Preprint typeset in JHEP style - HYPER VERSION

CoulombHiggs.m v6.3

Boris Pioline

Abstract: Basic documentation for the Mathematica package `CoulombHiggs.m` available from `http://www.lpthe.jussieu.fr/~pioline/computing.html` and from `https://github.com/bpioline/CoulombHiggs`

Contents

---

1. Summary

The MATHEMATICA package `CoulombHiggs.m` provides a suite of routines compute (specializations of) the Hodge polynomial

$$\Omega(\gamma,\zeta,y,t) := \sum_{p,q=0}^{d} (-y)^{p+q-d} t^{p-q}\, h_{p,q}(\mathcal{M}_Q) \tag{1.1}$$

of the moduli space $\mathcal{M}_Q$ of semi-stable representations of a quiver Q with antisymmetric adjacency matrix α_{ij} (such that α_{ij} counts the net number of arrows $i \to j$, with arrows $j \to i$ counted negatively), dimension vector $\gamma = (N_1, \ldots, N_K)$ and stability parameters $(\zeta_1, \ldots, \zeta_K)$ such that $\sum_i N_i\zeta_i = 0$. When $\mathcal{M}_Q$ is a smooth projective variety of complex dimension d, (1.1) is a Laurent polynomial with integer coefficients, invariant under separate inversions $y \to 1/y, t \to 1/t$. When γ is not primitive, it is useful to introduce the *rational invariant*

$$\bar\Omega(\gamma,\zeta,y,t) = \sum_{d|\gamma} \frac{y-1/y}{d(y^d-y^{-d})} \Omega(\gamma/d,\zeta,y^d,t^d) \tag{1.2}$$

which satisfies simpler wall-crossing identities. Finally, $\bar\Omega(\gamma,\zeta,y,t)$ may be expressed in terms of the *stack invariants* $G_{\rm Higgs}(\gamma,\zeta,y,t)$ defined by [**?**, (4.1)]

$$\bar\Omega(\gamma,\zeta,y,t) = \sum_{\substack{\{\vec M^{(i)}\} \\ \sum_{i=1}^\ell \vec M^{(i)} = \vec N \\ \vec M^{(i)} \parallel \vec N \text{ for } i=1,\ldots,\ell}} \frac{1}{\ell\,(y-1/y)^{\ell-1}} \prod_{i=1}^{\ell} G_{\rm Higgs}(\{M_1^{(i)},\cdots M_K^{(i)}\};\{\zeta_1,\cdots\zeta_K\};y,t)\,. \tag{1.3}$$

where $\vec N$ is the dimension vector corresponding to γ. The stack invariants satisfy yet simpler wall-crossing identities, but their physical meaning is less transparent. Note that they differ from the *stacky invariants* by a factor $(1/y - y)$.

The package implements the following main formulae:

- the *Higgs branch formula* is based on Reineke's solution to the Harder-Narasimhan recursion [**?**] for quivers with primitive dimension vector and no closed loops. The formula apparently also applies to quivers with oriented loops but without superpotential and for non-primitive dimension vector, provided the Hodge numbers $h_{p,q}(\mathcal{M}_Q)$ are defined using intersection homology;

- the *Coulomb branch* formula [**?**, **?**, **?**, **?**] is based on a physical picture of BPS states as bound states of single-centered black holes; it applies to any quivers with or without oriented loops and expresses the index in terms of 'single-centered invariants' $\Omega_{\rm S}(\gamma_i, t)$, which are independent of stability conditions, and conjecturally depend only on t;

- the *Joyce formula* [**?**] expresses the rational invariants $\bar\Omega(\gamma, \zeta_1, y, t)$ in terms of the invariants $\bar\Omega(\gamma_i, \zeta_2, y, t)$, and similarly the stack invariants $G_{\rm Higgs}(\gamma, \zeta_1, y, t)$ in terms of $G_{\rm Higgs}(\gamma_i, \zeta_2, y, t)$, where $\gamma = \sum_i \gamma_i$.

- the *JK residue formulae* [**?**] (see also [**?**, **?**]) is based on localization, and evaluates the χ-index $\Omega(\gamma, \zeta, y, y)$ as a suitable sum of residues of a certain rational or trigonometric function; when γ is not primitive, the residue formula computes the value at $y = t$ of the rational invariant (1.2).

- the *flow tree formula* [**?**] is based on the attractor flow tree conjecture, and expresses the total index in terms of attractor indices $\Omega_*(\gamma_i, y, t)$ which are also independent of stability conditions, but may depend on both y and t; the attractor index is defined by $\Omega_*(\gamma, y, t) = \Omega(\gamma, \zeta_*, y, t)$ where ζ_* is a generic perturbation of the attractor point $\zeta_a = -\kappa_{ab} N_b$;

- the *attractor tree formula* [**?**] (based on ideas in [**?**]) is (believed to be) equivalent to the *flow tree formula*, but it does not rely on perturbing the DSZ matrix γ_{ij} in intermediate steps.

- the *Quiver Yangian algorithm* [**?**], based on ideas in [**?**], computes unrefined NCDT invariants for brane tilings.

The package file `CoulombHiggs.m` and various example files can be obtained from the author's webpage,

`http://www.lpthe.jussieu.fr/~pioline/computing.html`

1.1 Basic usage

Assuming that the file `CoulombHiggs.m` is present in the user's MATHEMATICA Application directory, the package is loaded by entering

```
In[1]:= <<CoulombHiggs`
Out[1]= CoulombHiggs 6.0 - A package for evaluating quiver
        invariants.
```

If the file `CoulombHiggs.m` has not yet been copied in the user's MATHEMATICA Application directory but is in the same directory as the notebook, evaluate instead

```
In[1]:= SetDirectory[NotebookDirectory[]]; <<CoulombHiggs`
Out[1]= CoulombHiggs 6.0- A package for evaluating quiver
        invariants.
```

1.1.1 Coulomb branch formula

The basic usage of `CoulombBranchFormula` is illustrated below: [1]

```
In[1]:= Simplify[CoulombBranchFormula[4{{0, 1, -1},{-1, 0, 1}, {1,
        -1, 0}}, {1/2, 1/6, -2/3}, {1, 1, 1}]]
Out[1]= 2 + 1/y^2 + y^2 + OmS({1, 1, 1}, y, t)
```

The first argument corresponds to the matrix of DSZ products α_{ij} (an antisymmetric matrix of integers), the second to the FI parameters ζ_i (a vector of rational numbers), the third to the dimension vector N_i (a vector of integers). Here, the Dolbeault polynomial of the moduli space of a three-node Abelian cyclic quiver $C_{4,4,4}$ with 4 arrows between each consecutive node is expressed in terms of the single-centered index $\Omega_{\rm S}(\gamma_1+\gamma_2+\gamma_3, y, t)$.

1.1.2 Higgs branch formula

The arguments of `HiggsBranchFormula` are the same as for `CoulombBranchFormula`:

```
In[1]:= Simplify[HiggsBranchFormula[{{0, 3},{-3, 0}}, {1/2,-1/2},
        {2, 2}]]
Out[1]= -((y^2+1)(y^8+y^4+1))/y^5
```

The above command computes the invariant $\Omega(\gamma,\zeta,t,y)$ for the Kronecker quiver $K_3(2,2)$ with 3 arrows, dimension vector $\gamma=(2,2)$ and stability parameters $\zeta=(1/2,-1/2)$. This is computing by expressing $\Omega(\gamma)$ in terms of the rational invariant $\bar\Omega(\gamma)$, then $\bar\Omega(\gamma)$ in terms of the stack invariant, which is then evaluated using Reineke's formula. Note that the result is a symmetric Laurent polynomial with integer coefficients, despite the fact that γ is not primitive.

[1] Note the following changes in v2.0: the fugacity y is no longer a parameter of `CoulombBranchFormula` and `QuiverBranchFormula`, and the former computes the Dolbeault polynomial in terms of $\Omega_{\rm S}(\alpha_i,t)$, rather than expressing the Poincaré polynomial in terms of $\Omega_{\rm S}(\alpha_i)$. Starting in v2.1, if $\sum_i N_i\zeta_i$ does not vanish, rather than issuing an error message, a uniform translation is applied internally to the ζ_i's. Other changes are highlighted by margin notes below.

1.1.3 Jeffrey-Kirwan residue formula

Beware: the routine JKIndex appears to have become corrupted due to changes in v5. Please use JKIndexSplit or revert to v4.

The third main routine JKIndex implements the Jeffrey-Kirwan residue formula. Its first argument ChargeMatrix is an extended charge matrix, where the rows encode the charges of the chiral multiplets under $U(1)^{\sum N_i}$, along with the R-charges and the multiplicity. The second argument Nvec is the dimension vector N_i, and the last argument Etavec is the stability condition η, a vector of length $\sum_i N_i$. These arguments can be generated by using JKInitialize, which takes as arguments the antisymmetric DSZ matrix α_{ij}, symmetric R-matrix r_{ij}, stability vector ζ_i and dimension vector N_i, and sets various global variables including JKChargeMatrix and JKEta. For example, the index of a two-node quiver with three arrows, dimension vector $N_i = (1, 2)$, stability vector $\eta = (1, -2/5, -3/5)$ is obtained from

```
In[1]:= JKInitialize[{{0, 3}, {-3, 0}}, {{0, 0}, {0, 0}}, {1, 2},
        {1, -1/2}]; JKIndex[ JKChargeMatrix, {1, 2},JKEtavec]
Out[1]= {y^2 + 1 + 1/y^2}
```

The result produces a list of contributions from each contributing stable flag, which consists of a single flag in this case. The routine first determines the stable flags which give a non-zero contribution to the Euler number $\Omega(\gamma, \zeta, 1, 1)$, and then uses the same flags to compute the χ-genus $\Omega(\gamma, \zeta, y, y)$, which is more time-consuming. The result for the Euler number can be accessed from the global variable JKEuler. Note that flavor fugacities can be switched on by using the routine FlavoredRMatrix to construct the matrix of R-charges (which was chosen to vanish in the previous example).

