores y exteriores de una superficie (ep_in y ep_out) y que retorna un arreglo con los valores de $\frac{f}{1+f}$. Se define el valor de d_1 y los valores iniciales de d_2 y de las distribuciones de carga aparente (σ_{Γ_a} y σ_{Γ_b}):

```

1 def f(sigma, fun, K, ep_in, ep_out):
2     Ksigma = K * sigma
3     En = fun - Ksigma
4     mu = - alpha * np.tanh(- gamma)
5     h = alpha * np.tanh(beta * En.coefficients - gamma) + mu
6     f = ep_in/(ep_out - ep_in) - h
7     f_div = f/(1+f)
8     return f_div
9
10 d2 = 1
11
12 d1 = - qtotal/ep_iii
13
14 sigma_1 = bempp.api.GridFunction(dirichlet_ion_1, coefficients =
15     ↪ np.zeros(dirichlet_ion_1.global_dof_count))
16
17 sigma_2 = bempp.api.GridFunction(dirichlet_ion_2, coefficients =
18     ↪ np.zeros(dirichlet_ion_2.global_dof_count))

```

Luego, se itera sobre los sistemas 3.24 y 3.31:

```

1  for i in range(ITER):
2      blocked = bempp.api.BlockedOperator(6, 6)
3      if i != 0:
4          d2 = solution_neumann_stern.integrate()[0]
5          f_it_1 = f(sigma_1,en_1,K_L_I_a_a_adjoint,ep_i,ep_iii)
6          f_fun_1 = bempp.api.GridFunction(neumann_ion_1, coefficients = f_it_1)
7          f_op_1 =
8          ↪ bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun_1,
9                                  neumann_ion_1, neumann_ion_1, neumann_ion_1)
10         f_it_2 = f(sigma_2,en_2,K_L_II_b_b_adjoint,ep_ii,ep_iii)
11         f_fun_2 = bempp.api.GridFunction(neumann_ion_2, coefficients = f_it_2)
12         f_op_2 =
13         ↪ bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun_2,
14                                 neumann_ion_2, neumann_ion_2, neumann_ion_2)
15         blocked[1,1] = G_L_III_a_a * f_op_1
16         blocked[1,3] = G_L_III_a_b * f_op_2
17         blocked[3,1] = G_L_III_b_a * f_op_1
18         blocked[3,3] = G_L_III_b_b * f_op_2
19         blocked[4,1] = G_L_III_c_a * f_op_1
20         blocked[4,3] = G_L_III_c_b * f_op_2
21     else:
22         blocked[1,1] = G_L_III_a_a
23         blocked[1,3] = G_L_III_a_b
24         blocked[3,1] = G_L_III_b_a
25         blocked[3,3] = G_L_III_b_b
26         blocked[4,1] = G_L_III_c_a
27         blocked[4,3] = G_L_III_c_b

```

```

26
27     blocked[0,0] = 0.5 * identity_ion_1 + K_L_I_a_a
28     blocked[0,1] = - G_L_I_a_a
29
30     blocked[1,0] = 0.5 * identity_ion_1 - K_L_III_a_a
31     blocked[1,2] = - K_L_III_a_b
32     blocked[1,4] = K_L_III_a_c
33     blocked[1,5] = - G_L_III_a_c
34
35     blocked[2,2] = 0.5 * identity_ion_2 + K_L_II_b_b
36     blocked[2,3] = - G_L_II_b_b
37
38     blocked[3,0] = - K_L_III_b_a
39     blocked[3,2] = 0.5 * identity_ion_2 - K_L_III_b_b
40     blocked[3,4] = K_L_III_b_c
41     blocked[3,5] = - G_L_III_b_c
42
43     blocked[4,0] = - K_L_III_c_a
44     blocked[4,2] = - K_L_III_c_b
45     blocked[4,4] = 0.5 * identity_stern + K_L_III_c_c
46     blocked[4,5] = - G_L_III_c_c
47
48     blocked[5,4] = 0.5 * identity_stern - K_Y_IV_c_c
49     blocked[5,5] = G_Y_IV_c_c * (d1/d2)
50
51     blocked = blocked.strong_form()
52

```

```

53 x, info_1 = gmres(blocked, rhs, maxiter=MAXITER_GM, tol=TOL)
54
55 solution_dirichlet_ion_1 = bempp.api.GridFunction(dirichlet_ion_1, coefficients
↪ = x[:dirichlet_ion_1.global_dof_count])
56
57 solution_neumann_ion_1 = bempp.api.GridFunction(neumann_ion_1, coefficients =
↪ x[dirichlet_ion_1.global_dof_count:(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count)])
58
59 solution_dirichlet_ion_2 = bempp.api.GridFunction(dirichlet_ion_2, coefficients
↪ = x[(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_ion_2.global_dof_count)])
60
61 solution_neumann_ion_2 = bempp.api.GridFunction(neumann_ion_2, coefficients =
↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
↪ dirichlet_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_ion_2.global_dof_count +
↪ neumann_ion_2.global_dof_count)])
62
63 solution_dirichlet_stern = bempp.api.GridFunction(dirichlet_stern, coefficients
↪ = x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
↪ dirichlet_ion_2.global_dof_count +
↪ neumann_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_ion_2.global_dof_count +
↪ neumann_ion_2.global_dof_count + dirichlet_stern.global_dof_count)])
64

```

```

65     solution_neumann_stern = bempp.api.GridFunction(neumann_stern, coefficients =
    ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
    ↪ dirichlet_ion_2.global_dof_count + neumann_ion_2.global_dof_count +
    ↪ dirichlet_stern.global_dof_count):])
66
67     rhs_sigma_1 = G_L_I_a_a * solution_neumann_ion_1 - (- 0.5 * identity_ion_1 +
    ↪ K_L_I_a_a) * solution_dirichlet_ion_1
68     rhs_sigma_2 = G_L_II_b_b * solution_neumann_ion_2 - (- 0.5 * identity_ion_2 +
    ↪ K_L_II_b_b) * solution_dirichlet_ion_2
69
70     sigma_1, info_2 = bempp.api.linalg.gmres(G_L_I_a_a, rhs_sigma_1,
    ↪ maxiter=MAXITER_GM, tol=TOL)
71     sigma_2, info_3 = bempp.api.linalg.gmres(G_L_II_b_b, rhs_sigma_2,
    ↪ maxiter=MAXITER_GM, tol=TOL)

```

La energía de solvatación se calcula de la misma manera que en la sección [A.4.1](#).

A.4.3 Formulación SLIC con la Mejora Propuesta

Se mapea la distribución σ_{d1} a la superficie de la *stern layer* y se crea un operador integral que contenga sus valores de igual forma que lo expuesto en la sección [A.3.3](#). La iteración para resolver los sistemas luce de la forma:

```

1  for i in range(ITER):
2      blocked = bempp.api.BlockedOperator(6, 6)
3      if i != 0:

```

```

4      d2 = solution_neumann_stern.integrate()[0]
5      f_it_1 = f(sigma_1,en_1,K_L_I_a_a_adjoint,ep_i,ep_iii)
6      f_fun_1 = bempp.api.GridFunction(neumann_ion_1, coefficients = f_it_1)
7      f_op_1 = bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun_1,
      ↪ neumann_ion_1, neumann_ion_1, neumann_ion_1)
8      f_it_2 = f(sigma_2,en_2,K_L_II_b_b_adjoint,ep_ii,ep_iii)
9      f_fun_2 = bempp.api.GridFunction(neumann_ion_2, coefficients = f_it_2)
10     f_op_2 = bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun_2,
      ↪ neumann_ion_2, neumann_ion_2, neumann_ion_2)
11     blocked[1,1] = G_L_III_a_a * f_op_1
12     blocked[1,3] = G_L_III_a_b * f_op_2
13     blocked[3,1] = G_L_III_b_a * f_op_1
14     blocked[3,3] = G_L_III_b_b * f_op_2
15     blocked[4,1] = G_L_III_c_a * f_op_1
16     blocked[4,3] = G_L_III_c_b * f_op_2
17     else:
18         blocked[1,1] = G_L_III_a_a
19         blocked[1,3] = G_L_III_a_b
20         blocked[3,1] = G_L_III_b_a
21         blocked[3,3] = G_L_III_b_b
22         blocked[4,1] = G_L_III_c_a
23         blocked[4,3] = G_L_III_c_b
24
25     blocked[0,0] = 0.5 * identity_ion_1 + K_L_I_a_a
26     blocked[0,1] = - G_L_I_a_a
27
28     blocked[1,0] = 0.5 * identity_ion_1 - K_L_III_a_a

```

```

29     blocked[1,2] = - K_L_III_a_b
30     blocked[1,4] = K_L_III_a_c
31     blocked[1,5] = - G_L_III_a_c
32
33     blocked[2,2] = 0.5 * identity_ion_2 + K_L_II_b_b
34     blocked[2,3] = - G_L_II_b_b
35
36     blocked[3,0] = - K_L_III_b_a
37     blocked[3,2] = 0.5 * identity_ion_2 - K_L_III_b_b
38     blocked[3,4] = K_L_III_b_c
39     blocked[3,5] = - G_L_III_b_c
40
41     blocked[4,0] = - K_L_III_c_a
42     blocked[4,2] = - K_L_III_c_b
43     blocked[4,4] = 0.5 * identity_stern + K_L_III_c_c
44     blocked[4,5] = - G_L_III_c_c
45
46     blocked[5,4] = 0.5 * identity_stern - K_Y_IV_c_c
47     blocked[5,5] = G_Y_IV_c_c * (d1/d2)
48
49     blocked = blocked.strong_form()
50
51     x, info_1 = gmres(blocked, rhs, maxiter=MAXITER_GM, tol=TOL)
52
53     solution_dirichlet_ion_1 = bempp.api.GridFunction(dirichlet_ion_1, coefficients
↪      = x[:dirichlet_ion_1.global_dof_count])
54

