We can find a good way to construct $G^0_{ab}(i\omega_n)$ later, perhaps it can be done simply in momentum space when we leave out the perturbation but for the Hubbard dimer I'll just use the exact fully interacting Green function and set the interactions to zero, i.e. $U \to 0$. Here a and b represent spacial indices, the spin doesn't need to be explicitly taken into account for the Hubbard dimer at half filling.

With a given $G^0_{ab}(i\omega_n)$ we can find the polarization as:

$$\Pi_{abcd}(\tau) = -G_{da}(\tau)G^T_{bc}(-\tau) \tag{1}$$

Where $G_{ab}(\tau) = \mathcal{F}_{i\omega_n \to \tau}\{G_{ab}(i\omega_n)\}$ This is implemented as follows:

```
def polarization(self, G):

    G = make_gf_from_fourier(G)
    P = make_gf_from_fourier(BlockGf(mesh = self.bmesh, gf_struct = self.gf_struct,
    target_rank = 4))

    for name, g in P:
        indices = [0, 1]
        for a, b, c, d in product(indices, indices, indices, indices):
            P[name].data[:, a, b, c, d] = -G[name].data[:, d, a] * G[name].
   transpose().data[::-1, b, c]

    return P
```

Here `self.bmesh` is the bosonic Matsubara frequency mesh and later on I'll also use the fermionic mesh `self.fmesh`. The structure of the Green function and all the other terms are defined by `self.gf_struct = [('up', 2), ('dn', 2)]`.

The Coulomb potential V_{abcd} is a given and together with the polarization we can find the full screened potential:

$$W_{abcd}(i\omega_n) = V_{abcd} + \sum_{efgh} V_{abef}\Pi_{fegh}(i\omega_n)W_{hgcd}(i\omega_n) \tag{2}$$

For which we have to Fourier transform the polarization: $\Pi_{abcd}(i\omega_n) = \mathcal{F}_{\tau \to i\omega_n}\{\Pi_{abcd}(\tau)\}$. This is implemented as follows:

```
def screenedPotential(self, P, v):
    P = make_gf_from_fourier(P)

    W = BlockGf(mesh = self.bmesh, gf_struct = self.gf_struct, target_rank = 4)

    reshaped = BlockGf(mesh = self.bmesh, gf_struct = [('up', 4), ('dn', 4)],
   target_rank = 2)

    v_reshaped = reshaped.copy()
    P_reshaped = reshaped.copy()
    idm = reshaped.copy()

    size = np.shape(W['up'].data)[0]

    for name, g, in W:
        v_reshaped[name].data[:] = v[name].data.reshape(size, 4, 4)
        P_reshaped[name].data[:] = P[name].data.reshape(size, 4, 4)
        idm[name].data[:] = np.eye(4)

    W_reshaped = (idm - 2 * v_reshaped * P_reshaped).inverse() * v_reshaped

    for name, g in W:
        W[name].data[:] = W_reshaped[name].data.reshape(size, 2, 2, 2, 2)

    return W
```

What the code does is transform the Green function objects from $(2,2,2,2)$ matrices to $(4,4)$ matrices and then inverts the Dyson equation: $W = (1 - 2v\Pi)^{-1}v$. The factor 2 for the polarization is also still there, regardless of the spin flag. For the computation of the self energy the full screened potential is split up into a dynamical part dependent of $i\omega_n$ and a static part:

$$W_{abcd}(i\omega_n) = \tilde{W}_{abcd}(i\omega_n) + V_{abcd} \tag{3}$$

The self energy is then obtained from just the dynamical part, the static contribution is added later. For the self energy we have:

$$\tilde{\Sigma}_{ab}(\tau) = -\sum_{cd} \tilde{W}_{abcd}(\tau) G_{cd}(\tau) \tag{4}$$

For which just the dynamical part of the screened interaction was Fourier transformed: $\tilde{W}_{abcd}(\tau) = \mathcal{F}_{i\omega_n \to \tau}\{\tilde{W}_{abcd}(i\omega_n)\}$. This was implemented as follows:

```
def selfEnergy(self, G, W):

    G = make_gf_from_fourier(G)
    W_dynamic = make_gf_from_fourier(W - self.v)
    sigma = make_gf_from_fourier(BlockGf(mesh = self.fmesh, gf_struct = self.gf_struct, target_rank = 2))
    cpy = sigma.copy()

    for name, g in sigma:
        indices = [0, 1]
        for a, b in product(indices, indices):
            acc = cpy.copy()
            indices = [0, 1]
            for c, d in product(indices, indices):
                acc[name].data[:, a, b] += -W_dynamic[name].data[:, a, c, b, d] * G[name].data[:, c, d]

            sigma[name].data[:, a, b] = acc[name].data[:, a, b]

    return sigma
```