A variant JKIndexSplit of the same routine is provided, which computes the same index by first splitting the vector multiplet determinants as a sum over (equivalence classes of) permutations, using Cauchy's determinant formula, and computes the Jeffrey-Kirwan residue of each term separately (the two procedures are of course identical for Abelian quivers). This simplifies both the enumeration of stable flags (as the number of singular hyperplanes due to vector multiplets is reduced from $N_i(N_i-1)$ to $\mathcal{O}(N_i)$) and the computation of the residues. Moreover, for quivers with loops this splitting appears to match the sum over decompositions $\gamma = \sum \alpha_i$ in the Coulomb branch formula [**?**]. For example, applying this splitting procedure on the second node for the previous example gives

```
In[1]:= JKInitialize[{{0, 3}, {-3, 0}}, {{0, 0}, {0, 0}}, {1, 2},
        {1, -1/2}]; JKIndexSplit[ JKChargeMatrix, {1, 2},JKEta,{2}];
Out[1]= {{(-y^8-y^4-1)/(2y^4)}, {(y^4+y^2+1)^2/(2y^4)}}
```

which sums up to the same result as before $y^2 + 1 + 1/y^2$. Each entry in the result corresponds to the contribution of a given multi-partition of the dimension vector, the list of which is stored (along with respective multiplicities) in the global variable JKListAllPerms. Again, the intermediate results for the Euler number can be retrieved from the global variable JKEuler.

For the evaluation of the residue, we note that it is often more efficient to use a rational representation of the integrand (using exponentiated Cartan variables) than a trigonometric representation. The former is used when $QuiverTrig is set to False. If this variable is set to a value different from True or False, then the routine does not attempt to compute the full χ-genus and returns the Euler number instead.

Finally, by setting $QuiverTrig to True and $QuiverMaxPower > 0, the routine will attempt to compute the elliptic genus up to order $q^{\$\text{QuiverMaxPower}}$ using the residue prescription in [?]. Note however that it will only include the same hyperplanes which contribute to the Euler number, and not hyperplanes which decouple as $\tau \to i\infty$.

1.1.4 Flow tree formula

The arguments for FlowTreeFormula are the same as for CoulombBranchFormula:

```
In[1]:= Simplify[FlowTreeFormula[4{{0, 1, -1},{-1, 0, 1}, {1, -1,
        0}}, {1/2, 1/6, -2/3}, {1, 1, 1}]]
Out[1]= OmAtt({1, 1, 1}, y)
```

In this case, the sum over tree vanishes. Comparing with the result from the Coulomb branch formula above, one concludes that the attractor index $\Omega_*(\gamma) = \Omega_S(\gamma) + y^2 + 2 + 1/y^2$, corresponding to the sum of the contributions of single centered and scaling solutions.

1.1.5 Attractor tree formula

As of version 5.2, the package contains an implementation of a formula implicit in [?], which allows to express the index $\Omega(\gamma, \zeta, y, t)$ in terms of attractor indices $\Omega_\star(\gamma, \zeta, y, t)$, without perturbing the DSZ matrix at any stage. The syntax is identical to the flow tree formula,

```
In[1]:= Simplify[AttractorTreeFormula[4{{0, 1, -1},{-1, 0, 1}, {1,
        -1, 0}}, {1/2, 1/6, -2/3}, {1, 1, 1}]]
Out[1]= OmAtt({1, 1, 1}, y)
```

1.1.6 Quiver Yangian algorithm

The algorithm relies on a matrix hMat whose entry (i, j) lists the heights h_a of arrows $a : i \to j$. These heights are linear combinations of three parameters

h_1, h_2, h_3 such that, for all vertices $i \in Q_0$

$$\forall i : \quad \sum_{a:i\to j} h(a) - \sum_{a:j\to i} h(a) = 0 \tag{1.4}$$

and for all monomials $F \in Q_2$ in the superpotential $W = W_+ - W_-$,

$$\forall F : \quad \sum_{a\in F} h'(a) = h_3 \tag{1.5}$$

These heights, as well as the bipartite potential $W = W_+ - W_-$, are pre-computed for a number of common brane tilings, listed by

```
In[1]:= Simplify[ListKnownBraneTilings]
```

Out[1]:=
$1 : C^3$
$2 : \text{Conifold}$
$3 : C^2 \times C/2$
$4 : C^2 \times C/3$
$5 : C^3/2 \times 2$
$6 : SPP$
$7 : L131$
$8 : P2 = C^3/(1,1,1)$
$9 : F0.1 = P1 \times P1$
$\ldots$

For computing unrefined NCDT invariants for the conifold up to order x^5, you may use

```
In[1]:= {Tiling, Fan, hMat, Wp, Wm, v1, v2} =
        BraneTilingsData[[2]]; NCDTSeriesFromCrystal[hMat, {1, 0},
        5]
```

Out[1]:= $1 + x[1] + 2y^3x[1]x[2] + 4y^5x[1]^2x[2] + 2y^5x[1]^3x[2] + y^4x[1]x[2]^2 + 8y^{10}x[1]^2x[2]^2 + 14y^{14}x[1]^3x[2]^2 + 4y^{13}x[1]^2x[2]^3$

and set $y = -1$ in the final result. Setting $y = 1$ instead gives the number of molten crystals at each order.

1.2 Online documentation

The package contains many more routines, documented below, which can be used independently. Basic inline documentation can be obtained by typing e.g.

```
In[1]:= ?CoulombBranchFormula
Out[1]= CoulombBranchFormula[Mat_,Cvec_,Nvec_] expresses the
        Dolbeault polynomial of a quiver with dimension vector gam
        in terms of the single center degeneracies OmS[alpha_i,t]
        using the Coulomb branch formula, computing all CoulombH
        factors recursively using the minimal modification
        hypothesis.  Also provides list of CoulombH factors if
        $QuiverDisplayCoulombH is set to True
```

1.3 History

The first version of this package was released together with the preprint [**?**] where a general algorithm for computing the index of the quantum mechanics of multi-centered BPS black holes (the Coulomb index) was outlined. The version 2.0, released along with the preprint [**?**], allowed to compute the Dolbeault-Laurent polynomial, relax assumptions on single-centered indices for basis vectors, study the effect of generalized mutations, and more. Version 2.1, released along with the review [**?**], was optimized to speed up the evaluation of Coulomb indices. Version 3.0 introduced an early version of the Jeffrey-Kirwan residue formula. Version 4.0, released along with [**?**], introduced the flow tree formula. Version 5.0, released along with [**?**], introduced more robust implementation of the Jeffrey-Kirwan residue formula. Version 6.0, released along with [**?**], introduces the attractor tree formula and the Quiver Yangian algorithm, along with many routines for dealing with brane tilings. Version 6.2, released along with [**?**], includes routines for computing the index of Abelian quivers by solving deformed Denef equations.

1.4 Usage in literature

The following papers by other authors have acknowledged using this package for part of their computations (non exhaustive list): [**?**, **?**, **?**, **?**, **?**, **?**].

2. Variables and Symbols

2.1 Symbols

- `y`: fugacity conjugate to the sum of Dolbeault degrees $p + q$ (i.e. angular momentum);
- `z`: chemical potential, $y = e^{i\pi z}$;
- `t`: fugacity conjugate to the difference of Dolbeault degrees $p - q$;
- `q`: modular parameter in the elliptic genus, $q = e^{2\pi i\tau}$;
- `tau`: elliptic modulus, $q = e^{2i\pi\tau}$, for elliptic genus computations;

- `Om[`*charge vector_*`,`*y_*`]`: denotes the refined index $\Omega(\gamma, y)$;
- `Omb[`*charge vector_*`,`*y_*`]`: denotes the rational refined index $\bar{\Omega}(\gamma, y)$;
- `OmS[`*charge vector_*`,`*y_*`,`*t_* `]`: denotes the single-centered index $\Omega_{\rm S}(\gamma, y, t)$.
- `OmS[`*charge vector_*`,`*y_*`]`: denotes $\Omega_{\rm S}(\gamma, y) \equiv \Omega_{\rm S}(\gamma, y, t = 1)$.
- `OmS[`*charge vector_*`]`: denotes $\Omega_{\rm S}(\gamma, y)$, under the assumption that it is independent of y (which is conjectured to be the case for generic superpotential)
- `OmAtt[`*gam_*`,`*y_*`]`: denotes the attractor index with charge `gam`
- `OmAttb[`*gam_*`,`*y_*`]`: denotes the rational attractor index with charge `gam`
- `OmT[`*charge vector_*`,`*y_*`]`: denotes the (unevaluated) function $\Omega_{\rm tot}(\gamma, y)$;
- `Coulombg[`*list of charge vectors_*`,`*y_*`]`:: denotes the (unevaluated) Coulomb index $g_{\rm Coulomb}(\{\alpha_i\}, \{c_i\}, y)$, leaving the FI parameters unspecified;
- `HiggsG[`*charge vector_*`,`*y_*`]`: denotes the (unevaluated) stack invariant $G_{\rm Higgs}(\gamma, y)$;
- `CoulombH[`*list of charge vectors_*`,`*multiplicity vector_*`,`*y_*`]`: denotes the (unevaluated) factor $H(\{\alpha_i\}, \{n_i\}, y)$ appearing in the formula for $\Omega_{\rm tot}(\sum n_i\alpha_i, y)$ in terms of $\Omega_{\rm S}(\alpha_i, y)$.
- `QFact[`*n_*`,`*y_*`]`: represents the (non-evaluated) q-deformed factorial $[n, y]!$
- `u[i,s]`: s-th Cartan variable for the i-th gauge group when the trigonometric representation is used, exponentiated version $e^{2\pi i u_{i,s}}$ of the same when a rational representation is used.
- `ut[i,s]`: Cartan variables in rotated basis adapted to a singularity, so that the flag is $\tilde{u}_{1,1} = \cdots = u_{K,N_K} = 0$ in trigonometric representation, or $\tilde{u}_{1,1} = \cdots = u_{K,N_K} = 1$ in rational representation.
- `th[`*i_*`]`: Chemical potential for flavor symmetry, used in `FlavoredRMatrix`
- `Theta[z_]`: Jacobi Theta series $\theta_1(z, \tau) = -\mathrm{i}q^{1/8}(e^{\mathrm{i}\pi z} - e^{-\mathrm{i}\pi z}) \prod_{k\geq 1}(1-q^k)(1-e^{2\pi\mathrm{i}z}q^k)(1-e^{-2\pi\mathrm{i}z}q^k)$
- `Eta`: Dedekind Eta series $\eta(\tau) = q^{1/24}\prod_{k\geq 1}(1-q^k)$
- `h1,h2,h3`: Parameters for the heights $h(a)$ used by the Quiver Yangian algorithm

2.2 Global variables

- JKListuAll: Flat list of all Cartan variables $u[i,s]$, $i = 1 \ldots K$, $s = 1, \ldots N_i$
- JKListuDisplay: Same as JKListuAll, only used by FlagToHyperplane for display
- JKListu: Flat list of unfrozen Cartan variables $u[i,s]$
- JKListut: Flat list of unfrozen rotated Cartan variables $ut[i,s]$
- JKFrozenCartan: List of pairs $\{i,s\}$ which specifies the list of Cartan variables $u_{i,s}$ which should be frozen to 0 (or 1 in rational representation), rather than integrated over.
- JKFrozenMask: Vector of booleans indicating non-frozen entries in JKListuAll
- JKFrozenRuleEuler: Rule for replacing the frozen $u[i,s]$ by 0
- JKFrozenRuleRat: Rule for replacing the frozen $u[i,s]$ by 1
- JKEuler: List of contributions of all stable flags to the Euler number, as computed by JKIndex or JKIndexSplit
- JKChiGenus: List of contributions of all stable flags to the chi-genus, as computed by JKIndex or JKIndexSplit
- JKListAllSings: List of singularities, as computed by JKIndex or JKIndexSplit
- JKListAllStableFlags: List of all stable flags, as computed by JKIndex or JKIndexSplit
- JKRelevantStableFlags: List of stable flags contributing to the Euler number, as computed by JKIndex or JKIndexSplit
- JKListAllPerms: List of multi-partitions, as computed by JKIndexSplit
- JKVertexCoordinates: Coordinates of vertices for DisplayFlagTree, set by JKInitialize
- JKVertexLabels: Labels of vertices for DisplayFlagTree, set by JKInitialize
- BraneTilingsData: List of {Name, Fan, hMat, Wp, Wm, v1,v2 } for known brane tilings, where Fan is the toric fan (see PlotToricFan), hMat the height matrix (see NCDTSeriesFromCrystal), $W =$ Wp $-$ Wm the superpotential (see ListPerfectMatchings) and v1, v2 the two basis vectors used by PlotTiling to display the tiling.

