User Guide for `TwoVect.m`

A Mathematica Package for 2–Vector Spaces

Daniel A. Roberts

September 20, 2012

Contents

1 Introduction

This package was designed as an aid to performing calculations in 2Vect, the 2-category of 2-vector spaces. Particularly important examples of such calculations involve verifying the axioms of in monoidal 2-vector spaces, of which modular tensor categories (MTCs) are of particular interest. The package was developed as part of my Master's thesis at the University of Oxford [7]. Please see the thesis for a more in depth overview of the category theoretic structures involved and the underlying design of the package.

1.1 2-Categories and 2-Vector Spaces

A category consists of a set of *objects* and a set of *arrows* or *morphisms* that go between the objects. 2-categories are further generalizations in category theory and may be thought of as *categories of categories* [6]. A 2-category extends the idea of a category—objects and morphisms—to include a higher level structure: morphisms between parallel morphisms. In lieu of such complicated terminology, the field has created the terms *0-cell*, *1-cell*, and *2-cell* to mean *object*, *morphism*, and *morphism between parallel morphisms*, respectively.

Vect is the category of vector spaces. The objects are n-dimensional vector spaces and morphisms are the $n \times m$ linear maps between n-dimensional vector spaces and m-dimensional vector spaces.

2Vect is the category of 2-vector spaces. The objects, **N**, are n copies of the 1-category **Vect** in a Cartesian product. **2Vect** is a *symmetric monoidal* 2-category—it is endowed with a *bifunctor* that maps 2 objects to another (and the order of the combination may be swapped). In this paradigm, modular tensor categories are structures to be found *within* **2Vect**. In fact, the structure of **2Vect** is general enough to represent many different types of categories, not just MTCs.

Practically, 2-vector spaces involve a lot of 2-matrices whose elements themselves are matrices (i.e. matrices of matrices).Composition and tensor product of 0-cells, 1-cells, and 2-cells in **2Vect** is "finite dimensional higher linear algebra"—i.e. a "2"-linear algebra—which is just as important as ordinary finite-dimensional linear algebra, but technically much more involved. The package `TwoVect.m` implements this 2-linear algebra.

2Vect	label	actual representation	isomorphism
0-cell	$\mathbf{2}$	$\begin{bmatrix} \mathbf{Vect}_a \\ \mathbf{Vect}_b \end{bmatrix}$	2
1-cell	$f : \mathbf{2} \to \mathbf{3}$	$\begin{bmatrix} \mathbb{C}^n & \mathbb{C}^m \\ \mathbb{C}^p & \mathbb{C}^q \\ \mathbb{C}^r & \mathbb{C}^s \end{bmatrix}$	$\begin{bmatrix} n & m \\ p & q \\ r & s \end{bmatrix}$
"state" of 0-cell: **2**	$\psi \in \mathbf{2}$	$\begin{bmatrix} \mathbb{C}^a \\ \mathbb{C}^b \end{bmatrix}$	$\begin{bmatrix} a \\ b \end{bmatrix}$
"state" as a 1-cell	$\psi : \mathbf{1} \to \mathbf{2}$	$\begin{bmatrix} \mathbb{C}^a \\ \mathbb{C}^b \end{bmatrix}$	$\begin{bmatrix} a \\ b \end{bmatrix}$

Table 1: Example of a 0-cell, 1-cell, and "states" of a 0-cell for **2Vect**. The 0-cell is actually two copies of the category **Vect**, but we represent it as the number 2. The 1-cell is actually a 3×2 matrix of vector spaces, but we represent it as a 3×2 matrix of numbers. A state of **2** is a 2×1 matrix of vector spaces (i.e. a column vector), but we represent it as a 2×1 matrix of numbers. Furthermore, the equivalence between the two ways of thinking about "states" should be clear. From the categorical viewpoint of **2Vect**, we do not "look inside" the 0-cells to find the states, but can access them via 1-cells. We can then apply morphisms to them via 1-cell composition.

To represent **2Vect**, we make use of isomorphisms between the n-cells in the 2-category and various orders of matrices [3, 1, 5]. We use natural numbers to represent 0-cells, matrices of natural numbers to represent 1-cells, and matrices of matrices of complex numbers to represent 2-cells. Formally, we've modeled **2Vect** in *Mathematica* as a *skeletal*, *non-strict* monoidal 2-category. So, in our *Mathematica* model, *equivalent* 0-cells are *equal* and *isomorphic* 1-cells are *equal*. An example of the actual and isomorphic representations of 0-cells and 1-cells is shown in Table 1. An example of a 2-cell is shown in equation 2.

Consider the following example. If we have the following 1-cells $f : \mathbf{2} \to \mathbf{3}$ and $g : \mathbf{2} \to \mathbf{3}$, they will both be 3×2 dimensional matrices going from 2 copies of **Vect** to 3 copies of **Vect**. Each element represents a vector space of dimension given by that element. Let them be given by

$$f = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 2 & 1 \end{bmatrix} \tag{1}$$

$$g = \begin{bmatrix} 3 & 4 \\ 2 & 5 \\ 2 & 1 \end{bmatrix}$$

then a 2-cell $\alpha : f \Rightarrow g$ would have the form

$$\begin{bmatrix} \begin{bmatrix} a_{1,1} \\ a_{2,1} \\ a_{3,1} \end{bmatrix} & \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \\ b_{4,1} & b_{4,2} \end{bmatrix} \\ \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \end{bmatrix} & \begin{bmatrix} d_{1,1} & d_{1,2} & d_{1,3} & d_{1,4} \\ d_{2,1} & d_{2,2} & d_{2,3} & d_{2,4} \\ d_{3,1} & d_{3,2} & d_{3,3} & d_{3,4} \\ d_{4,1} & d_{4,2} & d_{4,3} & d_{4,4} \\ d_{5,1} & d_{5,2} & d_{5,3} & d_{5,4} \end{bmatrix} \\ \begin{bmatrix} e_{1,1} & e_{1,2} \\ e_{2,1} & e_{2,2} \end{bmatrix} & [f_{1,1}] \end{bmatrix} \tag{2}$$

1.2 Structural Isomorphisms

A further complication arises due to our representation not being strict. While working with *symmetric 2-monoidal 2-categories* like **2Vect**, we quickly find that many expressions that are equal when working with *symmetric monoidal categories* will only hold up to isomorphism. The presence of 2-cells provides an extra "degree of freedom" in the structure by which certain equations no longer hold.