```

```

55 solution_neumann_ion_1 = bempp.api.GridFunction(neumann_ion_1, coefficients =
    ↪ x[dirichlet_ion_1.global_dof_count:(dirichlet_ion_1.global_dof_count +
    ↪ neumann_ion_1.global_dof_count)])
56
57 solution_dirichlet_ion_2 = bempp.api.GridFunction(dirichlet_ion_2, coefficients
    ↪ = x[(dirichlet_ion_1.global_dof_count +
    ↪ neumann_ion_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
    ↪ neumann_ion_1.global_dof_count + dirichlet_ion_2.global_dof_count)])
58
59 solution_neumann_ion_2 = bempp.api.GridFunction(neumann_ion_2, coefficients =
    ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
    ↪ dirichlet_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
    ↪ neumann_ion_1.global_dof_count + dirichlet_ion_2.global_dof_count +
    ↪ neumann_ion_2.global_dof_count)])
60
61 solution_dirichlet_stern = bempp.api.GridFunction(dirichlet_stern, coefficients
    ↪ = x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
    ↪ dirichlet_ion_2.global_dof_count +
    ↪ neumann_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
    ↪ neumann_ion_1.global_dof_count + dirichlet_ion_2.global_dof_count +
    ↪ neumann_ion_2.global_dof_count + dirichlet_stern.global_dof_count)])
62
63 solution_neumann_stern = bempp.api.GridFunction(neumann_stern, coefficients =
    ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
    ↪ dirichlet_ion_2.global_dof_count + neumann_ion_2.global_dof_count +
    ↪ dirichlet_stern.global_dof_count):])
64

```

```

65     rhs_sigma_1 = G_L_I_a_a * solution_neumann_ion_1 - (- 0.5 * identity_ion_1 +
↪ K_L_I_a_a) * solution_dirichlet_ion_1
66     rhs_sigma_2 = G_L_II_b_b * solution_neumann_ion_2 - (- 0.5 * identity_ion_2 +
↪ K_L_II_b_b) * solution_dirichlet_ion_2
67
68     sigma_1, info_2 = bempp.api.linalg.gmres(G_L_I_a_a, rhs_sigma_1,
↪ maxiter=MAXITER_GM, tol=TOL)
69     sigma_2, info_3 = bempp.api.linalg.gmres(G_L_II_b_b, rhs_sigma_2,
↪ maxiter=MAXITER_GM, tol=TOL)

```

La energía de solvatación se calcula de la misma manera que en las formulaciones anteriores.

A.5 Configuración 3: Interfaces Dieléctricas y *Stern Layers* Separadas

Se importan las mallas, se crean los espacios funcionales en Bempp y se definen las constantes para el problema y los operadores integrales a utilizar:

```

1     grid_dielectric_1 = import_msms_mesh('dielectric_1.face', 'dielectric_1.vert')
2     grid_dielectric_2 = import_msms_mesh('dielectric_2.face', 'dielectric_2.vert')
3
4     grid_stern_1 = import_msms_mesh('stern_1.face', 'stern_1.vert')
5     grid_stern_2 = import_msms_mesh('stern_2.face', 'stern_2.vert')
6
7     dirichlet_ion_1 = bempp.api.function_space(grid_dielectric_1, 'DP', 0)
8     neumann_ion_1 = bempp.api.function_space(grid_dielectric_1, 'DP', 0)
9

```

```

10  dirichlet_stern_1 = bempp.api.function_space(grid_stern_1, 'DP', 0)
11  neumann_stern_1 = bempp.api.function_space(grid_stern_1, 'DP', 0)
12
13  dirichlet_ion_2 = bempp.api.function_space(grid_dielectric_2, 'DP', 0)
14  neumann_ion_2 = bempp.api.function_space(grid_dielectric_2, 'DP', 0)
15
16  dirichlet_stern_2 = bempp.api.function_space(grid_stern_2, 'DP', 0)
17  neumann_stern_2 = bempp.api.function_space(grid_stern_2, 'DP', 0)
18
19  om = 0
20  ep_v = 80
21  ep_iv = 80
22  ep_iii = 1
23  ep_ii = 80
24  ep_i = 1

```

```

1  identity_ion_1 = bempp.api.operators.boundary.sparse.identity(dirichlet_ion_1,
↪  dirichlet_ion_1, neumann_ion_1)
2  K_L_I_a_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_1,
↪  dirichlet_ion_1, neumann_ion_1)
3  K_L_I_a_a_adjoint =
↪  bempp.api.operators.boundary.laplace.adjoint_double_layer(dirichlet_ion_1,
↪  dirichlet_ion_1, neumann_ion_1)
4  G_L_I_a_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_1,
↪  dirichlet_ion_1, neumann_ion_1)
5  K_L_II_a_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_1,
↪  dirichlet_ion_1, neumann_ion_1)

```

```

6  G_L_II_a_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_1,
    ↪  dirichlet_ion_1, neumann_ion_1)
7  K_L_II_a_b = bempp.api.operators.boundary.laplace.double_layer(dirichlet_stern_1,
    ↪  dirichlet_ion_1, neumann_ion_1)
8  G_L_II_a_b = bempp.api.operators.boundary.laplace.single_layer(neumann_stern_1,
    ↪  dirichlet_ion_1, neumann_ion_1)
9  K_L_II_b_a = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_1,
    ↪  dirichlet_stern_1, neumann_stern_1)
10 G_L_II_b_a = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_1,
    ↪  dirichlet_stern_1, neumann_stern_1)
11 identity_stern_1 = bempp.api.operators.boundary.sparse.identity(dirichlet_stern_1,
    ↪  dirichlet_stern_1, neumann_stern_1)
12 K_L_II_b_b = bempp.api.operators.boundary.laplace.double_layer(dirichlet_stern_1,
    ↪  dirichlet_stern_1, neumann_stern_1)
13 G_L_II_b_b = bempp.api.operators.boundary.laplace.single_layer(neumann_stern_1,
    ↪  dirichlet_stern_1, neumann_stern_1)
14 K_Y_V_b_b =
    ↪  bempp.api.operators.boundary.modified_helmholtz.double_layer(dirichlet_stern_1,
    ↪  dirichlet_stern_1, neumann_stern_1, om)
15 G_Y_V_b_b =
    ↪  bempp.api.operators.boundary.modified_helmholtz.single_layer(neumann_stern_1,
    ↪  dirichlet_stern_1, neumann_stern_1, om)
16 K_Y_V_b_d =
    ↪  bempp.api.operators.boundary.modified_helmholtz.double_layer(dirichlet_stern_2,
    ↪  dirichlet_stern_1, neumann_stern_1, om)
17 G_Y_V_b_d =
    ↪  bempp.api.operators.boundary.modified_helmholtz.single_layer(neumann_stern_2,
    ↪  dirichlet_stern_1, neumann_stern_1, om)

```

```

18 identity_ion_2 = bempp.api.operators.boundary.sparse.identity(dirichlet_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
19 K_L_III_c_c = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
20 K_L_III_c_c_adjoint =
    ↪ bempp.api.operators.boundary.laplace.adjoint_double_layer(dirichlet_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
21 G_L_III_c_c = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
22 K_L_IV_c_c = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
23 G_L_IV_c_c = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
24 K_L_IV_c_d = bempp.api.operators.boundary.laplace.double_layer(dirichlet_stern_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
25 G_L_IV_c_d = bempp.api.operators.boundary.laplace.single_layer(neumann_stern_2,
    ↪ dirichlet_ion_2, neumann_ion_2)
26 K_L_IV_d_c = bempp.api.operators.boundary.laplace.double_layer(dirichlet_ion_2,
    ↪ dirichlet_stern_2, neumann_stern_2)
27 G_L_IV_d_c = bempp.api.operators.boundary.laplace.single_layer(neumann_ion_2,
    ↪ dirichlet_stern_2, neumann_stern_2)
28 identity_stern_2 = bempp.api.operators.boundary.sparse.identity(dirichlet_stern_2,
    ↪ dirichlet_stern_2, neumann_stern_2)
29 K_L_IV_d_d = bempp.api.operators.boundary.laplace.double_layer(dirichlet_stern_2,
    ↪ dirichlet_stern_2, neumann_stern_2)
30 G_L_IV_d_d = bempp.api.operators.boundary.laplace.single_layer(neumann_stern_2,
    ↪ dirichlet_stern_2, neumann_stern_2)

```

```

31 K_Y_V_d_b =
    ↪ bempp.api.operators.boundary.modified_helmholtz.double_layer(dirichlet_stern_1,
    ↪ dirichlet_stern_2, neumann_stern_2, om)
32 G_Y_V_d_b =
    ↪ bempp.api.operators.boundary.modified_helmholtz.single_layer(neumann_stern_1,
    ↪ dirichlet_stern_2, neumann_stern_2, om)
33 K_Y_V_d_d =
    ↪ bempp.api.operators.boundary.modified_helmholtz.double_layer(dirichlet_stern_2,
    ↪ dirichlet_stern_2, neumann_stern_2, om)
34 G_Y_V_d_d =
    ↪ bempp.api.operators.boundary.modified_helmholtz.single_layer(neumann_stern_2,
    ↪ dirichlet_stern_2, neumann_stern_2, om)

```

Se definen los arreglos con la información sobre los átomos, se define la carga total del sistema y la encerrada por cada *stern layer*:

```

1 atoms = np.array([[0,0,0,q1],[x,0,0,q2]])
2 qtotal=np.sum(atoms[:,3])
3 atoms_1 = np.array([atoms[0]])
4 atoms_2 = np.array([atoms[1]])
5 qtotal_1=np.sum(atoms_1[:,3])
6 qtotal_2=np.sum(atoms_2[:,3])

```

Y se crea el lado derecho del sistema

```

1  @bempp.api.real_callable
2  def rhs_function_1(x, n, domain_index, result):
3      result[:] = atoms_1[:,3]/(4 * np.pi * ep_i * impl1(x-atoms_1[:,3]))
4
5  @bempp.api.real_callable
6  def rhs_function_2(x, n, domain_index, result):
7      result[:] = atoms_2[:,3]/(4 * np.pi * ep_iii * impl1(x-atoms_2[:,3]))
8
9  rhs_1 = bempp.api.GridFunction(dirichlet_ion_1, fun=rhs_function_1)
10 rhs_2 = np.zeros(neumann_ion_1.global_dof_count)
11 rhs_3 = np.zeros(dirichlet_stern_1.global_dof_count)
12 rhs_4 = np.zeros(neumann_stern_1.global_dof_count)
13 rhs_5 = bempp.api.GridFunction(dirichlet_ion_2, fun=rhs_function_2)
14 rhs_6 = np.zeros(neumann_ion_2.global_dof_count)
15 rhs_7 = np.zeros(dirichlet_stern_2.global_dof_count)
16 rhs_8 = np.zeros(neumann_stern_2.global_dof_count)
17 rhs = np.concatenate([rhs_1.coefficients, rhs_2, rhs_3, rhs_4, rhs_5.coefficients,
↪  rhs_6, rhs_7, rhs_8])

```

A.5.1 Formulación PB

Se crea el operador bloqueado y luego se resuelve el sistema:

```

1  blocked = bempp.api.BlockedOperator(8, 8)
2
3  blocked[0,0] = 0.5 * identity_ion_1 + K_L_I_a_a
4  blocked[0,1] = - G_L_I_a_a