2.3 Environment variables

- $QuiverPerturb1: Sets the size of the perturbation $\epsilon_1 = 1/$$QuiverPerturb of the DSZ products, set to 1000 by default.

- $QuiverPerturb2: Sets the size of the perturbation $\epsilon_2 = 1/$$DSZPerturb of the DSZ products, set to 10^{10} by default.

- $QuiverNoLoop: If set to True, the quiver will be assumed to have no oriented loop, hence all H factors and all $\Omega_S(\alpha)$ will be set to zero (unless α is a basis vector). Set to False by default.

- $QuiverTestLoop: If set to True, all H factors and $\Omega_S(\alpha)$ corresponding to subquivers without loops will be set to zero (unless α is a basis vector). Set to True by default. Determining whether a subquiver has loops is time-consuming, so for large quivers it may be advisable to disable this feature. Note that $QuiverNoLoop takes precedence over this variable.

- $QuiverMultiplier: Set to 1 by default. If $m =$ $QuiverMultiplier is a positive scalar (possibly a formal variable), then all entries of the DSZ matrix Mat used in evaluations of Coulombg, Treeg, or HiggsG are effectively multiplied by m. If m is a matrix, then entries Mat$[[i,j]]$ are multiplied by m_{ij}.

- $QuiverMultiplierAssumption: Specifies assumptions about formal variables used in $QuiverMultiplier, for example $m \in$ Integers

- $QuiverVerbose: If set to False, all consistency tests on data and corresponding error messages will be skipped. Set to True by default.

- $QuiverDisplayCoulombH: If set to True, the routine CoulombBranchFormula will return a list $\{$Q, R$\}$ where Q is the Poincaré-Laurent polynomial and R is a list of replacement rules for the CoulombH factors. Set to False by default.

- $QuiverPrecision: Sets the numerical precision with which all consistency tests are carried out. This is set to 0 by default since all data are assumed to be rational numbers. This can be set to a small real number when using real data, however the user is warned that rounding errors tend to grow quickly.

- $QuiverRecursion: If set to 1 (default value), then the new recursion relations from [**?**, v2] are used for computing CoulombF; if set to 0 the recursion relation from [**?**, v1] is used instead.

- $QuiverOmSbasis: Set to 1 by default. If set to 0, the routines SimplifyOmSbasis and SimplifyOmSbasismult are deactivated, so that the assumption that basis vectors carry $\Omega_S(\ell\gamma_i) = \delta_{\ell,1}$ is relaxed.

- `$QuiverCoulombOpt`: Set to 1 by default. If set to 0, the routines `CoulombF`, `CoulombG`, `CoulombIndex` will use the non-optimized code provided in version 2.0, otherwise they use the optimized code provided in version 2.1. Before v5.1, this was called `$QuiverOpt` !

- `$QuiverFlowTreeOpt`: Set to 3 by default. If set to 0, the original formulation in [**?**, (2.57)] is used. If set to 1 or 2, the first or second alternative recursion in [**?**, (2.64)] will be used to evaluate the tree index. If set to 3, an optimized version of the original flow tree formula in [**?**] is used.

- `$QuiverFlowTreeMethod`: Set to 0 by default. If set to 1, the wall-crossing in `NonAbelianTreeFlowFormula` will be computed the Coulomb branch formula, otherwise it is computed using Reineke's formula for Abelian stack invariants.

- `$QuiverNoVM`: Set to False by default. If set to True, singular hyperplanes from vector multiplet determinant will be ignored in `JKIndex` and `JKIndexSplit`.

- `$QuiverTrig`: Set to True by default. If set to False, exponentiated variables will be used for the residue computation in `JKIndex` and `JKIndexSplit`.

- `$QuiverMaxPower`: Maximal power in the q-expansion of the elliptic genus. Set to 0 by default.

- `$QuiverMutationMult`: Equal to 1 by default. Set to M, defined in Eq. (1.8) of [**?**] when dealing with generalized quivers.

- `$QuiverVertexLabels`: specifies the vertex labels to be used by `PlotQuiver` and `PlotTiling`.

3. Higgs branch formula

- `HiggsBranchFormula[`*Mat_*,*Cvec_*,*Nvec_* `]`: standalone routine which computes the Poincaré-Laurent polynomial of a quiver with DSZ products $\alpha_{ij} =$ `Mat[[`i,j`]]` (possibly rescaled by `$QuiverMultiplier`), dimension vector $N_i =$ `Nvec[[`i`]]`, FI parameters $\zeta_i =$ `Cvec[[`i`]]`, using Reineke's formula. It is assumed, but not checked, that the quiver has no oriented loop;

- `StackInvariant[`*Mat_*,*Cvec_*,*Nvec_*,*y_* `]`: gives the stack invariant $G_{\text{Higgs}}(\gamma,\zeta,y)$ of a quiver with DSZ matrix $\alpha_{ij} =$ `Mat[[`i,j`]]`, possibly rescaled by an overall factor of `$QuiverMultiplier`, FI parameters $\zeta_i =$ `Cvec[[`i`]]`, dimension vector $N_i =$ `Nvec[[`i`]]`, using Reineke's formula; the answer is written in terms of unevaluated q-deformed factorials `QFact[n,y]`;

- `AbelianStackInvariant[`*Mat_*`,`*Cvec_*`,`*y_*`]`: gives the Abelian stack invariant of a quiver with DSZ matrix $\alpha_{ij} =$ `Mat`$[[i,j]]$, possibly rescaled by an overall factor of `$QuiverMultiplier`, FI parameters $\zeta_i =$ `Cvec`$[[i]]$, using Reineke's formula; coincides with `StackInvariant` with `Nvec`$= \{1,\dots 1\}$ except that tests of marginal or threshold stability are performed (unless `$QuiverVerbose` is set to False);

- `OmToOmb[`*f_*`]`: expresses any $\Omega(\gamma, y)$ in f in terms of $\bar{\Omega}(\gamma, y)$'s;

- `OmbToOm[`*f_*`]`: expresses any $\bar{\Omega}(\gamma, y)$ in f in terms of $\Omega(\gamma, y)$'s;

- `OmbToHiggsG[`*Cvec_*`,`*f_*`]`: expresses any $\bar{\Omega}(\gamma, y)$ in `f` in terms of the (unevaluated) stack invariants $G_{\rm Higgs}(\gamma, y)$ using [**?**, (4.1)]; if the first argument is omitted, a generic stability condition is assumed.

- `HiggsGToOmb[`*Nvec_*`,`*y_*`]`: expresses any $G_{\rm Higgs}(\gamma, y)$ in `f` in terms of the rational refined indices $\bar{\Omega}(\gamma, y)$ using [**?**, (4.5)]; if the first argument is omitted, a generic stability condition is assumed.

- `EvalReinekeIndex[`*Mat_*`,`*Cvec_*`,`*f_*`]`: evaluates any `Coulombg[Li, y]` appearing in `f` as `ReinekeIndex[Mat2,Cvec2,y]`, where `Mat2`, `PMat2`, `Cvec2` are computed from the list of vectors `Li` and the quiver data `Mat`, `Cvec`.

- `ReinekeIndex[`*Mat_*`,`*Cvec_*`,`*y_*`]`: computes the stack invariant after perturbing the stability parameters.

4. Coulomb branch formula

- `CoulombBranchFormula[`*Mat_*`,`*Cvec_*`,`*Nvec_*`]`: computes the Dolbeault polynomial of a quiver with DSZ products $\alpha_{ij} =$ `Mat`$[[i,j]]$, dimension vector $N_i =$ `Nvec`$[[i]]$, FI parameters $\zeta_i =$ `Cvec`$[[i]]$, in terms of single-centered invariants $\Omega_{\rm S}$. This standalone routine first constructs the Poincaré-Laurent polynomial, evaluates the Coulomb indices $g_{\rm Coulomb}$, determines the H factors recursively using the minimal modification hypothesis and finally replaces y by t in the argument of $\Omega_{\rm S}$ to construct the Dolbeault polynomial. If `$QuiverDisplayCoulombH` is set to True, the routine returns a list $\{$`Q`, `R`$\}$, where `Q` is the Poincaré polynomial and `R` is a list of replacement rules for the `CoulombH` factors. For quivers without loops, the process can be sped up greatly by setting `$QuiverNoLoop` to True. For complicated quivers it is advisable to implement the Coulomb branch formula step by step, using the more elementary routines described below.