For instance, in the definition of a symmetric monoidal category (for instance, **Vect**), we are told $S \circ (f \otimes g) = (g \otimes f) \circ S$, where f and g are morphisms, and S is the swap operation [2]. In **2Vect**, these 1-cells will no longer be of the same *type*. Furthermore, related equations between 2-cells will no longer hold. To mediate between type, **2Vect** is endowed with a structural 2-cell isomorphism, denoted $\sigma_{f,g}$, to allow monoidally combined 1-cells to commute with the swap.[1]

However, by no longer of the same *type*, we mean something subtle. The two 1-cells, $S \circ (f \boxtimes g)$ and $(g \boxtimes f) \circ S$, must always give the same functor (i.e. the same 1-cell matrix) in our representation. But these are not the *same* one cell, even if they have the same representation. This is due to the strictification implicit in our representation of 1-cells, our treating matrices of vector spaces as matrices of numbers. Composed parallel 1-cells leads to horizontal composition in the 2-cells that go between them. At the 2-cell level, we will have different 2-cells, and without any subtlety, $id_S \bullet (\mu \boxtimes \nu) \neq (\nu \boxtimes \mu) \bullet id_S$, where $\mu : f \Rightarrow f'$ and $\nu : g \Rightarrow g'$.[2] Essentially, this is just a statement about 2-matrices, and in many cases about commutativity or associativity of the underlying tensor products and direct sums. From a non-categorical perspective, going from $S \circ (f \boxtimes g)$ to $(g \boxtimes f) \circ S$ will induce an implicit change of basis either the element-matrices or outer-matrices of the associated 2-cells (or both).

As a result, we require three such structures, the swap structure σ, the interchange structure τ, and the associator structure ω. To represent them, we make use of the

[1] The 2-cell is denoted $\sigma_{f,g}$ because it is the structure tailored to two *specific* 1-cells, f and g, that is a 2-cell. The components of σ are 2-cells, and σ itself is some higher level 2-transformation. (cf. the relationship between natural transformations and their component arrows). This will be true of all of the structural isomorphism presented in this section.

[2] This is not a surprise since we are not using any strictification in our representation of 2-cells—we are representing them as they actually are.

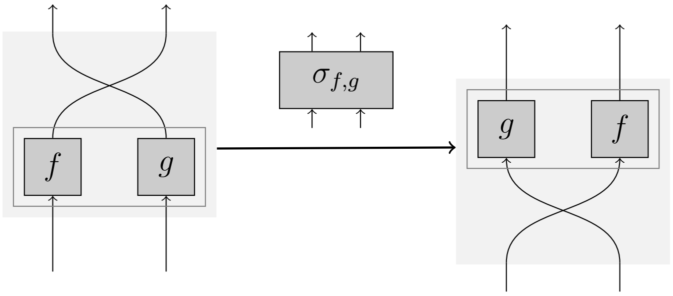


Figure 1: σ structural isomorphism

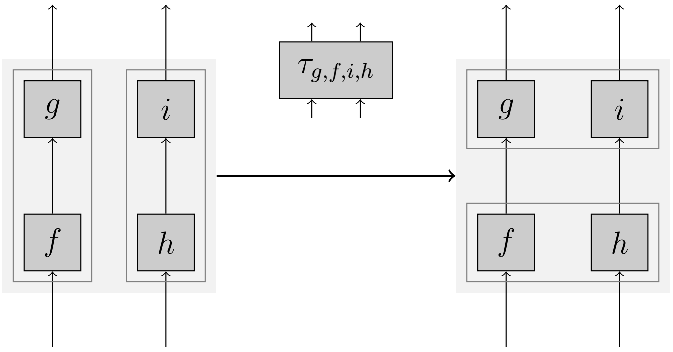


Figure 2: structural isomorphism τ

diagrammatic language (or graphical calculus) as formulated by Joyal and Street for working with symmetric monoidal categories [4], but we extend it to work with symmetric monoidal 2-categories, such as **2Vect**.

For 1-categories, each picture represents a morphism or *process*[3]—time flows from bottom to top, objects are represented by lines, and morphisms are represented by boxes. Objects are monoidally combined by being placed next to each other. The identity morphism is just a line—i.e. nothing happens to the object. Two different pictures, two different morphisms, are either equal or unequal; the structure of 1-categories does not allow for any other relationship.

In 2-categories, pictures can be related in many ways—they can be equal, they can be isomorphic, or related by a non-invertible 2-cell. Thus, we can have arrows between pictures—arrows between morphisms—i.e. 2-cells. Additionally, these diagrams are bracketed in 2-dimensions or *boxed* in order to specify their type. Figure 1 is the diagram for the 2-cell $\sigma_{f,g}$, which moves the 1-cells f and g past a swap (and in the process swaps their order). Figure 2 is the diagram for the 2-cell $\tau_{g,f,i,h}$, which mediates between the 1-cells $(g \circ f) \boxtimes (i \circ h)$ and $(g \boxtimes i) \circ (f \boxtimes h)$. Figure 3 is the diagram for the 2-cell $\omega_{h,g,f}$, which mediates between the 1-cells $h \circ (g \circ f)$ and $(h \circ g) \circ f$.

For 1-categories, bracketing is important, and one usually makes use of different structures to rebracket. Similarly, for 2-categories, this generalization of rebracketing—boxing—is necessary to compute equations.

For more information on these structures and for their explicit construction, please

[3] This goes hand-in-hand with the focus of category theory being the morphisms rather than the objects.

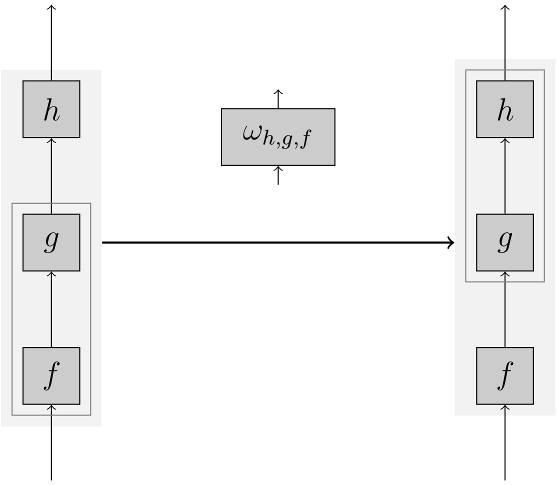


Figure 3: structural isomorphism ω

see the thesis [7].

1.3 Finding 1-Categories, N-Dimensional 2-Vector Spaces (e.g. MTCs)

Since the objects of **2Vect** are categories, we will find our 1-categories of interest, which are necessarily N-dimensional 2-vector spaces, as objects inside **2Vect**. For instance, **Fib**, the Fibonacci modular tensor category (which is a N-dimensional 2-vector space with $N = 2$ and will be discussed at length in the first tutorial in Section 5.1), is thus the object **2**, endowed with a bunch of extra structure. [4] As such, the N-dimensional 2-vector space can be "accessed" as $\mathrm{HomCat}(1, N)$ inside **2Vect**. So, an *object* of N corresponds to a 1-cell in **2Vect**, and a *morphism* of N corresponds to a 2-cell of **2Vect**.