```

```

5
6 blocked[1,0] = 0.5 * identity_ion_1 - K_L_II_a_a
7 blocked[1,1] = (ep_i/ep_ii) * G_L_II_a_a
8 blocked[1,2] = K_L_II_a_b
9 blocked[1,3] = - G_L_II_a_b
10
11 blocked[2,0] = - K_L_II_b_a
12 blocked[2,1] = (ep_i/ep_ii) * G_L_II_b_a
13 blocked[2,2] = 0.5 * identity_stern_1 + K_L_II_b_b
14 blocked[2,3] = - G_L_II_b_b
15
16 blocked[3,2] = 0.5 * identity_stern_1 - K_Y_V_b_b
17 blocked[3,3] = (ep_ii/ep_v) * G_Y_V_b_b
18 blocked[3,6] = - K_Y_V_b_d
19 blocked[3,7] = (ep_iv/ep_v) * G_Y_V_b_d
20
21 blocked[4,4] = 0.5 * identity_ion_2 + K_L_III_c_c
22 blocked[4,5] = - G_L_III_c_c
23
24 blocked[5,4] = 0.5 * identity_ion_2 - K_L_IV_c_c
25 blocked[5,5] = (ep_iii/ep_iv) * G_L_IV_c_c
26 blocked[5,6] = K_L_IV_c_d
27 blocked[5,7] = - G_L_IV_c_d
28
29 blocked[6,4] = - K_L_IV_d_c
30 blocked[6,5] = (ep_iii/ep_iv) * G_L_IV_d_c
31 blocked[6,6] = 0.5 * identity_stern_2 + K_L_IV_d_d

```

```

32 blocked[6,7] = - G_L_IV_d_d
33
34 blocked[7,2] = - K_Y_V_d_b
35 blocked[7,3] = (ep_ii/ep_v) * G_Y_V_d_b
36 blocked[7,6] = 0.5 * identity_stern_2 - K_Y_V_d_d
37 blocked[7,7] = (ep_iv/ep_v) * G_Y_V_d_d
38
39 blocked = blocked.strong_form()
40
41 x, info_1 = gmres(blocked, rhs, maxiter=MAXITER_GM, tol = TOL)

```

Y se extraen las soluciones para luego calcular la energía de solvatación:

```

1 solution_dirichlet_ion_1 = bempp.api.GridFunction(dirichlet_ion_1, coefficients =
  ↪ x[:dirichlet_ion_1.global_dof_count])
2 solution_neumann_ion_1 = bempp.api.GridFunction(neumann_ion_1, coefficients =
  ↪ x[dirichlet_ion_1.global_dof_count:(dirichlet_ion_1.global_dof_count +
  ↪ neumann_ion_1.global_dof_count)])
3
4 solution_dirichlet_stern_1 = bempp.api.GridFunction(dirichlet_stern_1, coefficients
  ↪ = x[(dirichlet_ion_1.global_dof_count +
  ↪ neumann_ion_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
  ↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count)])
5 solution_neumann_stern_1 = bempp.api.GridFunction(neumann_stern_1, coefficients =
  ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
  ↪ dirichlet_stern_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
  ↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
  ↪ neumann_stern_1.global_dof_count)])

```

```

6
7 solution_dirichlet_ion_2 = bempp.api.GridFunction(dirichlet_ion_2, coefficients =
  ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
  ↪ dirichlet_stern_1.global_dof_count +
  ↪ neumann_stern_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
  ↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
  ↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count)])
8 solution_neumann_ion_2 = bempp.api.GridFunction(neumann_ion_2, coefficients =
  ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
  ↪ dirichlet_stern_1.global_dof_count + neumann_stern_1.global_dof_count +
  ↪ dirichlet_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
  ↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
  ↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count +
  ↪ neumann_ion_2.global_dof_count)])
9
10 solution_dirichlet_stern_2 = bempp.api.GridFunction(dirichlet_stern_2, coefficients
  ↪ = x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
  ↪ dirichlet_stern_1.global_dof_count + neumann_stern_1.global_dof_count +
  ↪ dirichlet_ion_2.global_dof_count +
  ↪ neumann_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
  ↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
  ↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count +
  ↪ neumann_ion_2.global_dof_count + dirichlet_stern_2.global_dof_count)])
11 solution_neumann_stern_2 = bempp.api.GridFunction(neumann_stern_2, coefficients =
  ↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
  ↪ dirichlet_stern_1.global_dof_count + neumann_stern_1.global_dof_count +
  ↪ dirichlet_ion_2.global_dof_count + neumann_ion_2.global_dof_count +
  ↪ dirichlet_stern_2.global_dof_count):])

```

```

12
13 S0_q_1 = bempp.api.operators.potential.laplace.single_layer(neumann_ion_1,
↪ atoms_1[:, :3].transpose())
14 D0_q_1 = bempp.api.operators.potential.laplace.double_layer(dirichlet_ion_1,
↪ atoms_1[:, :3].transpose())
15 phi_q_1 = S0_q_1 * solution_neumann_ion_1 - D0_q_1 * solution_dirichlet_ion_1
16 en_sol_1 = 2 * np.pi * 332.064 * np.sum(atoms_1[:, :3] * phi_q_1).real
17
18 S0_q_2 = bempp.api.operators.potential.laplace.single_layer(neumann_ion_2,
↪ atoms_2[:, :3].transpose())
19 D0_q_2 = bempp.api.operators.potential.laplace.double_layer(dirichlet_ion_2,
↪ atoms_2[:, :3].transpose())
20 phi_q_2 = S0_q_2 * solution_neumann_ion_2 - D0_q_2 * solution_dirichlet_ion_2
21 en_sol_2 = 2 * np.pi * 332.064 * np.sum(atoms_2[:, :3] * phi_q_2).real
22
23 en_sol = en_sol_1 + en_sol_2

```

A.5.2 Formulaci3n SLIC

Se llevan a las mallas los t3rminos de las ecuaciones de campo el3ctrico, se definen los valores de $d1$ y $d3$, los valores iniciales de $d2$, de $d4$ y de las distribuciones σ y, utilizando una funci3n f similar a la definida en [A.4.2](#), se itera sobre los sistemas [3.34](#) y [3.41](#):

```

1 @bempp.api.real_callable
2 def en_function_1(x, n, domain_index, result):
3     result[:] = np.sum(np.dot(n,(np.transpose(x-atoms_1[:, :3]))/(4 * np.pi *
4         ↪ impl1(x-atoms_1[:, :3])**3)) * (atoms_1[:, :3])))
5
6 @bempp.api.real_callable
7 def en_function_2(x, n, domain_index, result):
8     result[:] = np.sum(np.dot(n,(np.transpose(x-atoms_2[:, :3]))/(4 * np.pi *
9         ↪ impl1(x-atoms_2[:, :3])**3)) * (atoms_2[:, :3])))
10
11
12 en_1 = bempp.api.GridFunction(dirichlet_ion_1, fun=en_function_1)
13 en_2 = bempp.api.GridFunction(dirichlet_ion_2, fun=en_function_2)
14
15
16 d2 = 1
17 d4 = 1
18
19 d1 = - qtotal_1/ep_ii
20 d3 = - qtotal_2/ep_iv
21
22
23 sigma_1 = bempp.api.GridFunction(dirichlet_ion_1, coefficients =
24     ↪ np.zeros(dirichlet_ion_1.global_dof_count))
25 sigma_2 = bempp.api.GridFunction(dirichlet_ion_2, coefficients =
26     ↪ np.zeros(dirichlet_ion_2.global_dof_count))
27
28
29 for i in range(ITER):
30     blocked = bempp.api.BlockedOperator(8, 8)
31     if i != 0:

```

```

24     d2 = solution_neumann_stern_1.integrate()[0]
25     d4 = solution_neumann_stern_2.integrate()[0]
26     f_it_1 = f(sigma_1,en_1,K_L_I_a_a_adjoint,ep_i,ep_ii)
27     f_fun_1 = bempp.api.GridFunction(neumann_ion_1, coefficients = f_it_1)
28     f_op_1 =
    ↪ bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun_1,
29                                     neumann_ion_1, neumann_ion_1, neumann_ion_1)
30     f_it_2 = f(sigma_2,en_2,K_L_III_c_c_adjoint,ep_iii,ep_iv)
31     f_fun_2 = bempp.api.GridFunction(neumann_ion_2, coefficients = f_it_2)
32     f_op_2 =
    ↪ bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun_2,
33                                     neumann_ion_2, neumann_ion_2, neumann_ion_2)
34     blocked[1,1] = G_L_II_a_a * f_op_1
35     blocked[2,1] = G_L_II_b_a * f_op_1
36     blocked[3,3] = (d1/d2) * G_Y_V_b_b
37     blocked[3,7] = (d3/d4) * G_Y_V_b_d
38     blocked[5,5] = G_L_IV_c_c * f_op_2
39     blocked[6,5] = G_L_IV_d_c * f_op_2
40     blocked[7,3] = (d1/d2) * G_Y_V_d_b
41     blocked[7,7] = (d3/d4) * G_Y_V_d_d
42
43     else:
44         blocked[1,1] = G_L_II_a_a
45         blocked[2,1] = G_L_II_b_a
46         blocked[3,3] = G_Y_V_b_b
47         blocked[3,7] = G_Y_V_b_d
48         blocked[5,5] = G_L_IV_c_c

```

```

49         blocked[6,5] = G_L_IV_d_c
50         blocked[7,3] = G_Y_V_d_b
51         blocked[7,7] = G_Y_V_d_d
52
53         blocked[0,0] = 0.5 * identity_ion_1 + K_L_I_a_a
54         blocked[0,1] = - G_L_I_a_a
55
56         blocked[1,0] = 0.5 * identity_ion_1 - K_L_II_a_a
57         blocked[1,2] = K_L_II_a_b
58         blocked[1,3] = - G_L_II_a_b
59
60         blocked[2,0] = - K_L_II_b_a
61         blocked[2,2] = 0.5 * identity_stern_1 + K_L_II_b_b
62         blocked[2,3] = - G_L_II_b_b
63
64         blocked[3,2] = 0.5 * identity_stern_1 - K_Y_V_b_b
65         blocked[3,6] = - K_Y_V_b_d
66
67
68         blocked[4,4] = 0.5 * identity_ion_2 + K_L_III_c_c
69         blocked[4,5] = - G_L_III_c_c
70
71         blocked[5,4] = 0.5 * identity_ion_2 - K_L_IV_c_c
72         blocked[5,6] = K_L_IV_c_d
73         blocked[5,7] = - G_L_IV_c_d
74
75         blocked[6,4] = - K_L_IV_d_c