4.1 Generating the sum over all splittings

- `QuiverPoincarePolynomial[`*Nvec_*,*y_*`]`: constructs the Poincaré-Laurent polynomial $\Omega(\gamma, \zeta, y, t)$ as a sum over all partitions of the dimension vector `Nvec`. Coincides with `QuiverPoincarePolynomialRat` for primitive dimension vector;

- `QuiverPoincarePolynomialRat[`*Nvec_*,*y_*`]`: constructs the rational Poincaré-Laurent polynomial $\bar{\Omega}(\gamma, \zeta, y, t)$ as a sum over all partitions of the dimension vector `Nvec` ;

- `QuiverPoincarePolynomialExpand[`*Mat_*,*PMat_*,*Cvec_*, *Nvec_*, *Q_*`]`: evaluates the Coulomb indices g_{Coulomb} and total single-centered indices $\Omega_{\text{tot}}(\alpha_i, y)$ appearing in the Poincaré-Laurent polynomial `Q` of a quiver with DSZ products $\alpha_{ij} =$ `Mat`$[[i, j]]$, perturbed to `PMat`$[[i, j]]$, dimension vector $N_i =$ `Nvec`$[[i]]$, FI parameters $\zeta_i =$ `Cvec`$[[i]]$;

- `ListAllPartitions[`*gam_*`]`: returns the list of unordered partitions $\{\alpha_i\}$ of the positive integer vector γ as a sum of positive, non-zero integer vectors α_i;

- `ListAllPartitionsMult[`*gam_*`]`: returns the list of unordered partitions $\{\alpha_i, m_i\}$ of the positive integer vector γ as a sum of positive, non-zero integer vectors α_i with multiplicity m_i;

- `ListSubQuivers[`*Nvec_*`]`: gives a list of all dimension vectors less or equal to `Nvec`;

- `SymmetryFactor[`*Li_*`]`: gives the symmetry factor $1/|\text{Aut}(\{\alpha_1, \alpha_2, \cdots, \alpha_n\}|$ for the list of charge vectors `Li`;

- `OmTRat[`*Nvec_*,*y_*`]`: gives the rational total invariant $\bar{\Omega}_{\text{tot}}(\gamma, y)$ in terms of $\Omega_{\text{tot}}(\gamma, y)$. Coincides with the latter if γ is primitive.

- `OmTToOmS[`*f_*`]`: expands out any $\Omega_{\text{tot}}(\gamma, y)$ in f into H factors and Ω_{S}'s;

4.2 Computing the Coulomb index

- `SubDSZ[`*Mat_*,*PMat_*,*Cvec_*,*Li_*`]`: gives the DSZ matrix, perturbed DSZ matrix and FI parameters of the Abelian subquiver made of the charge vectors in list `Li`;

- `CoulombF[`*Mat_*,*Cvec_*`]`: returns the index of collinear solutions $F(\{\tilde{\alpha}_1, \cdots \tilde{\alpha}_n\}, \{\tilde{c}_1, \cdots \tilde{c}_n\})$ with DSZ products $\tilde{\alpha}_{ij} =$ `Mat`$[[i, j]]$, FI terms $\tilde{c}_i =$ `Cvec`$[[i]]$ and trivial ordering.

- `CoulombG[`*Mat_*`]`: returns the index of scaling collinear solutions $G(\{\hat{\alpha}_1, \cdots \hat{\alpha}_n\})$ with DSZ products $\hat{\alpha}_{ij} =$ `Mat`$[[i, j]]$ and trivial ordering. The total angular momentum $\sum_{i<j} Mat[[i, j]]$ must vanish;

- CoulombIndex[*Mat_*,*PMat_*,*Cvec_*,*y_*]: evaluates the Coulomb index $g_{\rm Coulomb}(\{\alpha_1,\cdots \alpha_n\};\{c_1,\cdots c_n\},y)$ with DSZ products $\alpha_{ij}=$ Mat$[[i,j]]$, perturbed to PMat[[i,j]] so as to lift accidental degeneracies, possibly rescaled by an overall factor of $QuiverMultiplier, FI terms $c_i=$ Cvec$[[i]]$, angular momentum fugacity y;
- CoulombFNum[*Mat_*]: computes numerically the index $F(\{\tilde\alpha_1,\dots\tilde\alpha_n\},\{\tilde c_1,\dots\tilde c_n\})$ with DSZ matrix $\tilde\alpha_{ij}=$ Mat$[[i,j]]$ and FI parameters $\tilde c_i=$ Cvec$[[i]]$. For testing purposes only, works for up to 5 centers.
- CoulombGNum[*Mat_*]: computes numerically the scaling index $G(\hat\alpha_1,\dots\hat\alpha_n)$ with DSZ matrix $\hat\alpha_{ij}=$ Mat$[[i,j]]$. For testing purposes only, works for up to 6 centers.
- CoulombIndexNum[*Mat_*,*PMat_*,*Cvec_*,*y_*]: returns the Coulomb index $g_{\rm Coulomb}(\{\alpha_1,\cdots \alpha_n\};\{c_1,\cdots c_n\},y)$ with DSZ products $\alpha_{ij}=$ Mat$[[i,j]]$, possibly rescaled by an overall factor of $QuiverMultiplier, FI terms $c_i=$ Cvec$[[i]]$, angular momentum fugacity y, by searching collinear solutions numerically; For testing purposes only, works for up to 5 centers. The output is a list of contributions from each collinear solution.
- EvalCoulombIndex[*Mat_*,*PMat_*,*Cvec_*,*f_*]: evaluates any Coulombg[Li, y] appearing in f as CoulombIndex[Mat2, PMat2, Cvec2, y], where Mat2, PMat2, Cvec2 are computed from the list of vectors Li and the quiver data Mat, PMat, Cvec.
- EvalCoulombIndexAtt[*Mat_*,*PMat_*,*f_*]: evaluates any Coulombg[Li, y] appearing in f as CoulombIndex[Mat2, PMat2, Cvec2, y], where Mat2, PMat2 are computed from the list of vectors Li and the quiver data Mat, PMat and Cvec2 are the respective attractor moduli.

4.3 Contributions from scaling solutions

- CoulombBranchFormulaFromH[*Mat_*,*Cvec_*,*Nvec_*,*R_*]: returns the Dolbeault polynomial of a quiver with DSZ products $\alpha_{ij}=$ Mat$[[i,j]]$, dimension vector $N_i=$ Nvec$[[i]]$, FI parameters $\zeta_i=$ Cvec$[[i]]$, using the rule R to replace all CoulombH factors.
- CoulombHSubQuivers[*Mat_*,*PMat_*,*Nvec_*,*y_*]: computes recursively all CoulombH factors for DSZ matrix Mat, perturbed to PMat, and any dimension vector strictly less than Nvec; relies on the next two routines:
- ListCoulombH[*Nvec_*,*Q_*]: returns returns a pair {ListH, ListC} where ListH is a list of CoulombH factors possibly appearing in the Poincaré-Laurent polynomial Q of a quiver with dimension vector Nvec, and ListC is the list of coefficients which multiply the monomials in $\Omega_{\rm S}(\alpha_i,y)$ canonically associated to the H factors in Q.
- SolveCoulombH[*ListH_*,*ListC_*, *R_*]: returns a list of replacement rules for the CoulombH factors listed in ListH, by applying the minimal modification hypothesis to the coefficients listed in ListC. The last argument is a replacement rule for CoulombH factors associated to subquivers.

- `MinimalModif[f_]`: returns the symmetric Laurent polynomial which coincides with the Laurent expansion expansion of the symmetric rational function f at $y = 0$, up to strictly positive powers of y. Here symmetric means invariant under $y \to 1/y$. In practice, `MinimalModif[f]` evaluates the contour integral in [?], Eq 2.9

$$\oint \frac{\mathrm{d}u}{2\pi \mathrm{i}} \frac{(1/u - u)\, f(u)}{(1 - uy)(1 - u/y)} \tag{4.1}$$

  by deforming the contour around 0 into a sum of counters over all poles of $f(u)$ and zeros of $(1 - uy)(1 - u/y)$. This trick allows to compute (4.1) even if the order of the pole of $f(y)$ at $y = 0$ is unknown, which happens if `$QuiverMultiplier` is a formal variable.

- `MinimalModifFast[f_]`: returns the symmetric Laurent polynomial which coincides with the Laurent expansion expansion of the symmetric rational function f at $y = 0$, up to strictly positive powers of y. This uses the Mathematica function `Residue`, assuming that the order of the pole at $y = 0$ is manifest.

- `EvalCoulombH3[Mat_,f_]`: evaluates any 3-center H factor with multiplicity vector $\{1, 1, 1\}$ appearing in f. No longer in use.

4.4 Simplifying the result

- `SimplifyOmSbasis[f_]`: replaces $\Omega_{\mathrm{S}}(\gamma, y) \to 1$ when γ is a basis vector, unless `$QuiverOmSbasis` is set to 0;

- `SimplifyOmSbasismult[f_]`: replaces $\Omega_{\mathrm{S}}(\gamma, y) \to 0$ when γ is a non-trivial multiple of a basis vector, unless `$QuiverOmSbasis` is set to 0;

- `CoulombHNoLoopToZero[Mat_,f_]`: sets to zero any H factor in `f` corresponding to subquivers without loop, assuming DSZ products $\alpha_{ij} =$ `Mat`$[[i, j]]$; active only on 2-node subquivers if `$QuiverTestLoop` is set to False

- `OmTNoLoopToZero[Mat_,f_]`: sets to zero any Ω_{tot} factor in `f` corresponding to subquivers without loop, assuming DSZ products $\alpha_{ij} =$ `Mat`$[[i, j]]$; active only on 2-node subquivers if `$QuiverTestLoop` is set to False

- `OmSNoLoopToZero[Mat_,f_]`: sets to zero any Ω_{S} factor in `f` corresponding to subquivers without loop, assuming DSZ products $\alpha_{ij} =$ `Mat`$[[i, j]]$; active only on 2-node subquivers if `$QuiverTestLoop` is set to False

- `DropFugacity[f_]`: replaces $\Omega_{\mathrm{S}}(\gamma, y^m, t^m)$ by $\Omega_{\mathrm{S}}(\gamma, t^m)$ everywhere in f

- `SwapFugacity[f_]`: replaces $\Omega_{\mathrm{S}}(\gamma, y^m)$ with $\Omega_{\mathrm{S}}(\gamma, y^m, t^m)$ everywhere in f

5. Mutations

The following routines and environment variables were introduced in `CoulombHiggs.m` v1.1, to allow investigation of mutations of generalized quivers [?]:

- `Mutate[`*Mat_*`,`*k_*`]`: Computes the DSZ matrix of the quiver obtained by applying a right-mutation with respect to the node k. If `k` is a list $\{k_i\}$, then the right mutations k_i are applied successively, starting from the last entry in `k`.
- `MutateRight[`*Mat_*`,`*Cvec_*`,`*Nvec_*`,`*k_*`]`: Computes the DSZ matrix, FI parameters and dimension vector of the quiver obtained by applying a right-mutation with respect to the node k. If `k` is a list $\{k_i\}$, then the right mutations k_i are applied successively, starting from the last entry in `k`. No consistency check on the FI parameters is performed.
- `MutateLeft[`*Mat_*`,`*Cvec_*`,`*Nvec_*`,`*k_*`]`: Computes the DSZ matrix, FI parameters and dimension vector of the quiver obtained by applying a left-mutation with respect to the node k. If `k` is a list $\{k_i\}$, then the right mutations k_i are applied successively, starting from the last entry in `k`. No consistency check on the FI parameters is performed.
- `OmStoOmS2[`*f_*`]`: replaces `OmS[gam, y, t]` by `OmS2[gam, y, t]` anywhere in `f`. This is useful for distinguishing the single-centered invariants of the mutated quiver from those of the original one.
- `MutateRightOmS[`*Mat_*`,`*k_*`,`*f_*`]`: expresses the single-centered invariants `OmS[gam, y, t]` of the original quiver with DSZ matrix `Mat` in terms of the single-centered invariants `OmS2[gam, y, t]` of the quiver obtained by right-mutation with respect to node k, using Eq. 1.13 in [?].
- `MutateLeftOmS[`*Mat_*`,`*k_*`,`*f_*`]`: expresses the single-centered invariants `OmS[gam, y, t]` of the original quiver with DSZ matrix `Mat` in terms of the single-centered invariants `OmS2[gam, y, t]` of the quiver obtained by left-mutation with respect to node k, using Eq. 1.13 in [?].
- `MutateRightOmS2[`*Mat_*`,`*k_*`,`*f_*`]`: expresses the single-centered invariants `OmS2[gam, y, t]` a quiver with DSZ matrix `Mat` in terms of the single-centered invariants `OmS[gam, y, t]` of the quiver obtained by right-mutation with respect to node k. Identical to `MutateRightOmS`, except for swapping `OmS[gam, y, t]` and `OmS2[gam, y, t]`.
- `MutateLefttOmS2[`*Mat_*`,`*k_*`,`*f_*`]`: expresses the single-centered invariants `OmS2[gam, y, t]` a quiver with DSZ matrix `Mat` in terms of the single-centered invariants `OmS[gam, y, t]` of the quiver obtained by right-mutation with respect to node k. Identical to `MutateLeftOmS`, except for swapping `OmS[gam, y, t]` and `OmS2[gam, y, t]`.
- `DropOmSNeg[`*f_*`]`: equates to 0 any $\Omega_{\rm S}(\gamma, y, t)$ where the dimension vector associated to γ has negative components.
- `CompareDSZMatrices[`*Mat1_*`,`*Mat2_*`]`: gives a list of permutations P such that `Mat1` = `Mat2`$[[P, P]]$, or an empty list if no such permutation exists; For `Mat1` = `Mat2`, gives the list of automorphisms of the antisymmetric matrix `Mat1`.

6. Jeffrey-Kirwan residue formula

Beware: the routine JKIndex appears to have become corrupted due to changes in v5. Please use JKIndexSplit or revert to v4.

- JKInitialize[*Mat_*,*RMat_*,*Cvec_*,*Nvec_*]: initializes the internal variables JKFrozenMask, JKFrozenRuleEuler, JKFrozenRuleRat, JKListu, JKListuAll, JKListuDisplay, JKListut, Etavec, JKVertexCoordinates, JKVertexLabels. To be run before JKIndex or JKIndexSplit.

- JKIndex[*ChargeMatrix_*,*Nvec_*,*Etavec_*]: standalone routine, which computes the χ_y genus of the GLSM with given charge matrix, dimension vector and stability parameter. The list of stable flags whose contributions are non-zero is stored in JKRelevantFlags, and the list of the corresponding contributions to the χ_y genus is stored in JKChiGenus.

- JKIndexSplit[*ChargeMatrix_*,*Nvec_*,*Etavec_*,*SplitNodes_*]: standalone routine, which computes the χ_y genus of the GLSM with given charge matrix, dimension vector and stability parameter, using Cauchy's formula to split the vector multiplet determinants associated to the nodes listed in SplitNodes. [**?**].

6.1 Constructing the integrand and extracting the residue

- ZEuler[*ChargeMatrix_*,*Nvec_*]: computes the integrand in the residue formula for the Euler number

- ZRational[*ChargeMatrix_*,*Nvec_*]: constructs the integrand in the residue formula for the χ_y genus in rational representation

- ZTrig[*ChargeMatrix_*,*Nvec_*]: constructs the integrand in the residue formula for the χ_y genus in trigonometric representation

- ZElliptic[*ChargeMatrix_*,*Nvec_*]: constructs the integrand in the residue formula for the elliptic genus

- ListPermutationsWithMultiplicity[*Nvec_*]: computes the list of all multi-partitions of Nvec, represented by a standard permutation, and their multiplicity

- PartitionMultiplicity[*pa_*]: computes the multiplicity of a partition in Cauchy-Bose formula

- ZEulerPartial[*ChargeMatrix_*,*Nvec_*,*ListPerm_*]: constructs the partial contribution to the integrand in the residue formula for the index, corresponding to the list of (possibly empty) permutations associated to each node Listperm.

- `ZTrigPartial[`*ChargeMatrix_*`,`*Nvec_*`,`*ListPerm_*`]`: constructs the partial contribution to the integrand in the residue formula for the χ_y genus in trigonometric representation, corresponding to the list of (possibly empty) permutations associated to each node `Listperm`
- `ZRationalPartial[`*ChargeMatrix_*`,`*Nvec_*`,`*ListPerm_*`]`: constructs the partial contribution to the integrand in the residue formula for the χ_y genus in rational representation, corresponding to the list of (possibly empty) permutations associated to each node `Listperm`
- `ZEllipticPartial[`*ChargeMatrix_*`,`*Nvec_*`,`*ListPerm_*`]`: constructs the partial contribution to the integrand in the residue formula for the elliptic genus in rational representation, corresponding to the list of (possibly empty) permutations associated to each node `Listperm`
- `JKResidueRat[`*Flags_*`,`*f_*`]`: extracts the sum of the residues of `f` (in rational representation) at the specified Flags, weighted with sign; the first entry in `Flags` is the intersection point, the second is a list of r-plets of charges for each flag
- `JKResidueTrig[`*Flags_*`,`*f_*`]`: extracts the sum of the residues of `f` (in trigonometric representation) at the specified Flags, weighted with sign; the first entry in `Flags` is the intersection point, the second is a list of r-plets of charges for each flag

6.2 Enumerating stable flags

- `FindSingularities[`*ChargeMatrix_*`]`: constructs the list of singular hyperplanes for the specified charge matrix. Each item is itself a list containing the intersection point and a list of extended charges associated to the hyperplanes meeting at that point.
- `FindIntersection[`*Sing_*`]`: computes the intersection points of the hyperplanes listed in `Sing`; this may include points on the cylinder, which contribute to the χ_y genus but not to the Euler number ! Ultimately, this will be generalized for the computation of the elliptic genus to include all points on the torus.
- `ListHyperplanesIntersectingAt[`*ListSings_*`,`*Inter_*`]`: collects all the hyperplanes in `ListSings` which intersect at `Inter`
- `TestProjectiveIntersection[`*ListSings_*`,`*Inter_*`]`: tests if the intersection point `Inter` of the list of hyperplanes `ListSings` is projective
- `CollectHyperplanes[`*ListInterrplets_*`,`*Inter_*`]`: collects all the hyperplanes from `ListInterrplets`, which intersect at the point `Inter`
- `TestStableFlag[`*ListHyper_*`,`*Flag_*`,`*Etavec_*`]`: tests if the flag `Flag` constructed out of the hyperplanes in `ListHyper` is stable with respect to `Etavec`; returns sign$(\det \kappa)$ if it is stable, 0 otherwise.

- `FindStableFlags[`*Inter_*,*ListSing_*,*Nvec_*,*Etavec_*`]`: constructs the list of stable flags with stability parameter `Etavec` from the specified list of singular hyperplanes intersecting at `Inter`. Each item in the output is a list containing the intersection point, a list of basis vectors, the reduced charge matrix and the κ matrix.

- `SameFlagQ[`*Q1_*,*Q2_*`]`: tests if the flags Q_1 and Q_2 (represented by square charge matrices) are equivalent

- `FindStableDomains[`*Inter_*,*ListSing_*,*Nvec_*,*Etavec_*`]`: Construct all the flags from the specified list of singular hyperplanes intersecting at `Inter`, and compute their stability domain. Each item in the output is a list containing the intersection point, the ordered hyperplanes defining the flag, and the stability condition. Unlikethe routine `FindStableFlags`, no attempt is made to eliminate equivalent flags.

- `FindDegrees[`*ListSings_*,*NumSing_*`]`: constructs a list of singularities and their degree, combining the poles from `ListSings` with the zeros from the list of hyperplanes `NumSing`

6.3 Conversion and visualisation

- `FlagToHyperplanes[`*Flag_*`]`: translates the flag `Flag`, given as r-plet of charge vectors, into a list of linear combinations of Cartan variables taken from `JKListuDisplay`

- `PartitionToPermutation[`*pa_*`]`: translates the partition `pa` into a standard permutation

- `PermutationToPartition[`*perm_*`]`: translates the standard permutation `perm` into a partition

- `DisplayFlagList[`*ListFlags_*,*ListDegrees_*`]`: Displays the list of flags in human-readable form. The first column gives the intersection point, the second the list of hyperplanes associated to the basis vectors, the third colum gives sign(det κ), the third column gives True if the intersection is projective, False otherwise; the last column gives the degree of the pole.

- `DisplaySingList[`*ListSings_*`]`: Displays the list of singularities in human-readable form. The first column gives the intersection point, the second the list of hyperplanes intersecting at that point, and the last columns gives True if the intersection is projective, False otherwise.

- `DisplayFlagTree[`*f_*`]`: Displays the tree associated to flag `f`, using node positions and labels defined in `JKVertexCoordinates` and `JKVertexLabels`, which is preset by `JKInitialize`.

6.4 Constructing charge matrices

- `ChargeMatrixFromQuiver[`*Mat_*,*RMat_*,*Nvec_*`]`: constructs the charge matrix for a quiver with DSZ matrix `Mat`, R-charge matrix `RMat`, and dimension vector `Nvec`; the last two columns are the R-charge and multiplicity. Do not forget to set `JKFrozenCartan` $= \{\{1,1\}\}$ to decouple the overall $U(1)$. For non-quiver gauge theories, `ChargeMatrix` must be provided by hand.