A monoidal 2-vector space can be seen as structure internal to **2Vect**, just as an ordinary algebra is structure internal to **Vect**. Therefore, just as checking the axioms for an ordinary algebra is an exercise in ordinary linear algebra, checking the axioms of a monoidal 2-vector space is an exercise in 2-linear algebra, which is perfectly suited for this package.

In addition handling monoidal 2-vector spaces, this package can also deal with fancier structures and properties—i.e. braidings, twists, duals for objects, modularity, *etc.* All of the axioms of MTCs are already built in, but the package may be easily extended to work with any 2-vector space of interest (see Section 5.2 for a tutorial).

1.4 Calculate String Diagrams

Finally, this package can make it very easy to perform string diagram calculations within a given monoidal 2-vector space, such as a modular tensor category. For instance, the diagram in Figure 4 in **Fib** is computed with the following code:

[4] i.e. a 1-cell $m : \mathbf{4} \to \mathbf{2}$ given by $\left[\begin{smallmatrix}1&0&0&1\\0&1&1&1\end{smallmatrix}\right]$ and a bunch of other 2-cells that all together satisfy the MTC axioms, some of which is discussed in Sections 4.5 and 5.2.

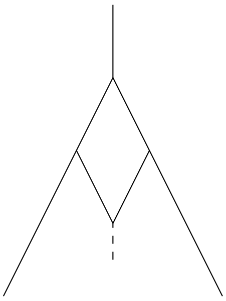


Figure 4: string diagram for a nontrivial process of type $\tau \otimes \tau \to \tau$

```
Needs["MTCategories`"]
SetMTCFib[]
proj = {{{−1, {1 → 0}}}, {{{1}}}} // TwoCell
inj = {{{{1}}}, {{−1, {0 → 1}}}} // TwoCell
proj · (proj ⊗ proj) · α[τ ⊗ τ, τ, τ] · (inv[α[τ, τ, τ]] ⊗ τ) ·
        ((τ ⊗ inj) ⊗ τ) · (inv[ρ[τ]] ⊗ τ) // FullSimplify // TwoCellForm
```

The first line loads the required package. The second line selects the monoidal category "Fib" to be active. The third and fourth lines define projection and injection maps. The final line is a direct algebraic representation of the string diagram, as given in equation 3. Mathematica returns the 2-cell

$$\begin{pmatrix} 0_{1\to 0} \\ \begin{pmatrix} -\frac{\sqrt{\text{GoldenRatio}}}{1+\text{GoldenRatio}} \end{pmatrix} \end{pmatrix}$$

as the solution.

$$\begin{tikzcd}
 & (\tau \otimes u) \otimes \tau \arrow[r, "(\mathrm{id}_\tau \otimes \eta_\tau)\otimes \mathrm{id}_\tau"] & (\tau \otimes (\tau \otimes \tau)) \otimes \tau \arrow[ddr, "\alpha^{-1}_{\tau,\tau,\tau} \otimes \mathrm{id}_\tau"] & \\
\tau \otimes \tau \arrow[ur, "\rho_\tau^{-1} \otimes \mathrm{id}_\tau"] & & & \\
 & & & ((\tau \otimes \tau) \otimes \tau) \otimes \tau \arrow[ddl, "\alpha_{\tau\otimes\tau,\tau,\tau}"] \\
\tau & & & \\
 & \tau \otimes \tau \arrow[ul, "p_\tau"] & (\tau \otimes \tau) \otimes (\tau \otimes \tau) \arrow[l, "p_\tau \otimes p_\tau"] &
\end{tikzcd}$$

(3)

For the full tutorial, please see Section 5.1.

2 Setup

2.1 Package "Installation"

The package consists of three separate package files: *TwoVect.m*, *MTCategories.m*, and *MTCategory.m*. To "install" them, either place them in a directory on *Mathematica*'s *path* or add their location to the *path*. To see directories on the *path*, enter the following in a notebook:

```
$Path
```

This will output a list of all the directories on the *path*. To add the current directory of the project files to the path, enter:

```
AppendTo[$Path, ToFileName[{PackageLocation}]];
```

where *PackageLocation* is a string to the directory where the files are located. For example, if the files are in the directory `C:\Category Theory\MTCPackage\`, you would enter:

```
AppendTo[$Path, ToFileName[{"C:\\Category Theory\\MTCPackage"}]];
```

Standard escape characters are necessary when entering strings in *Mathematica*, so you must enter "\\" in order to enter the backslash in the path to the files.

N.B. The *AppendTo* command must be used each time *Mathematica* is loaded (since the *path* resets). If, instead, the files are placed in a default *path* directory, no further action is necessary upon *Mathematica* restarts.

2.2 Using a Sub-Package

To use a function from any of the packages, *Mathematica* must be told that the function you are calling requires the package. This can be done at the top of your notebook using the *Needs* command. The format is the name of the package followed by the ' character. For instance, to use the three different packages, in order from lowest level to highest level, the commands would be:

```
Needs["TwoVect`"]
Needs["MTCategories`"]
Needs["MTCategory`"]
```

Furthermore, due to dependencies (the higher level sub-packages depend on all the lower level sub-packages), loading a higher level package will automatically load all the functions from the lower level package.

2.3 Sub-Package Differences

This package contains three different sub-packages, each operating at a different level (roughly low-level, mid-level, and high-level), where the level refers to how close the user is to the underlying 2-category, **2Vect**, and simplicity of the coding.

`TwoVect.m` gives you access to the raw 2-category **2Vect**, and has no structural support for containing special categories (like MTCs), although, of course, such structure can be implemented on top of it.

`MTCategories.m` implements exactly that structure. This sub-package allows for manipulation of any number of modular tensor categories and of monoidal functors between them.

`MTCategory.m` is the highest level package. It allows for manipulation of any particular monoidal 2-vector space, which the user can set globally. In most cases in the user guide, the monoidal 2-vector space of interest will be a modular tensor category, although it does not have to be. The MTC must be formally set using the **SetMTC** function, but the syntax is very simple and makes almost no references to 2-category formalism. In many cases, the syntax is identical to the labels you place on arrows in MTC arrow diagrams. Finally, string diagram calculations within the category can be performed directly using this package. Please see Section 5.1.1 for an example.

3 Input Format and Display Commands

3.1 *Mathematica* Special Symbols

To enter Greek letters, operators, or other special symbols into a *Mathematica* notebook, you can go to *Insert → Special Characters*, and then select the desired symbol.