```

```

76     blocked[6,6] = 0.5 * identity_stern_2 + K_L_IV_d_d
77     blocked[6,7] = - G_L_IV_d_d
78
79     blocked[7,2] = - K_Y_V_d_b
80     blocked[7,6] = 0.5 * identity_stern_2 - K_Y_V_d_d
81
82     blocked = blocked.strong_form()
83
84     x, info_1 = gmres(blocked, rhs, maxiter=MAXITER_GM, tol = TOL)
85
86     solution_dirichlet_ion_1 = bempp.api.GridFunction(dirichlet_ion_1, coefficients
87     ↪ = x[:dirichlet_ion_1.global_dof_count])
88
89     solution_neumann_ion_1 = bempp.api.GridFunction(neumann_ion_1, coefficients =
90     ↪ x[dirichlet_ion_1.global_dof_count:(dirichlet_ion_1.global_dof_count +
91     ↪ neumann_ion_1.global_dof_count)])
92
93     solution_dirichlet_stern_1 = bempp.api.GridFunction(dirichlet_stern_1,
94     ↪ coefficients = x[(dirichlet_ion_1.global_dof_count +
95     ↪ neumann_ion_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
96     ↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count)])
97
98     solution_neumann_stern_1 = bempp.api.GridFunction(neumann_stern_1, coefficients
99     ↪ = x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
100    ↪ dirichlet_stern_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
101    ↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
102    ↪ neumann_stern_1.global_dof_count)])

```

93

94

```

solution_dirichlet_ion_2 = bempp.api.GridFunction(dirichlet_ion_2, coefficients
↪ = x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
↪ dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count)])

```

95

96

```

solution_neumann_ion_2 = bempp.api.GridFunction(neumann_ion_2, coefficients =
↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
↪ dirichlet_stern_1.global_dof_count + neumann_stern_1.global_dof_count +
↪ dirichlet_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count +
↪ neumann_ion_2.global_dof_count)])

```

97

98

```

solution_dirichlet_stern_2 = bempp.api.GridFunction(dirichlet_stern_2,
↪ coefficients = x[(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count +
↪ neumann_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count +
↪ neumann_ion_2.global_dof_count + dirichlet_stern_2.global_dof_count)])

```

99

100

101

```

102     solution_neumann_stern_2 = bempp.api.GridFunction(neumann_stern_2, coefficients
↪     = x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
↪     dirichlet_stern_1.global_dof_count + neumann_stern_1.global_dof_count +
↪     dirichlet_ion_2.global_dof_count + neumann_ion_2.global_dof_count +
↪     dirichlet_stern_2.global_dof_count):])
103
104     rhs_sigma_1 = G_L_I_a_a * solution_neumann_ion_1 - (- 0.5 * identity_ion_1 +
↪     K_L_I_a_a) * solution_dirichlet_ion_1
105
106     rhs_sigma_2 = G_L_III_c_c * solution_neumann_ion_2 - (- 0.5 * identity_ion_2 +
↪     K_L_III_c_c) * solution_dirichlet_ion_2
107
108     sigma_1, info_2 = bempp.api.linalg.gmres(G_L_I_a_a, rhs_sigma_1, maxiter =
↪     MAXITER_GM, tol=TOL)
109
110     sigma_2, info_3 = bempp.api.linalg.gmres(G_L_III_c_c, rhs_sigma_2, maxiter =
↪     MAXITER_GM, tol=TOL)

```

La energía de solvatación del sistema se calcula de igual forma que en [A.5.1](#)

A.5.3 Formulación SLIC con la Mejora Propuesta

Se mapean las distribuciones σ_{d1} y σ_{d3} a sus respectivas mallas y se crean sus operadores integrales asociados:

```

1 @bempp.api.real_callable
2 def d1_function(x, n, domain_index, result):
3     result[:] = np.sum(atoms_1[:,3]/(impl1(x-atoms_1[:,3])))
4
5 d1_fun = bempp.api.GridFunction(neumann_stern_1, fun = d1_function)
6 d1_mat = -(qtotal_1/ep_ii) * d1_fun.coefficients/np.mean(d1_fun.coefficients)
7 d1_gridfun = bempp.api.GridFunction(neumann_stern_1, coefficients = d1_mat)
8 d1_op = bempp.api.assembly.boundary_operator.MultiplicationOperator(d1_gridfun,
↪ neumann_stern_1, neumann_stern_1, neumann_stern_1)
9
10 @bempp.api.real_callable
11 def d3_function(x, n, domain_index, result):
12     result[:] = np.sum(atoms_2[:,3]/(impl1(x-atoms_2[:,3])))
13
14 d3_fun = bempp.api.GridFunction(neumann_stern_2, fun = d3_function)
15 d3_mat = -(qtotal_2/ep_iv) * d3_fun.coefficients/np.mean(d3_fun.coefficients)
16 d3_gridfun = bempp.api.GridFunction(neumann_stern_2, coefficients = d3_mat)
17 d3_op = bempp.api.assembly.boundary_operator.MultiplicationOperator(d3_gridfun,
↪ neumann_stern_2, neumann_stern_2, neumann_stern_2)

```

Y se itera sobre los sistemas a resolver:

```

1 for i in range(ITER):
2     blocked = bempp.api.BlockedOperator(8, 8)
3     if i != 0:
4         d2 = solution_neumann_stern_1.integrate()[0]
5         d4 = solution_neumann_stern_2.integrate()[0]

```

```

6      f_it_1 = f(sigma_1,en_1,K_L_I_a_a_adjoint,ep_i,ep_ii)
7      f_fun_1 = bempp.api.GridFunction(neumann_ion_1, coefficients = f_it_1)
8      f_op_1 =
        ↪ bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun_1,
9                                     neumann_ion_1, neumann_ion_1, neumann_ion_1)
10     f_it_2 = f(sigma_2,en_2,K_L_III_c_c_adjoint,ep_iii,ep_iv)
11     f_fun_2 = bempp.api.GridFunction(neumann_ion_2, coefficients = f_it_2)
12     f_op_2 =
        ↪ bempp.api.assembly.boundary_operator.MultiplicationOperator(f_fun_2,
13                                     neumann_ion_2, neumann_ion_2, neumann_ion_2)
14     blocked[1,1] = G_L_II_a_a * f_op_1
15     blocked[2,1] = G_L_II_b_a * f_op_1
16     blocked[3,3] = (1/d2) * G_Y_V_b_b * d1_op
17     blocked[3,7] = (1/d4) * G_Y_V_b_d * d3_op
18     blocked[5,5] = G_L_IV_c_c * f_op_2
19     blocked[6,5] = G_L_IV_d_c * f_op_2
20     blocked[7,3] = (1/d2) * G_Y_V_d_b * d1_op
21     blocked[7,7] = (1/d4) * G_Y_V_d_d * d3_op
22
23     else:
24         blocked[1,1] = G_L_II_a_a
25         blocked[2,1] = G_L_II_b_a
26         blocked[3,3] = G_Y_V_b_b
27         blocked[3,7] = G_Y_V_b_d
28         blocked[5,5] = G_L_IV_c_c
29         blocked[6,5] = G_L_IV_d_c
30         blocked[7,3] = G_Y_V_d_b

```

```

31         blocked[7,7] = G_Y_V_d_d
32
33
34         blocked[0,0] = 0.5 * identity_ion_1 + K_L_I_a_a
35         blocked[0,1] = - G_L_I_a_a
36
37         blocked[1,0] = 0.5 * identity_ion_1 - K_L_II_a_a
38         blocked[1,2] = K_L_II_a_b
39         blocked[1,3] = - G_L_II_a_b
40
41         blocked[2,0] = - K_L_II_b_a
42         blocked[2,2] = 0.5 * identity_stern_1 + K_L_II_b_b
43         blocked[2,3] = - G_L_II_b_b
44
45         blocked[3,2] = 0.5 * identity_stern_1 - K_Y_V_b_b
46         blocked[3,6] = - K_Y_V_b_d
47
48         blocked[4,4] = 0.5 * identity_ion_2 + K_L_III_c_c
49         blocked[4,5] = - G_L_III_c_c
50
51         blocked[5,4] = 0.5 * identity_ion_2 - K_L_IV_c_c
52         blocked[5,6] = K_L_IV_c_d
53         blocked[5,7] = - G_L_IV_c_d
54
55         blocked[6,4] = - K_L_IV_d_c
56         blocked[6,6] = 0.5 * identity_stern_2 + K_L_IV_d_d
57         blocked[6,7] = - G_L_IV_d_d

```

```

58
59     blocked[7,2] = - K_Y_V_d_b
60     blocked[7,6] = 0.5 * identity_stern_2 - K_Y_V_d_d
61
62     blocked = blocked.strong_form()
63
64     x, info_1 = gmres(blocked, rhs, maxiter=MAXITER_GM, tol = TOL)
65
66     solution_dirichlet_ion_1 = bempp.api.GridFunction(dirichlet_ion_1, coefficients
67     ↪ = x[:dirichlet_ion_1.global_dof_count])
68
69     solution_neumann_ion_1 = bempp.api.GridFunction(neumann_ion_1, coefficients =
70     ↪ x[dirichlet_ion_1.global_dof_count:(dirichlet_ion_1.global_dof_count +
71     ↪ neumann_ion_1.global_dof_count)])
72
73     solution_dirichlet_stern_1 = bempp.api.GridFunction(dirichlet_stern_1,
74     ↪ coefficients = x[(dirichlet_ion_1.global_dof_count +
75     ↪ neumann_ion_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
76     ↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count)])
77
78     solution_neumann_stern_1 = bempp.api.GridFunction(neumann_stern_1, coefficients
79     ↪ = x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
80     ↪ dirichlet_stern_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
81     ↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
82     ↪ neumann_stern_1.global_dof_count)])

```

75

```

76 solution_dirichlet_ion_2 = bempp.api.GridFunction(dirichlet_ion_2, coefficients
↪ = x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
↪ dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count)])

```

77

```

78 solution_neumann_ion_2 = bempp.api.GridFunction(neumann_ion_2, coefficients =
↪ x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
↪ dirichlet_stern_1.global_dof_count + neumann_stern_1.global_dof_count +
↪ dirichlet_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count +
↪ neumann_ion_2.global_dof_count)])