- `FlavoredRMatrix[`*Mat_*`]`: Constructs a matrix of R-charges with generic flavor potentials θ_i, assuming no oriented closed loop

- `AbelianSubQuiver[`*Mat_*,*RMat_*,*Cvec_*,*Nvec_*,*perm_*`]`: Constructs the DSZ matrix, R-charge matrix and FI parameters for the Abelian quiver associated to the list of permutations `perm` in $\prod_{i=1}^{K} \Sigma_{N_i}$

- `CompleteChargeMatrix[`*ChargeMatrix_*,*Nvec_*`]`: constructs the extended charge matrix consisting of chiral multiplets and vector multiplets

- `PartialChargeMatrix[`*ChargeMatrix_*,*Nvec_*,*perm_*`]`: constructs the extended charge matrix consisting of chiral multiplets and vector multiplet contributions associated to the permutations perm

- `CompleteChargeNumMatrix[`*ChargeMatrix_*,*Nvec_*`]`: constructs the extended charge matrix for the numerators, including both chiral multiplets and vector multiplets

- `PartialChargeNumMatrix[`*ChargeMatrix_*,*Nvec_*,*perm_*`]`: constructs the extended charge matrix for the numerators, consisting of chiral multiplets and vector multiplet contributions associated to unsplit nodes (in which case the permutation is empty)

- `LegCharge[`*Nvec_*,*i1_*,*s1_*,*i2_*,*s2_*`]`: constructs a charge vector with 1 in position (i_1, s_1) and -1 in position (i_2, s_2)

- `TrimChargeTable[`*ChargeMatrix_*`]`: removes the last two columns and frozen entries in charge matrix, corresponding to the R-charge and multiplicity.

7. Flow tree formula

Note: starting with version 6.1, following a suggestion of S. Mozgovoy we no longer perturb the DSZ matrix but only the stability parameters. Moreover, by default the flow tree formula uses an optimized algorithm which only keeps contributions of splittings $\gamma = \gamma_L + \gamma_R$ with $\langle \gamma_L, \gamma_R \rangle > 0$.

- `FlowTreeFormula[`*Mat_*,*Cvec_*,*Nvec_*`]`: computes the index of a quiver with DSZ matrix `Mat`, stability parameters `Cvec` and dimension vector `Nvec` in terms of attractor indices, using the formula [**?**, (2.21)]

- `FlowTreeFormulaRat[`*Mat_*`,`*Cvec_*`,`*Nvec_*`,`*y_*`]`: computes the rational index of a quiver with DSZ matrix `Mat`, stability parameters `Cvec` and dimension vector `Nvec` in terms of rational attractor indices
- `TropicalVertex[`*m_*`,{`*n1_*`,`*n2_*`},`*LiOm1_*`,`*LiOm2_*`]`: computes the index $\Omega(n_1\gamma_1+n_2\gamma_2)$ for an outgoing ray created by scattering of 2 vectors γ_1, γ_2 with Dirac product $m > 0$, assuming that $\Omega(p\gamma_1, y)$ is given by the p-th entry in `LiOm1`, and similarly for γ_2

.

- `TreePoincarePolynomialRat[`*gam_*`,`*y_*`]`: expresses the rational BPS index in terms of terms of attractor indices and tree index
- `TreePoincarePolynomial[`*gam_*`,`*y_*`]`: expresses the BPS index in terms of terms of attractor indices and tree index
- `EvalTreeIndex[`*Mat_*`,`*Cvec_*`,`*f_*`]`: evaluates any `Treeg[Li,y]` appearing in `f` using `TreeIndexOpt` with arguments computed from the full DSZ matrix `Mat` and the stability parameters `Cvec`; if `$QuiverFlowTreeOpt` ¡3, `EvalTreeIndex` instead calls `TreeIndex`
- `TreeIndexOpt[`*Mat_*`,`*Cvec_*`,`*y_*`]`: computes the tree index by summing all planar binary trees with $\langle\gamma_L, \gamma_R\rangle > 0$ at each vertex, after perturbing the stability parameters
- `TreeIndexRecursive[`*Mat_*`,`*Cvec_*`,`*Nvec_*`,`*y_*`]`: recursively constructs the sum over planar binary trees with leaves decorated by basis vectors summing up to `Nvec`.
- `TreeIndex[`*Mat_*`,`*Cvec_*`,`*y_*`]`: computes the tree index by summing all partial tree indices computed using `TreeF`
- `TreeF[`*Mat_*`,`*Cvec_*`]`: computes the partial tree index by summing over stable planar trees using `PlaneTreeSign`
- `PlaneTreeSign[`*Mat_*`,`*Cvec_*`,`*Li_*`]`: computes the contribution to the partial tree index from the grouping `Li` recursively
- `TreeFAlt1[`*Mat_*`,`*Cvec_*`]`: computes the partial tree index by summing over stable planar trees using the first alternative recursion in [**?**, (2.64)]. Will be used by `TreeIndex` if `$QuiverOpt` is set to 1.
- `TreeFAlt1Att[`*Mat_*`]`: computes the attractor contribution to the partial tree index appearing in the first alternative recursion in [**?**, (2.64)]
- `TreeFAlt2[`*Mat_*`,`*Cvec_*`]`: computes the partial tree index by summing over stable planar trees using the second alternative recursion in [**?**, (2.64)]. Will be used by `TreeIndex` if `$QuiverOpt` is set to 2.
- `TreeFAlt2Att[`*Mat_*`]`: computes the attractor contribution to the partial tree index appearing in the second alternative recursion in [**?**, (2.64)]

- PlaneTreeSplitList[*n_*]: constructs all splittings of $\{1,\ldots,n\}$ appearing in the alternative recursions for the partial tree index
- DSZProdAbelian[*Mat_*,*Li1_*,*Li2_*]: computes the DSZ product for two vectors labelled by list of vertices
- SubDSZAbelian[*Mat_*,*Li_*]: computes the DSZ matrix γ_{ij} for the subquiver labelled by a list of vertices
- SubFIAbelian[*Mat_*,*Li_*]: computes the stability parameters c_i for the subquiver labelled by a list of vertices (formerly called SubCvecAbelian)
- OmAttToOmAttb[*f_*]: expresses any $\Omega_*(\gamma,y)$ in f in terms of $\bar{\Omega}_*(\gamma,y)$'s
- OmAttbToOmAtt[*f_*]: expresses any $\bar{\Omega}_*(\gamma,y)$ in f in terms of $\Omega_*(\gamma,y)$'s
- NonAbelianFlowTreeFormula[*Mat_*,*Cvec_*,*Nvec_*]: computes the rational index of a quiver with DSZ matrix Mat, stability parameters Cvec and dimension vector Nvec in terms of rational attractor indices, using the general (non-necessarily primitive) wall-crossing formula at each wall of marginal stability.
- ListFirstWalls[*Mat_*,*Cvec_*,*Nvec_*]: gives the list of relevant walls W_{γ_L,γ_R} for the non-Abelian flow tree formula for $\bar{\Omega}(\gamma,z)$, as a list of $\{\{\gamma_L,n_L\},,\{\gamma_R,n_R\}\}$ such that $\gamma=n_L\gamma_L+n_R\gamma_R$ where γ_L,γ_R are primitive vectors and n_L,n_R positive integers.
- BinarySplits[*Nvec_*]: gives the list of dimension vectors γ_L less than γ, quotiented by the equivalence relation $\gamma_L\to\gamma-\gamma_L$.

8. Attractor Tree formula

- AttractorTreeFormula[*Mat_*,*Cvec_*,*Nvec_*]: computes the index of a quiver with DSZ matrix Mat, stability parameters Cvec and dimension vector Nvec in terms of attractor indices.
- AttractorTreeFormulaRat[*Mat_*,*Cvec_*,*Nvec_*]: computes the rational index of a quiver with DSZ matrix Mat, stability parameters Cvec and dimension vector Nvec in terms of rational attractor indices
- AttractorIndex[*Mat_*,*Cvec_*,*y_*]: evaluates the Attractor index $g_{Sch}(\{\alpha_1,\cdots\alpha_n\};\{c_1,\cdots c_n\},y)$ with DSZ products $\alpha_{ij}=$ Mat$[[i,j]]$, possibly rescaled by an overall factor of $QuiverMultiplier, FI terms $c_i=$ Cvec$[[i]]$, angular momentum fugacity y, as a sum over rooted trees with valency greater or equal to 3.
- EvalAttractorIndex[*Mat_*,*Cvec_*,*f_*]: evaluates any Treeg[Li, y] appearing in f as AttractorIndex[Mat2, Cvec2, y], where Mat2, Cvec2 are computed from the list of vectors Li and the quiver data Mat, Cvec.

- `AttractorF[`*Li_*`,`*Mat_*`,`*Cvec_*`]`: computes the partial Attractor index by summing over Attractor trees; the first argument supplies the list of vertices in each Attractor tree, constructed once and for all by `AttractorIndex` using `AttractorTreeVertices`.

- `Attractorg[`*Mat_*`,`*Cvec_*`]`: computes the sign factor assigned to a given vertex in a Attractor tree, using the prescription from [**?**, (3.23)], $(\mathrm{sign}(0))^m \to 1/(m+1)$ if m is even, or zero if m is odd. If second argument missing, uses attractor stability condition instead.

- `AttractorTreeList[`*n_*`]`: constructs the list of rooted planar trees with valency greater or equal to 3, as a list of groupings (parenthesizings) of $\{1,\ldots,n\}$.

- `AttractorTreeVertices[`*t_*`]`: For a given Attractor tree represented as a grouping `t` of the list $\{1,\ldots,n\}$, constructs the list of $\{\mathrm{vertex},\{\mathrm{children}\}\}$, with the last one in the list being for the root vertex.

- `AttractorTreeTriples[`*t_*`]`: For a given Attractor tree represented as a grouping `t` of the list $\{1,\ldots,n\}$, constructs the list of $\{\mathrm{leftvertex},\mathrm{rightvertex},\mathrm{parent}\}$. Used by `AttractorTreeVertices` to construct the list of vertices.

- `OmAttNoLoopToZero[`*Mat_*`,`*f_*`]`: sets to zero any Ω_* factor in `f` corresponding to subquivers without loop, assuming DSZ products α_{ij} = `Mat[[i,j]]` ; active only on 2-node subquivers if `$QuiverTestLoop` is set to False

- `SimplifyOmAttbasis[`*f_*`]`: replaces $\Omega_*(\gamma,y)\to 1$ when γ is a basis vector, and $\Omega_*(\gamma,y)\to 0$ is γ is a multiple of a basis vector;

9. Joyce formula

The Joyce formula relates rational invariants for two stability conditions:

$$\bar\Omega(\gamma,\zeta_2,y,t)=\sum_{\gamma=\sum_{i=1}^n\gamma_i}\frac{g_{\mathrm{Joyce}}(\{\gamma_i\},\zeta_1,\zeta_2,y)}{|\mathrm{Aut}\{\gamma_i\}|}\prod_{i=1}^n\bar\Omega(\gamma_i,\zeta_1,y,t)\tag{9.1}$$

where

$$g_{\mathrm{Joyce}}(\{\gamma_i\},\zeta_1,\zeta_2,y):=\frac{(-1)^{n-1}}{(y-1/y)^{n-1}}\sum_{\sigma\in S_n}(-y)^{\sum_{i<j}\langle\gamma_{\sigma(i)},\gamma_{\sigma(j)}\rangle}U(\{\gamma_{\sigma(i)}\},\zeta_1,\zeta_2)\tag{9.2}$$

The extent of the validity of this formula, beyond the simple wall-crossing case considered in [**?**], is not yet clear to the author.