Alternatively, you can use keyboard shortcuts. This is done by pressing the *esc* key, entering the shortcut, and then pressing *esc* again. The shortcut for most Greek letters is their corresponding Roman letter. For instance, to enter α, you type *esc a esc.*

Finally, the shortcut for $\otimes$ is '*c**' (so you enter *esc c * esc*), the shortcut for $\cdot$ is '.', the shortcut for $\odot$ is '*c.*', and the shortcut for $\circ$ is '*sc*'.

3.2 0-Cells

In this representation, 0-Cells are numbers. Thus, there is no special input format or display command. Furthermore, they are usually implicitly handled by the 1-cells and 2-cells and are not normally inputted by themselves.

3.3 0-Dimensional Matrices

This package makes use of 0-dimensional matrices, where at least one of the dimensions (but not necessarily both) of the matrix are 0. These are entered in the form:

```
zeroMat = {-1,{c->r}}
```

where c is the number of columns, r is the number of rows, and at least one of r and c must be 0. The whole input $\{-1, \{c \to r\}\}$ stands in explicitly for the empty $r \times c$ matrix.

3.4 1-cells

In this representation, 1-cells are matrices. The safest way to input a 1-cell is as a matrix. The matrix can then be passed to the function **OneCell**. For instance, to input the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

(the fusion rules for **Fib**) and convert it to a 1-cell, you would use the following commands into a *Mathematica* notebook:

```
mFib = OneCell[{{1, 0, 0, 1}, {0, 1, 1, 1}}]
```

As a result, the input matrix

```
{{1, 0, 0, 1}, {0, 1, 1, 1}}
```

is converted to

```
{{{1, 0, 0, 1}, {0, 1, 1, 1}},{4 -> 2}}
```

which is the form in which 1-cells are stored. The second part shows that the 1-cell goes from the 0-Cell **4** to the 0-Cell **2** (it is also the number of columns and the number of rows of the input matrix, respectively).

Alternatively, you could have just inputted

```
mFib = {{{1, 0, 0, 1}, {0, 1, 1, 1}},{4 -> 2}}
```

with the same result. However, *be careful*, by inputting 1-cells directly, there is no check to ensure that the matrix corresponds correctly with the inputted 0-Cells. Instead, it is better to input a matrix and convert it.

To display the 1-cell in a form similar to the *Mathematica* **MatrixForm** command, you can use the command **OneCellForm**:

```
OneCellForm[mFib]
```

or, identically,

```
mFib // OneCellForm
```

with the result being

$$\left\{ \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}, (4 \rightarrow 2) \right\}$$

3.5 2-cells

In this representation, 2-cells are matrices of matrices, or 2-matrices. *Mathematica* can naturally handle 2-matrices, and then they can be converted to 2-cells with the command **TwoCellForm** (in analogy to 1-cells). Let us look at the associator for **Fib**. As a 2-cell, is has the form:

$$\alpha = \begin{bmatrix} [1] & \emptyset_{0\times 0} & \emptyset_{0\times 0} & [1] & \emptyset_{0\times 0} & [1] & [1] & [1] \\ \emptyset_{0\times 0} & [1] & [1] & [1] & [1] & [1] & [1] & \begin{bmatrix} \varphi^{-1} & \varphi^{-\frac{1}{2}} \\ \varphi^{-\frac{1}{2}} & -\varphi^{-1} \end{bmatrix} \end{bmatrix}$$

where, for instance (and using our conventions for the fusion rules in Section 4.4), the matrix in the bottom right represents $\alpha_{\tau}^{\tau\tau\tau}$. Additionally, φ is the golden ratio, and $\emptyset_{0\times 0}$ symbolizes 0-dimensional matrix of dimensions 0×0. To input this into *Mathematica* as a 2-matrix, we can either input it all at once, or input all the individual element-matrices first and then create a matrix of them. For this example, we will show the latter method:

```
α11 = {{1}}
α12 = {-1,{0->0}}
α13 = {-1,{0->0}}
α14 = {{1}}
α15 = {-1,{0->0}}
α16 = {{1}}
α17 = {{1}}
α18 = {{1}}
α21 = {-1,{0->0}}
α22 = {{1}}
α23 = {{1}}
α24 = {{1}}
α25 = {{1}}
α26 = {{1}}
α27 = {{1}}
α28 = {{GoldenRatio^-1,GoldenRatio^(-1/2)},{GoldenRatio^(-1/2),-GoldenRatio^-1}}
```

N.B. all of these entries MUST be matrices. Thus, for instance, for $\alpha 11$ we must use still use two curly brackets and not just enter {1}.

Now, the element matrices can be combined into a 2-matrix

```
αTwoMat = {{α11,α12,α13,α14,α15,α16,α17,α18},{α21,α22,α23,α24,α25,α26,α27,α28}}
```

We can display 2-matrices in a pleasing form with the command **TwoMatrixForm** in a manner similar to **MatrixForm**. The 2-matrix can then be converted into a 2-cell by:

```
αFib = TwoCell[αTwoMat]
```

which returns the ridiculously ungainly

```
{{{{{1}}, {-1, {0 -> 0}}, {-1, {0 -> 0}}, {{1}}, {-1, {0 -> 0}}, {{1}}, {{1}},
    {{1}}}, {{-1, {0 -> 0}}, {{1}}, {{1}}, {{1}}, {{1}}, {{1}}, {{1}},
    {{1/GoldenRatio, 1/Sqrt[GoldenRatio]}, {1/Sqrt[ GoldenRatio],
    -(1/GoldenRatio)}}}}}, {{{{1, 0, 0, 1, 0, 1, 1, 1},
    {0, 1, 1, 1, 1, 1, 1, 2}}, {8 -> 2}} -> {{{1, 0, 0, 1, 0, 1, 1, 1},
    {0, 1, 1, 1, 1, 1, 1, 2}}, {8 -> 2}}}}}
```

To display this in a reasonable form, use **TwoCellForm**:

```
TwoCellForm[αFib]
```

which displays

$$\left\{\begin{pmatrix} (1) & 0_{0\to0} & 0_{0\to0} & (1) & 0_{0\to0} & (1) & (1) & (1) \\ 0_{0\to0} & (1) & (1) & (1) & (1) & (1) & (1) & \begin{pmatrix} \frac{1}{GoldenRatio} & \frac{1}{\sqrt{GoldenRatio}} \\ \frac{1}{\sqrt{GoldenRatio}} & -\frac{1}{GoldenRatio} \end{pmatrix} \end{pmatrix},\right.$$

$$\left\{\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 2 \end{pmatrix},(8\to2)\right\}\to\left\{\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 2 \end{pmatrix},(8\to2)\right\}\}$$

where the $0_{0\to0}$ entries are the 0×0, 0-dimensional matrices. Following the 2-matrix part of the 2-cell is the source and destination 1-cells.