```

79

```

80 solution_dirichlet_stern_2 = bempp.api.GridFunction(dirichlet_stern_2,
↪ coefficients = x[(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count +
↪ neumann_ion_2.global_dof_count):(dirichlet_ion_1.global_dof_count +
↪ neumann_ion_1.global_dof_count + dirichlet_stern_1.global_dof_count +
↪ neumann_stern_1.global_dof_count + dirichlet_ion_2.global_dof_count +
↪ neumann_ion_2.global_dof_count + dirichlet_stern_2.global_dof_count)])

```

81

82

83

```

84     solution_neumann_stern_2 = bempp.api.GridFunction(neumann_stern_2, coefficients
↪     = x[(dirichlet_ion_1.global_dof_count + neumann_ion_1.global_dof_count +
↪     dirichlet_stern_1.global_dof_count + neumann_stern_1.global_dof_count +
↪     dirichlet_ion_2.global_dof_count + neumann_ion_2.global_dof_count +
↪     dirichlet_stern_2.global_dof_count):])

85
86     rhs_sigma_1 = G_L_I_a_a * solution_neumann_ion_1 - (- 0.5 * identity_ion_1 +
↪     K_L_I_a_a) * solution_dirichlet_ion_1

87
88     rhs_sigma_2 = G_L_III_c_c * solution_neumann_ion_2 - (- 0.5 * identity_ion_2 +
↪     K_L_III_c_c) * solution_dirichlet_ion_2

89
90     sigma_1, info_2 = bempp.api.linalg.gmres(G_L_I_a_a, rhs_sigma_1,
↪     maxiter=MAXITER_GM ,tol=TOL)

91
92     sigma_2, info_3 = bempp.api.linalg.gmres(G_L_III_c_c, rhs_sigma_2, tol=TOL,
↪     maxiter = MAXITER_GM)

```

La energía de solvatación del sistema se calcula según lo expuesto en [A.5.1](#).

B

Código para el Estudio de la Energía de Solvatación de Proteínas y Aminoácidos

B.1 Creación de Mallas

Las mallas utilizadas para simular la proteína y los aminoácidos fueron generadas usando el software NanoShaper, eligiendo una superficie excluída al sol-

vente, una esfera imaginaria de $1.4 \text{ [\AA]}$ y una capa de exclusión iónica de $2 \text{ [\AA]}$. El software recibe un archivo `.xyzr` y un archivo `config` con los parámetros elegidos para la malla y se llama desde el símbolo del sistema utilizando el comando:

```
NanoShaper64.exe ./config
```

Al igual que MSMS, NanoShaper crea archivos `.face` y `.vert` importables directamente a la librería Bempp. Las mallas creadas en esta sección tienen una densidad de al menos $4.5 \left[\frac{\text{caras}}{\text{\AA}^2} \right]$.

B.2 Simulación

Debido a la geometría del problema, el código utilizado para simular los aminoácidos y la proteína es en su mayoría idéntico al expuesto en la sección [A.3](#) (la primera configuración de superficies del problema de los dos átomos). Por esta razón, solo se exponen en este apéndice los bloques de código en los que este problema difiere del anteriormente mencionado.

B.2.1 Parámetros SLIC

Los parámetros SLIC elegidos para este caso se definen como:

```
1 alpha = 0.5
2 beta  = -60
3 gamma = -0.5
```

B.2.2 Definición de Constantes

Las constantes del problema son las siguientes:

```
1 om = 0.125
2 ep_i = 4
3 ep_ii = 80
4 ep_iii = 80
```

B.2.3 Información sobre las cargas

Debido a la cantidad de átomos que componen los aminoácidos y la proteína, la información sobre la posición y carga de los átomos es extraída de un archivo de texto .pqr. Este tipo de archivo incluye información sobre los átomos que componen la molécula: posición, carga, radio, entre otros parámetros. La información se extrae utilizando el código que sigue:

```
1 atoms=[]
2 pqr=open('archivo.pqr','r')
3 for line in pqr:
4     if line[:4]=="ATOM":
5         line_ = line.split()
6         x_c = [line_[5],line_[6],line_[7],line_[8]]
7         x_c=[float(i) for i in x_c]
8         atoms.append(x_c)
9 pqr.close()
```

```
10 atoms=np.array(atoms)
```

B.2.4 Precondicionadores

El preconditionador elegido es del tipo *block diagonal*. En un sistema:

$$Ax = b \quad (\text{B.1})$$

Donde A es una matriz de operadores integrales; el preconditionador *block diagonal* se genera tomando la diagonal de todos los operadores que se encuentran en la diagonal, primera subdiagonal y primera superdiagonal de A.

Debido a que en cada formulación el lado izquierdo del sistema es distinto, se deben crear preconditionadores específicos para el método PB, el método SLIC y el método SLIC con la mejora propuesta.

La función que crea el preconditionador en el método PB es la siguiente:

```
1 def block_diagonal_PB():
2     from scipy.sparse import diags, bmat, block_diag
3     from scipy.sparse.linalg import aslinearoperator
4     from bempp.api.operators.boundary import sparse, laplace, modified_helmholtz
5
6     # block-diagonal preconditioner
7     identity_protein = sparse.identity(dirichlet_protein, dirichlet_protein,
8     ↪ dirichlet_protein)
9     identity_protein_diag = identity_protein.weak_form().to_sparse().diagonal()
```

```

9      V_L_I_II_a_a_diag = laplace.single_layer(neumann_protein, dirichlet_protein,
    ↪   dirichlet_protein,
    ↪   assembler="only_diagonal_part").weak_form().get_diagonal()
10     K_L_I_II_a_a_diag = laplace.double_layer(dirichlet_protein, dirichlet_protein,
    ↪   dirichlet_protein,
    ↪   assembler="only_diagonal_part").weak_form().get_diagonal()
11     identity_stern = sparse.identity(dirichlet_stern, dirichlet_stern,
    ↪   dirichlet_stern)
12     identity_stern_diag = identity_stern.weak_form().to_sparse().diagonal()
13     V_L_II_b_b_diag = laplace.single_layer(neumann_stern, dirichlet_stern,
    ↪   dirichlet_stern, assembler="only_diagonal_part").weak_form().get_diagonal()
14     K_L_II_b_b_diag = laplace.double_layer(neumann_stern, dirichlet_stern,
    ↪   dirichlet_stern, assembler="only_diagonal_part").weak_form().get_diagonal()
15     V_Y_I_b_b_diag = modified_helmholtz.single_layer(neumann_stern, dirichlet_stern,
    ↪   dirichlet_stern, om,
    ↪   assembler="only_diagonal_part").weak_form().get_diagonal()
16     K_Y_I_b_b_diag = modified_helmholtz.double_layer(neumann_stern, dirichlet_stern,
    ↪   dirichlet_stern, om,
    ↪   assembler="only_diagonal_part").weak_form().get_diagonal()
17
18     diag11 = .5 * identity_protein_diag + K_L_I_II_a_a_diag
19     diag12 = -V_L_I_II_a_a_diag
20     diag21 = .5 * identity_protein_diag - K_L_I_II_a_a_diag
21     diag22 = (ep_i / ep_ii) * V_L_I_II_a_a_diag
22
23     d_aux = 1 / (diag22 - diag21 * diag12 / diag11)
24     diag11_inv = 1 / diag11 + 1 / diag11 * diag12 * d_aux * diag21 / diag11

```

```

25     diag12_inv = -1 / diag11 * diag12 * d_aux
26     diag21_inv = -d_aux * diag21 / diag11
27     diag22_inv = d_aux
28
29     diag33 = .5 * identity_stern_diag + K_L_II_b_b_diag
30     diag34 = -V_L_II_b_b_diag
31     diag43 = .5 * identity_stern_diag - K_Y_I_b_b_diag
32     diag44 = (ep_ii / ep_iii) * V_Y_I_b_b_diag
33
34     d_aux = 1 / (diag44 - diag43 * diag34 / diag33)
35     diag33_inv = 1 / diag33 + 1 / diag33 * diag34 * d_aux * diag43 / diag33
36     diag34_inv = -1 / diag33 * diag34 * d_aux
37     diag43_inv = -d_aux * diag43 / diag33
38     diag44_inv = d_aux
39
40     A = bmat([[diags(diag11_inv), diags(diag12_inv)], [diags(diag21_inv),
41     ↪ diags(diag22_inv)]])
42
43     B = bmat([[diags(diag33_inv), diags(diag34_inv)], [diags(diag43_inv),
44     ↪ diags(diag44_inv)]])
45
46     block_mat_precond = block_diag((A,B),format = 'csr')#.tocsr()
47
48     return aslinearoperator(block_mat_precond)

```

Para crear el preconditionador en el método SLIC sin la mejora propuesta se utiliza:

```

1 def block_diagonal_SLIC():
2     from scipy.sparse import diags, bmat, block_diag
3     from scipy.sparse.linalg import aslinearoperator
4     from bemptt.api.operators.boundary import sparse, laplace, modified_helmholtz
5
6     # block-diagonal preconditioner
7     identity_protein = sparse.identity(dirichlet_protein, dirichlet_protein,
8     ↪ dirichlet_protein)
9     identity_protein_diag = identity_protein.weak_form().to_sparse().diagonal()
10    V_L_I_II_a_a_diag = laplace.single_layer(neumann_protein, dirichlet_protein,
11    ↪ dirichlet_protein,
12    ↪ assembler="only_diagonal_part").weak_form().get_diagonal()
13    K_L_I_II_a_a_diag = laplace.double_layer(dirichlet_protein, dirichlet_protein,
14    ↪ dirichlet_protein,
15    ↪ assembler="only_diagonal_part").weak_form().get_diagonal()
16    identity_stern = sparse.identity(dirichlet_stern, dirichlet_stern,
17    ↪ dirichlet_stern)
18    identity_stern_diag = identity_stern.weak_form().to_sparse().diagonal()
19    V_L_II_b_b_diag = laplace.single_layer(neumann_stern, dirichlet_stern,
20    ↪ dirichlet_stern, assembler="only_diagonal_part").weak_form().get_diagonal()
21    K_L_II_b_b_diag = laplace.double_layer(neumann_stern, dirichlet_stern,
22    ↪ dirichlet_stern, assembler="only_diagonal_part").weak_form().get_diagonal()
23    V_Y_I_b_b_diag = modified_helmholtz.single_layer(neumann_stern, dirichlet_stern,
24    ↪ dirichlet_stern, om,
25    ↪ assembler="only_diagonal_part").weak_form().get_diagonal()
26    K_Y_I_b_b_diag = modified_helmholtz.double_layer(neumann_stern, dirichlet_stern,
27    ↪ dirichlet_stern, om,
28    ↪ assembler="only_diagonal_part").weak_form().get_diagonal()