- `JoyceFormula[`*Mat_*`,`*Cvec1_*`,`*Cvec2_*`,`*f_*`]`: replaces all $\bar\Omega(\gamma,y)$ and $G_{\mathrm{Higgs}}(\gamma,y)$ in `f`, all assumed to refer to stability `Cvec1`, with their corresponding values at `Cvec2`, using the formula (9.1) or its analogue for stack invariants (formerly called `JoyceSongFormula`);

- JoyceIndex[*Mat_*,*Li_*,*Cvec1_*,*Cvec2_*,*y_*]: computes the index $g_{\text{Joyce}}(\{\gamma_i\},\zeta_1,\zeta_2)$ defined in (9.2).
- UFactor[*Li_*,*Cvec1_*,*Cvec2_*]: computes the factor $U(\{\gamma_i\},\zeta_1,\zeta_2)$ defined in [**?**, §4], using stability condition defined by Slope
- SFactor[*Li_*,*Cvec1_*,*Cvec2_*]: computes the factor $S(\{\gamma_i\},\zeta_1,\zeta_2)$ defined in [**?**, §4], using stability condition defined by Slope
- LFactor[*Mat_*,*Li_*,*y_*]: computes the factor $\mathcal{L}(\{\gamma_i\})$ defined in [**?**, (5.4)]
- JoyceIndexAlt[*Mat_*,*Li_*,*Cvec1_*,*Cvec2_*,*y_*]: computes the index $g_{\text{Joyce}}(\{\gamma_i\},\zeta_1,\zeta_2,y)$ defined using the naive extension of [**?**, (5.5)] to $y \neq 1$
- Slope[*Nvec_*, *Cvec_*]: computes the slope $\sum N_i\zeta_i$.
- PartitionToInvervals[*pa_*]: maps an ordered integer partition of N into a set $0 < a_1 < \cdots < a_m = N$ such that $(a_{j-1}+1,\ldots,a_j)$ label the j-th part.
- CodeToLabeledTreeAlt[*li_*]: constructs the labelled tree with Prüfer code li, substitute for CodeToLabeledTree in Combinatorica package
- DTSpectrumFromOmAtt[*Mat_*,*Cvec_*,*Nvec_*]: computes all rational invariants with dimension vector less or equal to Nvec; the result is a list of replacement rules {Omb[gam,y] :> $\bar{\Omega}(\gamma,\zeta,y)$}
- TrivialStackInvariant[*Mat_*,*Cvec_*,*Nvec_*]: computes the stack invariant $G_{\text{Higgs}}(\gamma,0,y,t)$ for dimension vector $\gamma =$ Nvec and trivial stability condition, in terms of the rational invariants $\bar{\Omega}(\alpha_i,\zeta,y)$ for stability $\zeta =$ Cvec
- GaugeMotive[*Nvec_*,*y_*]: computes the motive $\prod_i \left(y^{2N_i^2}\prod_{j=1}^{N_i}(1-y^{-2j})\right)$ of the gauge group $\prod_i GL(N_i,\mathbb{C})$

10. Non-commutative Donaldson-Thomas invariants

- ListKnownBraneTilings[]: lists the names of brane tilings already coded in the package. The data for each can be extracted from the corresponding item in the global variable BraneTilingsData
- NCDTSeriesFromOmS[*Mat_*, *Framing_*, *Nmin_*,*Nmax_*]: constructs the generating function of NCDT invariants for the quiver with DSZ matrix Mat and framing Framing using the Coulomb branch formula, for dimension vectors Nmin up to NMax. Here Framing is a vector of integers f_i, giving the total number of arrows $q_i^\alpha : \infty \to i$ (when $f_i > 0$), or from $p_i^\beta : i \to \infty$ (when $f_i < 0$)
- NCDTSeriesFromOmAtt[*Mat_*, *Framing_*, *Nmin_*,*Nmax_*]: constructs the generating function of NCDT invariants for the quiver with DSZ matrix Mat and framing Framing using the Flow Tree formula, for dimension vectors Nmin up to NMax. Here Framing is a vector of integers f_i, giving the total number of arrows $q_i^\alpha : \infty \to i$ (when $f_i > 0$), or from $p_i^\beta : i \to \infty$ (when $f_i < 0$)

- `UnrefinedSeriesFromCrystal[`*hMat_*, *fMat_*,*Nn_*`]`: constructs the generating function of unrefined NCDT invariants for the quiver with height matrix `hMat` and framing data `fMat` using the Quiver Yangian algorithm, for dimension vectors with height up to `Nn`. Here `fMat` is a list of two elements: the first is a matrix of weights $h(q_i^\alpha)$ of arrows $q_i^\alpha : \infty \to i$, and the second is a matrix of weights $h(p_i)$ of arrows $p_i^\beta : \infty \to i$. The result is a generating series of Laurent polynomials in y, which reduce to the unrefined DT invariants at $y = 1$, and to the number of molten crystals at $y = -1$. Here y is *not* a refinement parameter !

- `NCDTSeriesFromCrystal[`*hMat_*, *fMat_*,*theta_*, *phi_*, *Nmax_*`]`: constructs the generating function of refined NCDT invariants for the quiver with height matrix `hMat` and framing data `fMat` using the refined Quiver Yangian algorithm with $\mathbb{C}^\times_{\theta,\phi}$ action, for dimension vectors with height up to `NMax`. Here `fMat` is a list of two elements: the first is a matrix of weights $h(q_i^\alpha)$ of arrows $q_i^\alpha : \infty \to i$, and the second is a matrix of weights $h(p_i)$ of arrows $p_i^\beta : \infty \to i$.

- `D6Framing[`*hMat_*,*i_*`]`: constructs the framing data `fMat` for a D6-brane associated to node i, with only one arrow $q_i : \infty \to i$

- `D4Framing[`*hMat_*,*i_*`]`: constructs the framing data `fMat` for a non-compact D4-brane associated to the k-th arrow $i \to j$, with one arrow $q_i : \infty \to i$, one arrow $p_j : j \to \infty$ and superpotential $W = W_0 + p_j \Phi_{ij}^k q_i$

- `FramedDSZ[`*Mat_*,*Framing_*`]`: starting from a quiver with DSZ matrix `Mat`, constructs the DSZ matrix of the framed quiver with f_i arrows from the framing node (labelled 0) to the node i.

- `FramedFI[`*Nvec_*`]`: constructs a random FI parameter for a framed quiver with dimension vector [1; `Nvec`], with first entry much larger than the other ones.

- `BondFactor[`*hMat_*,*i_*,*j_*,*z_*`]`: evaluates the bond factor $\varphi^{i\Rightarrow j}(z)$ in the notations of [**?**], where `hMat` is a matrix whose (i, j)-entry is the list of heights of the arrows from node i to node j. The heights are in turn linear combinations of parameters h_1, h_2, h_3.

- `ChargeFunction[`*hMat_*,*fMat_*, *Crys_*, *i_*, *z_*`]`: constructs the charge function $\Psi^i_{\mathcal{C}}(z)$ for the molten crystal $\mathcal{C}$ = `Crys` in the notations of [**?**]. The crystal $\mathcal{C}$ is encoded in a list of {color, height} for each atom.

- `AddToCrystal[`*hMat_*, *fMat_*, *i_*, *Crys_*`]`: Starting from the molten crystal `Crys`, apply the Quiver Yangian algorithm to construct the list of molten crystals with one additional atom of color i.

- `GrowCrystalList[`*hMat_*, *fMat_*, *CrysList_*`]`: starting from the list of molten crystals `CrysList`, apply the Quiver Yangian algorithm to construct the larger list of molten crystals with up to one additional atom of any color.

- `CrystalDim[`*r_*`,`*Crys_*`]`: computes the dimension vector of the crystal `Crys`, assuming that the colors can take values 1 up to r.

- `CrystaWeight[`*hMat_*, *fMat_*, *theta_*, *phi_*, *Crys_*`]`: computes the power of y associated to the molten crystal `Crys`, using the $\mathbb{C}^\times$ action specified by the angles (θ, ϕ). The default case $\theta = 0$ corresponds to a $\mathbb{C}^\times$ action that preserves the superpotential.

- `DirectedSign[`*theta_*, *phi_*, *lambda_*`]`: computes the sign of $\cos\theta(\lambda_1 \cos\phi + \sin\phi\lambda_2) + \sin\theta\lambda_3$, where $\lambda = \lambda_1 h_1 + \lambda_2 h_2 + \lambda_3 h_3$, used by `CrystalWeight`

- `EulerNorm[`*hMat_*`,`*Nvec_*`]`: Computes the Ringel-Tits norm of the dimension vector `Nvec` from the matrix `hMat`

- `PlotTiling[`*hMat_*`,`*Nn_*, *v_*, *Range_*`,`*Shor_*`,`*Perf_*`]`: produces a 2D plot of the brane tiling defined by the matrix `hMat`, by iterating the arrows `Nn` times, removing those which belong to the perfect matching `Perf`. The argument `v` should be a list of 2D vectors $\{$`v1`$,$`v2`$\}$ determining the vector v associated to an arrow with weight $x_1h_1 + x_2h_2 + x_3h_3$, according to $v = x_1v_1 + x_2v_2$. The plot range is set to `Range` and arrows are shortened by `Shor`. If the argument `Perf` is omitted, all arrows are included. The vertices are labelled from 1 to K, unless specified by `$QuiverVertexLabels`

- `PlotTiling3D[`*hMat_*`,`*Nn_*, *v_*, *Range_*`,`*Perf_*`]`: produces a 3D plot of the Calabi-Yau crystal defined by the matrix `hMat`, by iterating the arrows `Nn` times, removing those which belong to the perfect matching `Perf`. The argument `v` should be a list of 3D vectors $\{$`v1`$,$`v2`$,$`v3`$\}$ determining the vector v associated to an arrow with weight $x_1h_1 + x_2h_2 + x_3h_3$, according to $v = x_1v_1 + x_2v_2 + x_3v_3$. The plot range is set to `Range`. If the argument `Perf` is omitted, all arrows are included.