Alternatively, you can enter 2-cells directly in the correct form (without creating 2-matrices), but, again, *be careful* because there is no check to make sure the 1-cell and 0-Cells are in the correct form (i.e. that all the dimensions match up correctly).

Finally, and probably the easiest method for inputting 2-cells, you can get a 2-cells in the correct form with variable entries by specifying its source and destination 1-cells. The values of the individual element-matrix elements (i.e. the numbers) can then easily be filled in by giving values to those variables. This method would be analogous to entering a matrix by specifying its dimensions, getting a matrix of variables of those dimensions, and then individually setting those variables to their appropriate values.

As an example, let us enter the braid β for **Fib**. We want to enter

$$\beta = \begin{bmatrix} [1] & \emptyset_{0\times0} & \emptyset_{0\times0} & \left[e^{-\frac{4}{5}i\pi}\right] \\ \emptyset_{0\times0} & [1] & [1] & \left[e^{\frac{3}{5}i\pi}\right] \end{bmatrix} \tag{4}$$

Furthermore, we know that from the 2-category perspective, $\beta : m \Rightarrow m \circ S_{\mathbf{2},\mathbf{2}}$, with $S_{\mathbf{2},\mathbf{2}} : \mathbf{2} \boxtimes \mathbf{2} \to \mathbf{2} \boxtimes \mathbf{2}$ and $m : \mathbf{2} \boxtimes \mathbf{2} \to \mathbf{2}$, where m are the 1-cell fusion rules for **Fib** and $S_{\mathbf{2},\mathbf{2}}$ is the swap 1-cell. Thus, to get a β of the right form we can use the function **GenerateVariableTwoCell**. The inputs are the source and destination 1-cells. We already have the source, *mFib*, from Section 4.4. If not, we need to load it with the command

```
mFib = {{1, 0, 0, 1}, {0, 1, 1, 1}} // OneCell
```

To create the required swap 1-cell, we call

```
S22 = GetOneCellSwap[2, 2]
```

Now, we can generate the variable 2-cell

```
βFib = GenerateVariableTwoCell[mFib, mFib ∘ S22]
```

This gives the output

```
{{{{{a3}}, {-1, {0->0}}, {-1, {0->0}}, {{a4}}}, {{-1, {0->0}}, {{a5}}, {{a6}},
    {{a7}}}}, {{{{1, 0, 0, 1}, {0, 1, 1, 1}}, {4->2}} ->
    {{{1, 0, 0, 1}, {0, 1, 1, 1}}, {4->2}}}}}
```

We can look at this in a more reasonably form by entering

```
TwoCellForm[βFib]
```

Now we can set the variables $a3$ through $a7$ to their appropriate values. A quick comparison with Equation 4 shows that we need to enter

```
a3 = 1
a4 = Exp[-4*I*Pi/5]
a5 = 1
a6 = 1
a7 = Exp[3*I*Pi/5]
```

We can now confirm that what we have is correct by displaying the result

```
TwoCellForm[βFib]
```

This method of generating variable 2-cells (prior to setting the variables) can also be used to display the various equations that the 2-cells may satisfy and (in the cases where they are small enough) solve them.

3.6 Loading and Using a Category with *MTCategory.m*

The sub-project *MTCategory.m* allows you to load and work in a single specific MTC. However, the MTC must first be loaded. This can be done by individually specifying the structures and then passing them to the **SetMTC** function. Additionally, two MTCs have been preloaded, **Fib** and **Ising**, and can be set by the commands **SetMTCFib** and **SetMTCIsing**, respectively.

First, let us specify that we will be working in *MTCategory.m.* We assume that you have already added the package files to *Mathematica*'s *path* as explained in Section 3.1. As per the Section 3.2, we enter

```
Needs["MTCategory`"]
```

to load the package.

Now, if we want to work with **Fib**, we enter

```
SetMTCFib[]
```

and we are done. If we want to working with **Ising**, we enter

```
SetMTCIsing[]
```

and we are done. This command will output all the different structures that are defined, except the nontrivial anyons. Due to the way in which the structures are stored, the preloaded **SetMTC** commands *do not* load the nontrivial anyon(s), although it does tell you what they are. Instead, you can very easily set them yourself, see the tutorial in Section 5.1.1 for details. Now, we can check to make sure the loaded category is an MTC with the command

```
IsMTC[]
```

After a short time (while *Mathematica* is checking all the axioms), it will print out *True*.

If you want to work with a different MTC (or even see if a proposed set of structures satisfies any/all of the MTC axioms), you can specify your own structures. To set an MTC, you only need to input the 1-cells: fusion rules and the unit. This is enough to determine the form of the other structures, i.e. the 2-cells. Thus, **SetMTC** takes two arguments, the 1-cell fusion rules and the 1-cell unit.

Let us pretend that **Fib** was not preloaded. First, we shall enter the fusion rules and unit.

```
fuse = {{1,0,0,1},{0,1,1,1}} // OneCell
unit = {{1},{0}} // OneCell
```

Now, we can use set the form of the MTC

```
SetMTC[fuse, unit]
```

which prints out the 2-cell structures of the MTC and instructions for how to finish loading it.

While the structures are still variable, we can look at the equations that must be satisfied for the various axioms to hold and attempt to solve them (for instance, if searching for new solutions). For instance, type

```
GetPentagonEquation[]
```

to see in a nicely displayed form the system of pentagon equations for **Fib**, whose solution yields the **Fib** associator. To get them in a form that can be fed to a solver, use

```
GetPentagonEquation[False]
```

where the argument sets the *DisplayForm* variable to *False*, allow the output to be handled by *Mathematica*. Each and every axiom can be printed or checked. Please see Section 6.3 for more details about these (and similar) functions.

To set everything to the correct values for **Fib**, we enter (using semicolons to suppress the output)

```
a3 = 1;
a4 = 1;
a5 = 1;
a6 = 1;
a7 = 1;
a8 = 1;
a9 = 1;
a10 = 1;
a11 = 1;
a12 = 1;
a13 = 1;
a14 = GoldenRatio^-1;
a15 = GoldenRatio^(-1/2);
a16 = GoldenRatio^(-1/2);
a17 = -GoldenRatio^-1;
```

which completes the associator,

```
a22 = 1;
a23 = Exp[-4*I*Pi/5];
a24 = 1;
a25 = 1;
a26 = Exp[3*I*Pi/5];
```

which completes the braid,

```
a27 = 1;
a28 = Exp[4*I*Pi/5];
```

which completes the twist,

```
a18 = 1;
a19 = 1;
a20 = 1;
a21 = 1;
```

which completes the left and right unitors.[5] After all these values are set, run

```
IsMTC[]
```

to ensure that you entered everything correctly (or to test if your new structures satisfy all the axioms). In fact, each axiom can be checked individually. Please see Section 6.3 for a complete list. Furthermore, if we run

```
GetPentagonEquation[]
```

[5] N.B. the numbers are generated by *Mathematica* and may not always be the same (it creates an unused variable of the form *a#*, where # takes the next available value. If you run **SetMTC** again, the names will be different. The names used above are the names that generated for the first use in a session, assuming all those names are free.

now, we see that all the equations hold.