```

```

17
18     diag11 = .5 * identity_protein_diag + K_L_I_II_a_a_diag
19     diag12 = -V_L_I_II_a_a_diag
20     diag21 = .5 * identity_protein_diag - K_L_I_II_a_a_diag
21     diag22 = V_L_I_II_a_a_diag * f_op.weak_form().to_sparse().diagonal()
22
23     d_aux = 1 / (diag22 - diag21 * diag12 / diag11)
24     diag11_inv = 1 / diag11 + 1 / diag11 * diag12 * d_aux * diag21 / diag11
25     diag12_inv = -1 / diag11 * diag12 * d_aux
26     diag21_inv = -d_aux * diag21 / diag11
27     diag22_inv = d_aux
28
29     diag33 = .5 * identity_stern_diag + K_L_II_b_b_diag
30     diag34 = -V_L_II_b_b_diag
31     diag43 = .5 * identity_stern_diag - K_Y_I_b_b_diag
32     diag44 = (d1 / d2) * V_Y_I_b_b_diag
33
34     d_aux = 1 / (diag44 - diag43 * diag34 / diag33)
35     diag33_inv = 1 / diag33 + 1 / diag33 * diag34 * d_aux * diag43 / diag33
36     diag34_inv = -1 / diag33 * diag34 * d_aux
37     diag43_inv = -d_aux * diag43 / diag33
38     diag44_inv = d_aux
39
40     A = bmat([[diags(diag11_inv), diags(diag12_inv)], [diags(diag21_inv),
↪   diags(diag22_inv)]])
41     B = bmat([[diags(diag33_inv), diags(diag34_inv)], [diags(diag43_inv),
↪   diags(diag44_inv)]])

```

```

42
43     block_mat_precond = block_diag((A,B),format = 'csr')#.tocsr()
44     return aslinearoperator(block_mat_precond)

```

Mientras que para el método SLIC con la mejora propuesta:

```

1  def block_diagonal_SLIC_PROP():
2      from scipy.sparse import diags, bmat, block_diag
3      from scipy.sparse.linalg import aslinearoperator
4      from bemptt.api.operators.boundary import sparse, laplace, modified_helmholtz
5
6      # block-diagonal preconditioner
7      identity_protein = sparse.identity(dirichlet_protein, dirichlet_protein,
8      ↪ dirichlet_protein)
9      identity_protein_diag = identity_protein.weak_form().to_sparse().diagonal()
10     V_L_I_II_a_a_diag = laplace.single_layer(neumann_protein, dirichlet_protein,
11     ↪ dirichlet_protein,
12     ↪ assembler="only_diagonal_part").weak_form().get_diagonal()
13     K_L_I_II_a_a_diag = laplace.double_layer(dirichlet_protein, dirichlet_protein,
14     ↪ dirichlet_protein,
15     ↪ assembler="only_diagonal_part").weak_form().get_diagonal()
16     identity_stern = sparse.identity(dirichlet_stern, dirichlet_stern,
17     ↪ dirichlet_stern)
18     identity_stern_diag = identity_stern.weak_form().to_sparse().diagonal()
19     V_L_II_b_b_diag = laplace.single_layer(neumann_stern, dirichlet_stern,
20     ↪ dirichlet_stern, assembler="only_diagonal_part").weak_form().get_diagonal()
21     K_L_II_b_b_diag = laplace.double_layer(neumann_stern, dirichlet_stern,
22     ↪ dirichlet_stern, assembler="only_diagonal_part").weak_form().get_diagonal()

```

```

15 V_Y_I_b_b_diag = modified_helmholtz.single_layer(neumann_stern, dirichlet_stern,
↳ dirichlet_stern, om,
↳ assembler="only_diagonal_part").weak_form().get_diagonal()
16 K_Y_I_b_b_diag = modified_helmholtz.double_layer(neumann_stern, dirichlet_stern,
↳ dirichlet_stern, om,
↳ assembler="only_diagonal_part").weak_form().get_diagonal()
17
18 diag11 = .5 * identity_protein_diag + K_L_I_II_a_a_diag
19 diag12 = -V_L_I_II_a_a_diag
20 diag21 = .5 * identity_protein_diag - K_L_I_II_a_a_diag
21 diag22 = V_L_I_II_a_a_diag * f_op.weak_form().to_sparse().diagonal()
22
23 d_aux = 1 / (diag22 - diag21 * diag12 / diag11)
24 diag11_inv = 1 / diag11 + 1 / diag11 * diag12 * d_aux * diag21 / diag11
25 diag12_inv = -1 / diag11 * diag12 * d_aux
26 diag21_inv = -d_aux * diag21 / diag11
27 diag22_inv = d_aux
28
29 diag33 = .5 * identity_stern_diag + K_L_II_b_b_diag
30 diag34 = -V_L_II_b_b_diag
31 diag43 = .5 * identity_stern_diag - K_Y_I_b_b_diag
32 diag44 = (1 / d2) * V_Y_I_b_b_diag * d1_op.weak_form().to_sparse().diagonal()
33
34 d_aux = 1 / (diag44 - diag43 * diag34 / diag33)
35 diag33_inv = 1 / diag33 + 1 / diag33 * diag34 * d_aux * diag43 / diag33
36 diag34_inv = -1 / diag33 * diag34 * d_aux
37 diag43_inv = -d_aux * diag43 / diag33

```

```

38     diag44_inv = d_aux
39
40     A = bmat([[diags(diag11_inv), diags(diag12_inv)], [diags(diag21_inv),
↪     diags(diag22_inv)]])
41     B = bmat([[diags(diag33_inv), diags(diag34_inv)], [diags(diag43_inv),
↪     diags(diag44_inv)]])
42
43     block_mat_precond = block_diag((A,B),format = 'csr')#.tocsr()
44     return aslinearoperator(block_mat_precond)

```

Es importante notar que en cada iteración del método SLIC y el método SLIC con la mejora propuesta, se debe crear nuevamente el preconditionador. Las funciones utilizadas para crear los preconditionadores fueron adaptadas del repositorio Github de PBJ¹⁶, una librería para resolver problemas de electrostática molecular utilizando el modelo de Poisson-Boltzmann.

B.2.5 Solución del Sistema

Para incluir el preconditionador en la resolución del sistema para la formulación PB, se debe crear una vez la matriz del preconditionador y llamar a la función `gmres` de `scipy` con un nuevo parámetro:

```

1     precondition_PB = block_diagonal_PB()
2     x, info_1 = gmres(blocked, rhs, M = precondition_PB, maxiter=MAXITER_GM, tol = TOL)

```

Mientras que para el caso de SLIC sin la mejora propuesta se debe crear en

cada iteración:

```
1  if i != 0:
2      precondition_SLIC = block_diagonal_SLIC()
3  else:
4      precondition_SLIC = precondition_PB
5  x_prior = x
6  x, info_1 = gmres(blocked, rhs, M = precondition_SLIC, x0 = x_prior, maxiter=MAXITER_GM,
    ↪  tol = TOL)
```

Siendo i el número de iteración. x_0 es un parámetro de `gmres` que permite utilizar una solución previa como punto de partida del método de resolución. x_{prior} es la solución al sistema de la iteración SLIC anterior o, en el caso de la primera iteración, la solución al sistema con la formulación PB.

Para el caso de SLIC con la mejora propuesta, el código es similar:

```
1  if i != 0:
2      precondition_SLIC_PROP = block_diagonal_SLIC_PROP()
3  else:
4      precondition_SLIC = precondition_PB
5  x_prior = x
6  x, info_1 = gmres(blocked, rhs, M = precondition_SLIC_PROP, x0 = x_prior,
    ↪  maxiter=MAXITER_GM, tol = TOL)
```



Gráficos de las Variables de Interés para el Problema de los Aminoácidos

A continuación, se muestran los gráficos de las variables de interés en las superficies que componen cada aminoácido. Notar que, en algunos casos, el subgráfico que contiene los valores del potencial en la *stern layer* posee dos ejes: el de

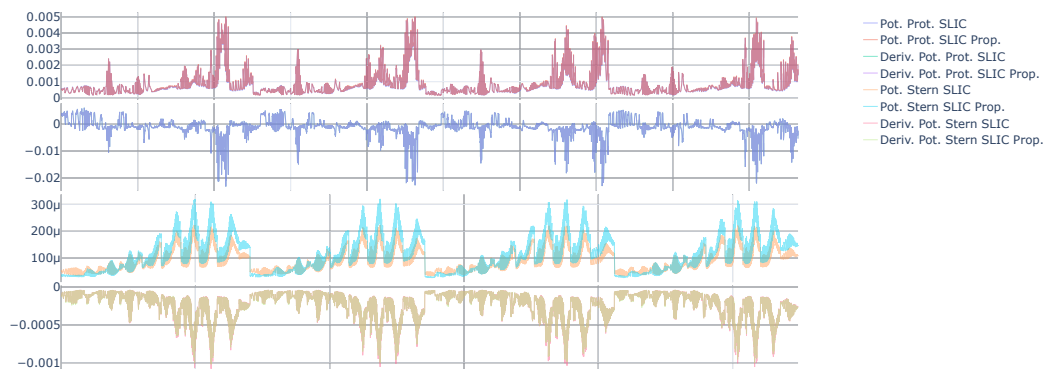


Figura C.1: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido arginina en su estado protonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

la izquierda se utiliza como referencia para los resultados calculados usando el modelo SLIC base y el de la derecha se utiliza como referencia para los valores calculados utilizando el modelo SLIC con la mejora propuesta.