- `PlotToricFan[`*Fan_*`]`: produces a produces a 2D plot of the polygon with vertices listed in `Fan`

- `ListPerfectMatchings[`*Wp_*`,`*Wm_*`]`: produces the list of cuts for the potential $W = W_+ - W_- =$ `Wp` $-$ `Wm`; each term in $W_\pm$ must be a sum of monomials in `Phi[i,j,k]` with unit coefficient; each perfect matching is represented by a list of triplets $\{i, j, k\}$ such that W is linear in each `Phi[i,j,k]`

- `HeightMatrixToDSZ[`*hMat_*`]`: computes the DSZ matrix associated to the height matrix `hMat`

- `HeightMatrixFromPotential[`*Wp_*`,`*Wm_*`,{`*i1_*`,`*j1_*`,`*k1_*`}, {`*i2_*`,`*j2_*`,`*k2_*`}]`: construct the matrix of heights `hMat` such that the arrow $\Phi^{k_1}_{i_1,j_1}$ has height h_1,the arrow $\Phi^{k_2}_{i_2,j_2}$ has height h_2 and all monomials in the potential $W =$ `Wp` $-$ `Wm` have height h_3

11. Index from deformed Denef equations

In [?], a procedure was proposed for computing the index of Abelian quivers with (or without) oriented loops by summing over solutions to the 'deformed Denef equations'

$$\forall i, \quad \sum_{j\neq i} \frac{\kappa_{ij}}{|z_i - z_j - \frac{\pi\Im z}{\beta} R_{ij}|} = 2\zeta_i \tag{11.1}$$

This is implemented in the following routines (due to G. Beaujard):

- `ExtendedCoulombIndex`[*Mat_*, *PMat_*, *RMat_*, *Cvec_*, *r_*,*y_*]: computes the index of an Abelian quiver with (or without) oriented loops by summing over solutions to (11.1), and computing the contribution of each by residue calculus. The result is a list of contributions associated to the various cluster shapes.

- `CoulombIndexResidue`[*ListSol_*,*Mat_*, *RMat_*,*r_*,*y_*]: computes the contributions of each collinear solution in `ListSol` by evaluating the residues of ione-loop determinant of chiral fields at suitable poles.

- `FindCollinearSolutions`[*Mat_*, *PMat_*, *RMat_*, *Cvec_*, *r_*]: numerically solves the deformed Denef equations with adjacency matrix `Mat` (perturbed to `PMat`), R-charges `RMat`, stability parameters `Cvec` and deformation parameter `r` $= \pi\Im z/\beta$. The algorithm goes through all possible signs of $z_i - z_j - \frac{\pi\Im z}{\beta} R_{ij}$, uses Mathematica `NSolve` function and checks that the resulting solutions are consistent with assumed signs. The result is a list of solutions $\{\{z_i \to \dots\}, \sigma\}$ where σ is the sign of the Hessian around the corresponding solution.

- `ListClusters`[*ListPos_*,*r_*]: produces a list of lists (or clusters) of indices i such that the positions $z_i =$ `ListPos`$[[i]]$ have relative square distances less than $|$`r`$|$ in each cluster.

- `CoulombBranchFormulaNum`[*Mat_*, *PMat_*, *RMat_*, *Cvec_*, *Nvec_*,*r_*,*y_*]: computes the index of a non-Abelian quiver with (or without) oriented loops by summing over solutions to (11.1), and computing the contribution of each by residue calculus. The result is a list of contributions associated to the various cluster shapes. The algorithm is less robust than for Abelian quivers.

- `CoulombBranchResidue`[*Mat_*, *PMat_*, *RMat_*, *Cvec_*, *Nvec_*,*r_*,*y_*]: computes the index of a non-Abelian quiver with (or without) oriented loops by summing over solutions to (11.1), and computing the contribution of each by residue calculus. The result is a list of contributions associated to the various cluster shapes.

12. Utilities

- `PlotQuiver[`*Mat_*`]`: Displays the quiver with DSZ matrix `Mat`. If the entries in `Mat` are lists of integers, then `Mat` is interpreted as the height matrix and the quiver is drawn accordingly. The vertices are labelled from 1 to K, unless specified by `$QuiverVertexLabels`
- `QuiverPlot[`*Mat_*`]`: Displays the quiver with DSZ matrix `Mat` (obsolete).
- `UnitStepWarn[`*x_*`]`: gives 1 for $x > 0$, 0 for $x < 0$, and 1/2 if $x = 0$. Produces a warning in this latter case, irrespective of the value of `$QuiverVerbose`. If so, the user is advised to run the computation again with a different random perturbation. For efficiency, this instruction is no longer used in v2.1, however a warning is still issued if one encounters a Heaviside function with zero argument in the evaluation of the Coulomb indices.
- `GrassmannianPoincare[`*k_*,*n_*,*y_*`]`: computes the Poincaré polynomial of the Grassmannian $G(k, n)$ via Eq. (6.22) in [**?**].
- `CyclicQuiverDSZ[`*Vec_*`]`: constructs the DSZ matrix for a cyclic quiver with a_i arrows from node i to node $i + 1$
- `CyclicQuiverOmS[`*Vec_*,*t_*`]`: computes the refined single-centered index $\Omega_{\rm S}(\gamma_1, \ldots, \gamma_K, t)$ associated to a cyclic Abelian quivers with DSZ matrix $\alpha_{i,i+1}$ = `Vec[[i]]` via Eq (4.29) in [**?**]
- `CyclicQuiverOmAtt[`*Vec_*,*y_*`]`: computes the refined attractor index $\Omega_* r(\gamma_1, \ldots, \gamma_K, y)$ associated to a cyclic Abelian quivers with DSZ matrix $\alpha_{i,i+1}$ = `Vec[[i]]` via Eq (4.21) in [**?**]
- `CyclicQuiverOmAttUnrefined[`*Vec_*`]`: computes the unrefined attractor index $\Omega_*(\gamma_1, \ldots, \gamma_K)$ associated to a cyclic Abelian quivers with DSZ matrix $\alpha_{i,i+1}$ = `Vec[[i]]` using a generalization of Eq (E.2) in [**?**]. This is much faster than `CyclicQuiverOmAtt`.
- `CyclicQuiverTrivialStacky[`*Vec_*,*y_*`]`: computes the stacky invariant associated to a cyclic Abelian quivers with DSZ matrix $\alpha_{i,i+1}$ = `Vec[[i]]` with trivial stability condition via Eq ?? in [**?**]
- `DerangementIndex[`*Vec_*,*y_*`]`: computes the number of derangements of a set of `Vec[i]` objects of color i, weighted by $y^{n_+ - n_-}$ where n_+/n_- are the number of objects which are displaced forward/backward with respect to the standard ordering $11 \ldots 222 \ldots K$ [**?**]
- `FIFromZ[`*Nvec_*,*Zvec_*`]`: computes the FI parameters $\{\zeta_i\}$ from the vector of central charges `Zvec` = $\{Z_i\}$ and dimension vector `Nvec` = $\{N_i\}$ via $\zeta_i = \Im(e^{-i\phi} Z_i)$, where ϕ is the argument of $\sum_i N_i Z_i$. The parameters ζ_i are rounded up to the nearest rational number with denominator less than `$QuiverPerturb1`
- `AttractorFI[`*Mat_*,*Nvec_*`]`: gives the attractor point $\zeta_i^* = -\sum_j \gamma_{ij} N_j$

- QDeformedFactorial[*n_*,*y_*]: gives the q-deformed factorial $[n, y]!$
- EvalQFact[*f_*]: evaluates any QFact[n,y] appearing in f
- ExpandTheta[*f_*]: replaces Theta and Eta by their q-expansions, truncated at order $QuiverMaxPower
- qSeries[*f_*]: Replaces τ by $\log q/(2\pi i)$, and Taylor expand around $q = 0$ up to order $QuiverMaxPower
- SubVectors[*Nvec_*]: List all positive dimension vectors which are less than Nvec, including the zero vector and Nvec itself
- EulerForm[*Mat_*]: construct the antisymmetric Ringel-Tits form from the intersection matrix Mat (coincides with the latter if Mat is antisymmetric)
- ToPrimitive[*Nvec_*]: gives the pair $\{\gamma', d\}$ such that γ' is primitive, $d \geq 1$ and $\gamma = d\gamma'$
- ListLoopRCharges[*Mat_*,*RMat_*]: Lists the oriented closed loops and corresponding R-charge
- TestNoLoop[*Mat_*,*Li_*]: tests if the subquiver associated to the charge vectors Li has oriented closed loops
- RandomDSZWithNoLoop[*n_*,*kmax_*]: generates a random antisymmetric $n \times n$ matrix with off-diagonal entries less than kmax in absolute value, ensuring that the quiver has no loop
- RandomDSZWithLoop[*n_*,*kmax_*]: generates a random antisymmetric $n \times n$ matrix with off-diagonal entries less than kmax in absolute value, ensuring that the quiver has at least one loop
- RandomFI[*Nvec_*]: generates a random set of FI parameters ζ_i between -1 and 1, such that $\sum \zeta_i$ Nvec[[i]] = 0; (previously called RandomCvec)
- FastResidue[*f_*,{ *x_*, *x0_* }]: computes the residue of f at $x = x_0$
- DSZProd[*Mat_*,*Nvec1_*, *Nvec2_*]: computes the Dirac product $\sum \alpha_{ij} N_i^1 N_j^2$
- ReduceDSZMatrix[*Mat_*,*Li_*]: sets Mat[[i, j]] = Mat[[j, i]] = 0 for all elements $\{i, j\}$ in Li, and returns the resulting matrix. If $i = j$, then the i-th row and column of Mat are set to 0.
- HiggsedDSZ[*Mat_*,*i_*,*j_*]: starting from a quiver with DSZ matrix Mat, constructs the DSZ matrix of the quiver where the node j has been merged with the node i.
- ConnectedQuiverQ[*Mat_*,*Nvec_*]: returns True is the restriction of the quiver with DSZ matrix Mat to the nodes where Nvec has non-trivial support is connected.
- PlethysticExp[*f_*,*Nmax_*]: computes the plethystic exponential of f, assuming that it is a function of x[i] and y only, keeping the first Nmax terms in the sum.
- PlethysticLog[*f_*,*Nmax_*]: computes the plethystic logarythm of f, assuming that it is a function of x[i] and y only, keeping the first Nmax terms in the sum.
- QuiverMultiplierMat[*i_*,*j_*]: Returns $QuiverMultiplier if it is a scalar, or $QuiverMultiplier[[$i, j$]] if $QuiverMultiplier is a matrix.