At any time, to view the loaded category, you can run

```
PrintMTC[]
```

Finally, if you want to load a structure directly, you can do that with a **Set** command. Let us do something silly. Get the form of the associator with the command

```
Getα[]
```

This will return the 2-cell α. Now, let us change the first entry to a 2 and set it back.

```
αbad = {{{{{1}}, {-1, {0 -> 0}}, {-1, {0 -> 0}}, {{1}}, {-1, {0 -> 0}}, {{1}},
    {{1}}, {{1}}}}, {{-1, {0 -> 0}}, {{1}}, {{1}}, {{1}}, {{1}}, {{1}}, {{1}},
    {{1/GoldenRatio, 1/Sqrt[GoldenRatio]}, {1/Sqrt[ GoldenRatio],
    -(1/GoldenRatio)}}}}, {{{{1, 0, 0, 1, 0, 1, 1, 1},
    {0, 1, 1, 1, 1, 1, 1, 2}}, {8 -> 2}} -> {{{1, 0, 0, 1, 0, 1, 1, 1},
    {0, 1, 1, 1, 1, 1, 1, 2}}, {8 -> 2}}}}
Setα[αbad]
```

Now, type

```
PrintMTC[]
```

and you can see that the bad α has been loaded.

Now, try

```
IsMTC[]
```

and you see that *Mathematica* (correctly) returns *False*.

Finally, as you might have guessed, you can use **Get** commands to get any of the loaded structures. For example,

```
Getm[]
Getu[]
Getα[]
Getβ[]
Getϑ[]
Getρ[]
Getλ[]
```

returns the 1-cell fusion rules, the 1-cell unit, the 2-cell associator, the 2-cell braid, the 2-cell twist, the 2-cell right unitor, and the 2-cell left unitor, respectively.

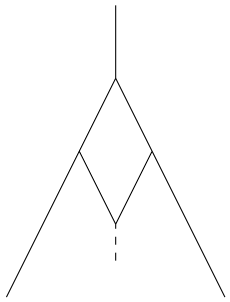


Figure 5: string diagram for a nontrivial process that takes $\tau \otimes \tau$ to τ

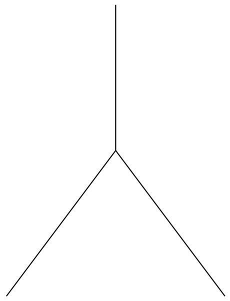


Figure 6: string diagram for the simplest process that takes $\tau \otimes \tau$ to τ

4 Tutorials

4.1 String Diagram Amplitudes

In this tutorial, we will calculate the amplitude of a "string" diagram, demonstrating how it can be done using the highest level *Mathematica* package.[6]

The specific string diagram that we will calculate will be within the modular tensor category **Fib** and is shown in Figure 5. **Fib** consists of one nontrivial particle species, the τ anyon. Figure 5 shows the fusion of two τ anyons (i.e. $\tau \otimes \tau$) into a τ anyon by way of some other processes. We want to compute the amplitude of this process compared to the process where the two τ's simply fuse into a τ anyon, which is shown in Figure 6. We can imagine needed to compute the amplitude for the nontrivial process due to some sort of topological obstruction (i.e. it takes place in a *handle* in the source 3-dimensional topological field theory) that prevents us from simplifying it to Figure 6. N.B. In this diagram (and all following string diagrams), time runs up from the bottom, the solid line represents τ anyons, and the dashed line represents the trivial u anyon (which is the same as the vacuum).

In standard arrow form, we can represent Figure 5 as

[6] By this, I mean "string" in the monoidal categorical sense, not the string theory sense.

$$\begin{tikzcd}
 & (\tau\otimes u)\otimes\tau \arrow[r, "(id_\tau\otimes\eta_\tau)\otimes id_\tau"] & (\tau\otimes(\tau\otimes\tau))\otimes\tau \arrow[ddr, "\alpha^{-1}_{\tau,\tau,\tau}\otimes id_\tau"] & \\
(\tau\otimes\tau) \arrow[ur, "\rho_\tau^{-1}\otimes id_\tau"] & & & \\
 & & & ((\tau\otimes\tau)\otimes\tau)\otimes\tau \arrow[ddl, "\alpha_{\tau\otimes\tau,\tau,\tau}"] \\
\tau & & & \\
 & \tau\otimes\tau \arrow[ul, "p_\tau"] & (\tau\otimes\tau)\otimes(\tau\otimes\tau) \arrow[l, "p_\tau\otimes p_\tau"] &
\end{tikzcd}$$

where ρ is the right unitor, α is the associator, η_A is a map from u to $A\otimes A^*$ (and since $\tau^*=\tau$, η_τ is a map from the vacuum to $\tau\otimes\tau$), and p_τ is the τ projector.[7] We are trying to calculate the composition of these six arrows, which is the amplitude for the process.

As stated before, this package contains three different sub-packages, each operating at a different level (roughly low-level, mid-level, and high-level), where the level refers to how close the user is to the underlying 2-category, **2Vect**, and simplicity of the coding. In our discussion of this tutorial so far, we have only operated at the 1-category level. Both the arrow diagrams and string diagrams are standard representations for MTCs at the l-category level.

4.1.1 Using MTCategory.m

To work within a single MTC at the 1-category level, we use the sub-package *MTCategory.m*. We assume that you have already added the package files to *Mathematica*'s *path* as explain in Section 3.1. Note: all *three* package files must be added as *MTCategory.m* depends on the other two.

To start, we must specify that we want to use the functions in *MTCategory.m*. As per the Section 3.2, we enter

```
Needs["MTCategory‘"]
```

to load the package. Now, we must "load" the MTC that we intend to work with. Since **Fib** is one of two preloaded MTCs, this is done by the command

```
SetMTCFib[]
```

This command will load all the structures of **Fib** (and prints them out in a nicely displayed form). However, due to the way in which the structures are stored, the preloaded **SetMTC** commands *do not* load the nontrivial anyon(s), although it does tell you what they are. Thus, next we need to load τ as a 1-cell (alright, so we cannot entirely escape the 2-category formalism). This is done by the command

```
τ= {{0},{1}} // OneCell
```

[7] Since the $\tau\otimes\tau$ fusion rule for **Fib** is $\tau\otimes\tau\to u\oplus\tau$, there is always projection map that takes you from $\tau\otimes\tau$ to τ. The arrow representation of Figure 6 is simply $\tau\otimes\tau \xrightarrow{p_\tau} \tau$.

i.e. we take the matrix $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ and convert it into a 1-cell. To confirm that this is correct, we can display it in 1-cell form by

```
τ // OneCellForm
```

and here we can easily see that this output is matched to the output for τ from the function **SetMTCFib**[].