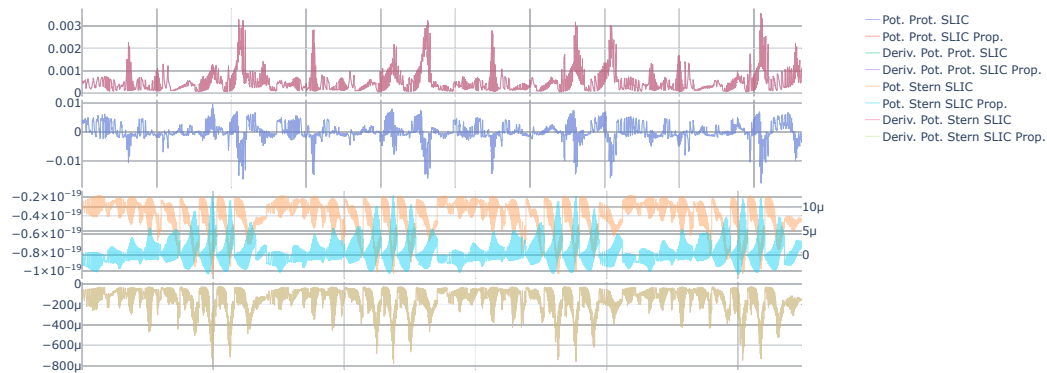


Figura C.2: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido arginina en su estado deprotonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

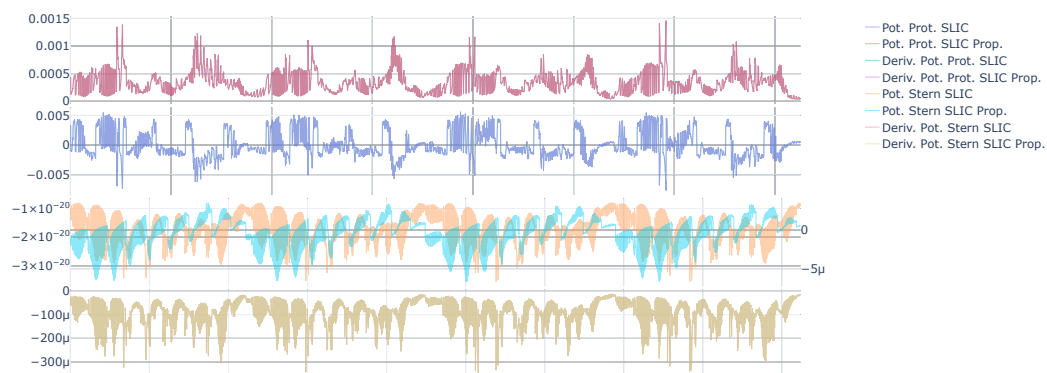


Figura C.3: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido ácido aspártico en su estado protonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

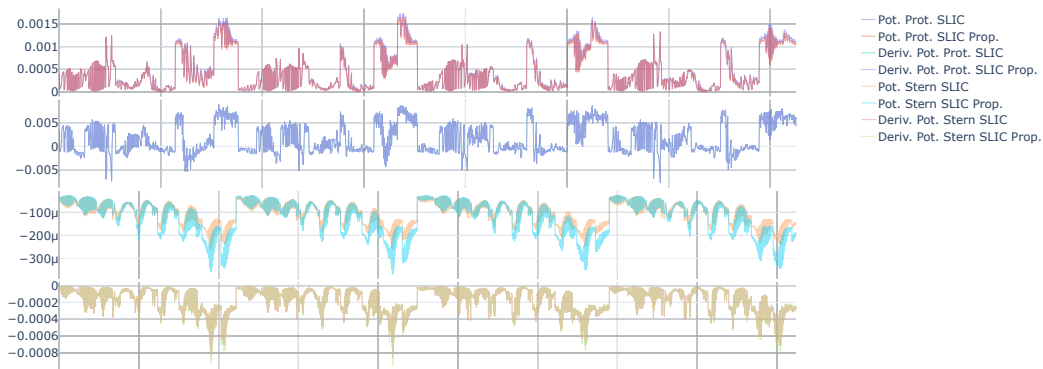


Figura C.4: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido ácido aspártico en su estado deprotonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

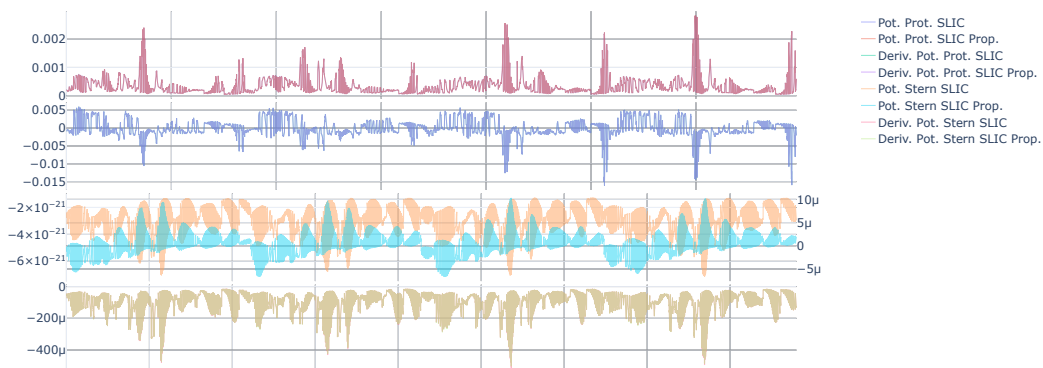


Figura C.5: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido cisteína en su estado protonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

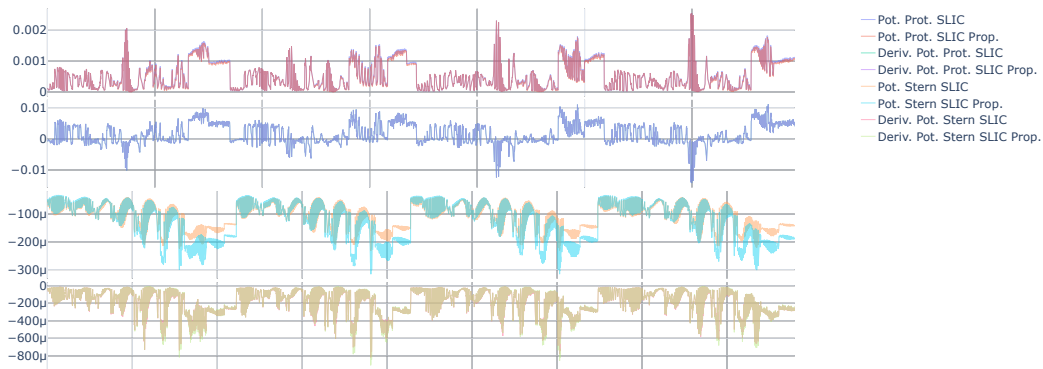


Figura C.6: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido cisteína en su estado deprotonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

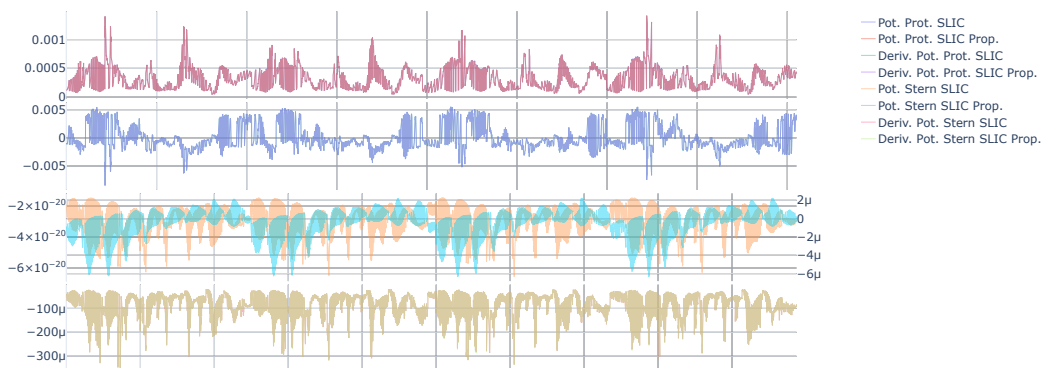


Figura C.7: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido ácido glutámico en su estado protonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

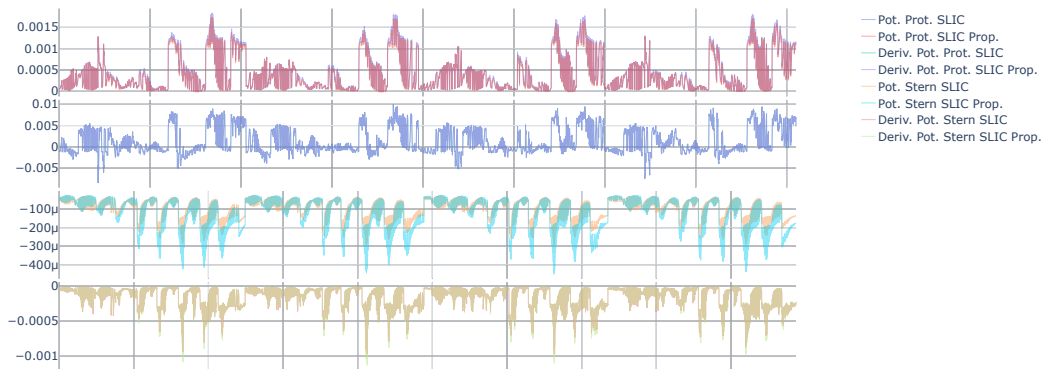


Figura C.8: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido ácido glutámico en su estado deprotonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

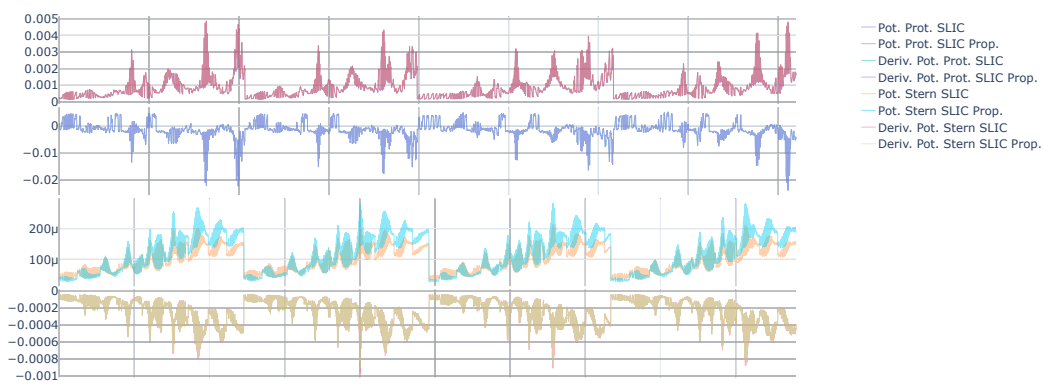


Figura C.9: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido histidina en su estado protonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

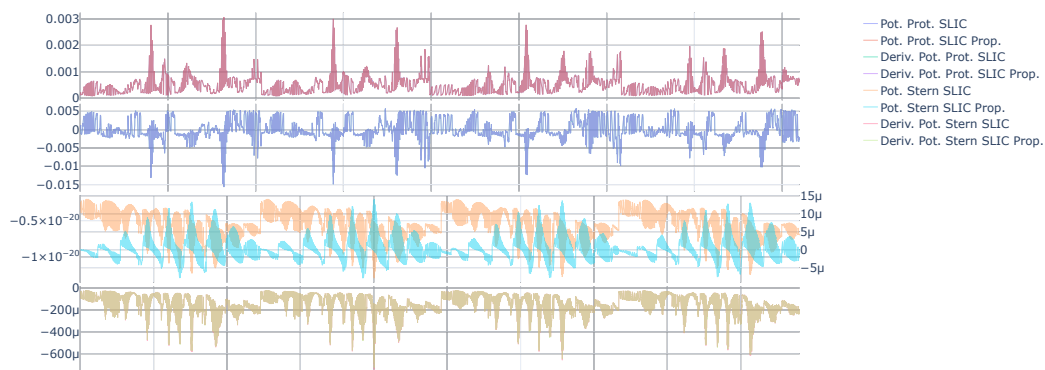


Figura C.10: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido histidina en su estado deprotonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