Note that the indices for the dynamic part of the screened potential are different than in the equation. This gives the correct result, perhaps due to the reshaping of the data? The contribution of the neglected static part of the full screened potential is computed as follows:

```
def density(self, G):
    rho = BlockGf(mesh = self.bmesh, gf_struct = self.gf_struct, target_rank = 2)
    for name, g in G:
        rho[name].data[:] = G[name].density()
    return rho

def static(self, G, V):
    rho = self.density(G)

    static = BlockGf(mesh = self.fmesh, gf_struct = self.gf_struct, target_rank = 2)

    for name, g in static:
        indices = [0, 1]
        for a, b in product(indices, indices):
            static[name].data[:-1, a, b] = self.v[name].data[:, a, b, a, b] * rho[name].data[:, a, b]
            static[name].data[-1, a, b] = self.v[name].data[-1, a, b, a, b] * rho[name].data[-1, a, b]  # Add last element as bmesh and fmesh are not the same size

    return static
```

This implementation comes from:

$$\Sigma_{ab}(\tau) - \tilde{\Sigma}_{ab}(\tau) = -\mathcal{F}_{i\omega_n\to\tau}\{V_{abab}\}\mathcal{F}_{i\omega_n\to\tau}\{G_{ab}(i\omega_n)\} \tag{5}$$

$$\begin{aligned}\mathcal{F}_{\tau\to i\omega_n}\{\Sigma_{ab}(\tau) - \tilde{\Sigma}_{ab}(\tau)\} &= -\mathcal{F}_{\tau\to i\omega_n}\{\mathcal{F}_{i\omega_n\to\tau}\{V_{abab}\}\mathcal{F}_{i\omega_n\to\tau}\{G_{ab}(i\omega_n)\}\} \\ &= -(V_{abab} * G_{ab})(i\omega_n) \\ &= -V_{abab}\frac{1}{\beta}\sum_n G_{ab}(i\omega_n) \\ &= V_{abab}\rho_{ab} \\ &= \Sigma_{ab}^{\text{static}}\end{aligned} \tag{6}$$

We have two more similar terms, the Hartree term and the Fock term:

$$\Sigma_{ab}^{\text{Hartree}} = \sum_{cd} V_{abcd}\rho_{cd} \tag{7}$$

$$\Sigma_{ab}^{\text{Fock}} = -\sum_{cd} V_{acdb}\rho_{dc} \tag{8}$$

These terms are implemented as follows:

```
def hartree(self, v, G):
    rho = self.density(G)

    hartree = BlockGf(mesh = self.fmesh, gf_struct = self.gf_struct, target_rank = 2)
    cpy = hartree.copy()

    for name, g in hartree:
        indices = [0, 1]
        for a, b in product(indices, indices):
            acc = cpy.copy()
            indices = [0, 1]
            for c, d in product(indices, indices):
                acc[name].data[:-1, a, b] += v[name].data[:, a, b, c, d] * rho[name].data[:, c, d]
                acc[name].data[-1, a, b] += v[name].data[-1, a, b, c, d] * rho[name].data[-1, c, d]

            hartree[name].data[:, a, b] = acc[name].data[:, a, b]

    return hartree

def fock(self, v, G):
    rho = self.density(G)

    fock = BlockGf(mesh = self.fmesh, gf_struct = self.gf_struct, target_rank = 2)
    cpy = fock.copy()

    for name, g in fock:
        indices = [0, 1]
        for a, b in product(indices, indices):
            acc = cpy.copy()
            indices = [0, 1]
            for c, d in product(indices, indices):
                acc[name].data[:-1, a, b] += v[name].data[:, a, c, d, b] * rho[name].data[:, d, c]
                acc[name].data[-1, a, b] += v[name].data[-1, a, c, d, b] * rho[name].data[-1, d, c]

            fock[name].data[:, a, b] = -acc[name].data[:, a, b]

    return fock
```

The full self energy is then given by:

$$\Sigma_{ab}(i\omega_n) = \mathcal{F}_{\tau \to i\omega_n}\{\tilde{\Sigma}_{ab}(\tau)\} + \Sigma_{ab}^{\text{static}} + \Sigma_{ab}^{\text{Hartree}} + \Sigma_{ab}^{\text{Fock}} \tag{9}$$

Flags are implemented on whether or not to include each term. If only the static term is added, the implementation completely agrees with the exact solutions given in the paper. The TRIQS tprf implementation for GW ignores the static term so when it's neglected my, implementation agrees with Hugo's implementation for all the possible combinations of Hartree and Fock flags, as well as the spinless flag which changes the Coulomb potential. Though in the case of not spinless, the Hartree term has to be added analytically as `+ self.U / 2`.

Lastly the interacting GW Green function is obtained by:

$$G_{ab}^{\text{GW}}(i\omega_n) = ((G_{ab}^{0})^{-1}(i\omega_n) - \Sigma_{ab}(i\omega_n))^{-1} \tag{10}$$

This is simply implemented as:

```
def greenFunction(self, G0, sigma):
    return (G0.inverse() - sigma).inverse()
```

Though perhaps I should implement a proper Dyson equation solver that also corrects the chemical potential to the chosen occupation. Though for 1-shot GW this doesn't seem to matter.