Finally, before we begin calculating the amplitude, we need to load two 2-cells that also are not preloaded. Remember, 1-cells correspond to the objects of the MTC and 2-cells correspond to the morphisms. So we need to create two of the morphisms from the arrow diagram we are trying to compute. Luckily, they are very simple. From the MTC point of view, we need to create the morphisms $p_\tau : \tau \otimes \tau \to \tau$ and $\eta_\tau : u \to \tau \otimes \tau$. From the 2-category point of view, these are given by

$$p_\tau = \begin{bmatrix} \emptyset_{0\times 1} \\ \\ [1] \end{bmatrix}$$

$$\eta_\tau = \begin{bmatrix} [1] \\ \\ \emptyset_{1\times 0} \end{bmatrix}$$

These can be inputted directly a 2-matrices and then converted to 2-cells via

```
pτ = {{{-1,{1->0}}},{{{1}}}} // TwoCell
ητ = {{{{1}}},{{-1,{0->1}}}} // TwoCell
```

However, they can also be inputted using helper functions to make the task less daunting (one only needs to know how to input 1-cells, which is really just inputting matrices). In this case, you would enter

```
pτ = GenerateVariableTwoCell[τ⊗τ, τ]
ητ = GenerateVariableTwoCell[Getu[], τ⊗τ]
```

where the function **Getu**[] returns the 1-cell unit for **Fib** (which is pre-loaded). These two commands return, respectively,

```
{{{{-1, {1->0}}}, {{{a3}}}}, {{{{1}, {1}}, {1->2}} -> {{{0}, {1}}, {1->2}}}}
{{{{{a4}}}, {{-1, {0->1}}}}, {{{{1}, {0}}, {1->2}} -> {{{1}, {1}}, {1->2}}}}
```

and thus we simply set

```
a3 = 1
a4 = 1
```

to complete the entry of the 2-cells.[8] Either way, please consult Section 4.5 for a full guide on inputting 2-cells.

[8] The amplitudes are set to 1 because: for η_τ, this is the morphism that satisfies the snake equation, and for p_τ, we want to calculate the amplitude of the string diagram in 5 compared to just this morphism—for simplicity we set it to 1.

Now that all the 1-cells and 2-cells are loaded, we can calculate the amplitude. With this sub-package, calculating morphisms is as simple as reading them off from the arrow labels and then composing them. The six morphisms (which, from the 2-category perspective are 2-cells) are:

```
a = inv[ρ[τ]]⊗id[τ];
b = (id[τ]⊗ητ)⊗id[τ];
c = inv[α[τ,τ,τ]]⊗id[τ];
d = α[τ⊗τ,τ,τ];
e = pτ⊗pτ;
f = pτ;
```

cf. the six arrows in the diagram above. The *Mathematica* code is essentially identical to the labels on the arrows, with the exceptions being the need to use the **inv** function (that computes the inverse of a 2-cell) and the need to treat subscripts as function arguments. Everything is built in (once an appropriate MTC is loaded with **SetMTC** or **SetMTCFib**) except for τ, $p\tau$, and $\eta\tau$, which we loaded before.

To find the result, we need to compose the morphisms. This can be done in two different ways. First, we can compose them using $\cdot$ operator, just as you would write out the composition. Using the convention that the earlier ones go to the right, we have

```
f·e·d·c·b·a // FullSimplify
```

where the **FullSimplify** is used to help simplify the result (and parenthesis are unnecessary since the composition is associative). Alternatively, you can make a list of the morphisms (in this case, with the earlier ones to the left) and use the **Compose2Cells** command

```
list = {a, b, c, d, e, f};
Compose2Cells[list] // FullSimplify
```

Of course, you don't need to make use of the temporary variables *a*, *b*, *c*, *d*, *e*, or *f*, instead, to calculate the amplitude you could have simply written,

```
(inv[ρ[τ]]⊗id[τ]) · ((id[τ]⊗ητ)⊗id[τ]) · (inv[α[τ,τ,τ]]⊗id[τ]) · α[τ⊗τ,τ,τ] ·
   ( pτ⊗pτ · pτ)) // FullSimplify
```

making sure to put parenthesis around each 2-cell to be composed, since *Mathematica* does not know to do $\otimes$ before $\cdot$. Either and any way, the result will be the same. The output will be a 2-cell. Use the **TwoCellForm** command to display it in a more meaningful form. The result should look like

$$\begin{bmatrix} \emptyset_{0\times 1} \\ \\ \begin{bmatrix} -\frac{\sqrt{\varphi}}{1+\varphi} \end{bmatrix} \end{bmatrix}$$

The top cell gives the amplitude for the output to be a u (i.e. the final fusion product). But we explicitly made the output a τ with our final morphism, $p\tau$. Thus, it is a 0-dimensional matrix showing that this does not happen. The bottom cell gives the amplitude for the output to be a τ. For some reason, *Mathematica* will not simplify this further, but since $1+\varphi=\varphi^2$, this is equivalent to $-\varphi^{-\frac{3}{2}}$, which is the correct answer. Thus, we find that there is a nontrivial amplitude for this process.

4.2 Solving New Equations

The sub-packages *MTCategories.m* and *MTCategory.m* are designed to work with many or one MTC. In order to work with other 2-vector spaces that are not modular tensor categories, we must work solely with *TwoVect.m*—i.e the low-level 2-linear algebra operations and the 2-cell structural isomorphisms discussed in Section 2.2.

However, for this example, we will show you how to implement an equation that the package already implements, the pentagon equation of monoidal categories. From this, it should be clear how to generalize the method to any equation of interest within 2-vector spaces.