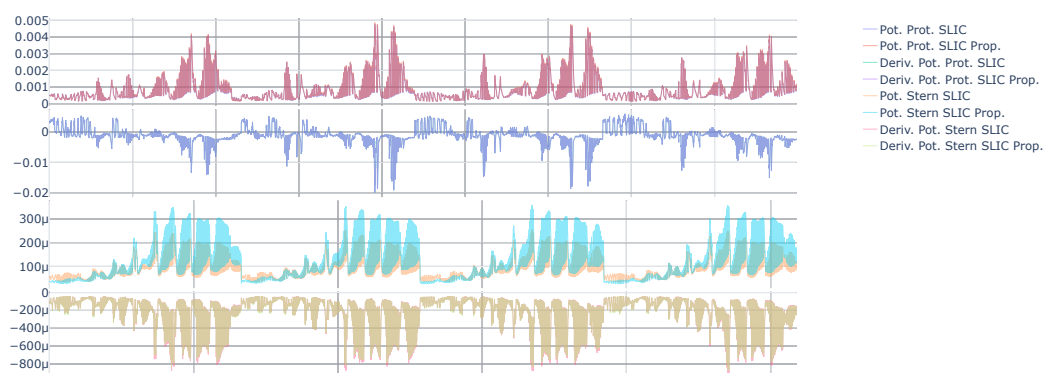


Figura C.11: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido lisina en su estado protonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

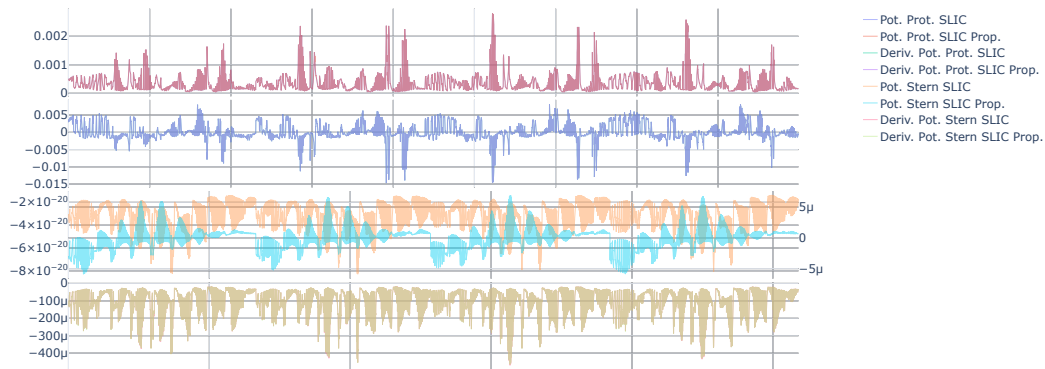


Figura C.12: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido lisina en su estado deprotonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

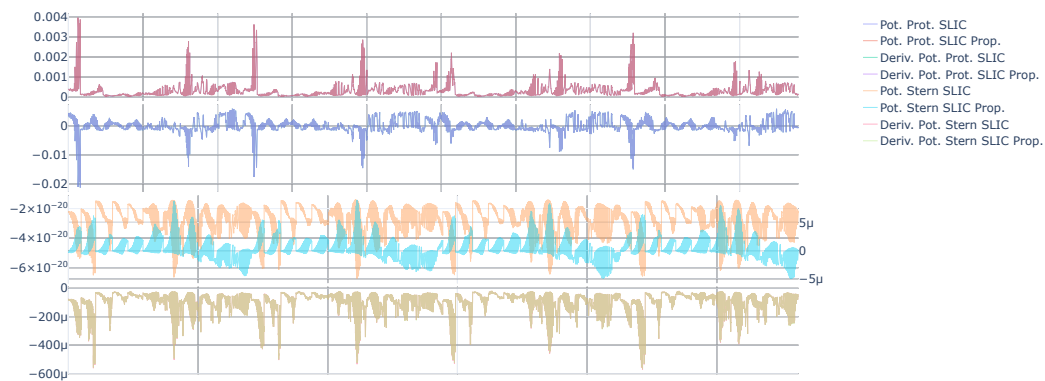


Figura C.13: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido tirosina en su estado protonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

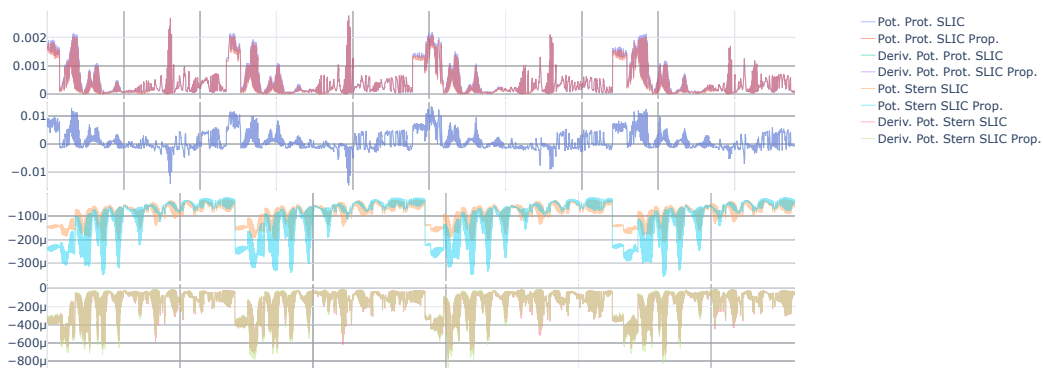


Figura C.14: Comparación entre los resultados del potencial y su derivada en la interfaz dieléctrica y *stern layer* del aminoácido tirosina en su estado deprotonado, obtenidos usando el modelo SLIC y el modelo SLIC con la mejora propuesta.

Referencias

- [1] Altman, M. D., Bardhan, J. P., White, J. K., & Tidor, B. (2009). Accurate solution of multi-region continuum biomolecule electrostatic problems using the linearized poisson–boltzmann equation with curved boundary elements. *Journal of Computational Chemistry*, 30, 132–153.
- [2] Bardhan, J. P. & Knepley, M. G. (2014). Modeling charge-sign asymmetric solvation free energies with nonlinear boundary conditions. *The Journal of Chemical Physics*, 141.
- [3] Betcke, T. & Scroggs, M. W. (2021). Bempp-cl: A fast python based just-in-time compiling boundary element library. *Journal of Open Source Software*, 6(59), 2879.
- [4] Cooper, C. (2015). *Biomolecular Electrostatics With Continuum Models: A Boundary Integral Implementation and Applications to Biosensors*. PhD thesis, Boston University, Boston, MA 02215, Estados Unidos.
- [5] Cooper, C. D., Clementi, N. C., Forsyth, G., & Barba, L. A. (2016). PyGBe: Python, GPUs and boundary elements for biomolecular electrostatics. *JOSS*, 1(4), 43.
- [6] Costabel, M. (1987). Principles of boundary element methods. *Computer Physics Reports*, 6, 243–274.
- [7] Decherchi, S., Spitaleri, A., Stone, J., & Rocchia, W. (2018). NanoShaper–VMD interface: computing and visualizing surfaces, pockets and channels in molecular systems. *Bioinformatics*, 35(7), 1241–1243.
- [8] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard,

- K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., & Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357–362.
- [9] Hunter, P. & Pullan, A. (2003). Fem/bem notes. https://auckland.figshare.com/articles/journal_contribution/FEM_BEM_Notes/5440000/1.
- [10] Inc., P. T. (2015). Collaborative data science.
- [11] Martinez, J. A. (2019). Chemistry 104 lecture notes.
- [12] Masunov, A. & Lazaridis, T. (2003). Potentials of mean force between ionizable amino acid side chains in water. *Journal of the American Chemical Society*, 125, 1722–1730.
- [13] Ohshima, H. (2013). *Electrical Double Layers*, (pp. 342–361). Springer Berlin Heidelberg: Berlin, Heidelberg.
- [14] Salih, A. (2016). Weighted residual methods. <https://www.iist.ac.in/sites/default/files/people/IN08026/WRM.pdf>.
- [15] Sanner, M. F., Olson, A. J., & Spehner, J.-C. (1995). Fast and robust computation of molecular surfaces. (pp. 406–407).
- [16] Search, S., Cooper, C., Godoy, M., & Styles, K. (2021). Pbj. <https://github.com/bem4solvation/pbj>.
- [17] Sharp, K. A. & Honig, B. (1990). Electrostatic interactions in macromolecules: Theory and applications. *Annual Reviews of Biophysics*, 19, 301–332.
- [18] Skyner, R. E., McDonagh, J., Groom, C. R., van Mourik, T., & Mitchell, J. B. O. (2015). A review of methods for the calculation of solution free energies and the modelling of systems in a solution. *Physical Chemistry Chemical Physics*, 17, 6137–6670.
- [19] Stern, O. (1924). Zur theorie der elektrolytischen doppelschicht. *Zeitschrift für Elektrochemie und Angewandte Physikalische Chemie*, 30, 508–516.
- [20] stuartarchibald (2018). Support axis kwarg in np.linalg.norm. <https://github.com/numba/numba/issues/2558>.

- [21] Styles, K. (2021). Repositorio de tesis. <https://github.com/kstyleesc/Tesis/>.
- [22] Tabrizi, A. M., Goossens, S., Rahimi, A. M., Cooper, C. D., Knepley, M. G., & Bardhan, J. P. (2017). Extending the solvation-layer interface condition continuum electrostatic model to a linearized poisson–boltzmann solvent. *Journal of Chemical Theory and Computation*, 13, 2897–2914.
- [23] Tiobe (2021). Tiobe index. <https://www.tiobe.com/tiobe-index/>.
- [24] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., & SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17, 261–272.
- [25] Yoshida, N. (2017). Role of solvation in drug design as revealed by the statistical mechanics integral equation theory of liquids. *Journal of Chemical Information and Modeling*, 57(11), 2646–2656. PMID: 28991467.