This entire tutorial is demonstrated in the notebook *New Equation (Pentagon) Tutorial.m.*

The pentagon equation, shown as equation 5, is an axiom of monoidal categories specifying that the two different ways of rebracketing four objects must be equal. By *Mac Lane's coherence theorem* [6], this will ensure that all the different ways of rebracketing objects for $n > 4$ will also be equal.

$$\begin{tikzcd}
 & ((A \otimes B) \otimes C) \otimes D \arrow[dl, "\alpha_{A,B,C} \otimes id_D"'] \arrow[dr, "\alpha_{A\otimes B,C,D}"] & \\
(A \otimes (B \otimes C)) \otimes D \arrow[d, "\alpha_{A,B\otimes C,D}"'] & & (A \otimes B) \otimes (C \otimes D) \arrow[d, "\alpha_{A,B,C\otimes D}"] \\
A \otimes ((B \otimes C) \otimes D) \arrow[rr, "id_A \otimes \alpha_{B,C,D}"'] & & A \otimes (B \otimes (C \otimes D))
\end{tikzcd} \tag{5}$$

4.2.1 Entering the Equation

To start, we must specify that we want to use the functions in *TwoVect.m.* As per the Section 3.2, we enter

```
Needs["TwoVect‘"]
```

to load the package. Remember, don't forget to first load the path as described in Section 3.1.

Let us start by defining our function:

```
PentagonEquation[m_, α_] :=  Module[{longpath, shortpath, n, nmn, mnn},
(*the rest of the code will go here*)
]
```

We define a function **PentagonEquation**, which is a function of the 1-cell product functor, m, and the 2-cell associator α. This function will either confirm that a given α solves the pentagon equation, or will generate the equations that a variable α must solve. The function **Module** tells *Mathematica* to create a list of variables whose scope is only within the brackets [ ]. Thus, we are creating local variables *longpath*, *shortpath*, *n*, *nmn*, and *mnn*, which we will soon see correspond to the two paths through the pentagon equation (one has three arrows, the other has two arrows, since it is a pentagon) and also various short notation for commonly used expressions.

Next, we will define these shortcut variables:

```
n = GetDest[m];
```

Since m is a 1-cell, its destination will be a 0-cell. By definition, n will be a number corresponding to the base number of objects in the category of interest—i.e. the n in the n-dimensional 2-vector space. For **Fib**, this is 2.

Next, let us define the next two variables, *nmn* and *mnn*.

```
nmn = (n⊙m)⊙n;
mnn = (m⊙n)⊙n;
```

It may be hard to see, but these are 1-cells. The operation $\odot$ computes the n-cell tensor-product. For 0-cells, this is simple multiplication. For 1-cells, this is the matrix tensor-product. For 2-cells, this is the 2-matrix tensor product. If one of the arguments is a different cell type than the other, the one that is lesser is automatically promoted to an $(n+1)$-cell by the operation **id**[*arg*], which takes either a 0-cell or a 1-cell and returns the identity 1-cell or 2-cell, respectively. In fact, there are many different ways of writing the above expressions. For instance,

```
nm = n⊙m;
nm = id[n]⊙m;
nm = GetIdentityOneCell[n]⊙m;
nm = OneCellTimes[id[n],m];
nm = OneCellTimes[GetIdentityOneCell[n],m];
```

all do the same thing. However,

```
nm = OneCellTimes[n,m];
```

will fail because **OneCellTimes** is a more primitive function and doesn't automatically promote n-cells to $(n+1)$-cells. For brevity and the fact that it works regardless of cell type, the first method is usually preferred. However, you must be careful to realize what your actual cell type will be. When in doubt, test the expression in a scrap notebook. Finally, since all the tensor products are associative, it does not matter how you bracket when using $\odot$, and in fact, $\odot$ is a n-ary operation.

Now, we are ready to begin coding the equation.

The first step, when presented with any such 1-category equation, is to convert it to a 2-category equation using diagrammatic notation. This will elucidate the required reboxing moves and thus indicate the proper expressions to enter into the *Mathematica* function.

First, for all the new structures, we need to establish the source and destination type—i.e. the boxing of the source and destination 1-cells—so we won't have a type mismatch when applying them. In this case, the new structure is α, and we will establish its type by boxing the source and destination as shown in Figure 7. From the perspective of 1-categories, α is a natural transformation with components $\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$. However, in **2Vect**, natural transformations are 2-cells, and the tensor product functor for the monoidal category inside **2Vect** (which was just shown as $\otimes$) is a 1-cell of **2Vect** that we call m. Thus, we represent $(A \otimes B) \otimes C$ as a diagram in **2Vect** with stacked m's, and we represent $A \otimes (B \otimes C)$ as a diagram in **2Vect** with stacked m's in a different way, to show their different bracketing. Additionally, we must *box* these

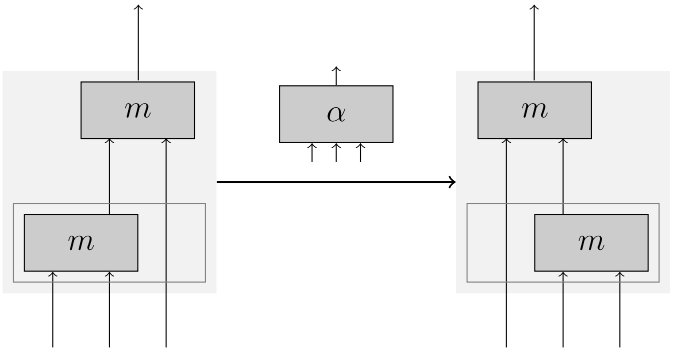


Figure 7: α

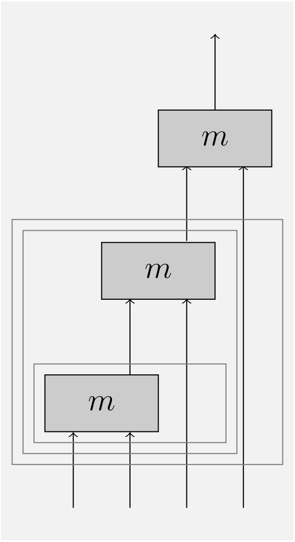


Figure 8: The start state to the pentagon equation, $((A \otimes B) \otimes C) \otimes D$.

source and destination diagrams. Since we are *defining* α, we can pick any boxing—i.e. since it's a definition we get to define its type. Boxing is done by picking two elements of the diagram at a time (that are either next to each other or follow each other) and putting a box around them.[9] Once the type is set, it must be used consistently through the same equation. Thus, in order to apply a 2-cell, such as α, it may be necessary to rearrange the boxing into the appropriate form to get the correct type. This can only be done by the structural isomorphisms, and this process of diagramming the equation will force us to insert all the necessary structures.

Once we have diagrams for all our new structures, we start by drawing the 1-cell start state, $((A \otimes B) \otimes C) \otimes D$, as a diagram in **2Vect**. This is done using the same method above to draw $A \otimes (B \otimes C)$, including the boxing. Again, the start state may be arbitrarily boxed (as long as the rules for boxing are followed) from the beginning as long as it is kept consistent. However, let us box in the way shown in Figure 8.

[9] N.B. Lines are implicitly the identity 1-cell. Thus, an arbitrary number of boxes may be drawn and removed from lines, since they may represent one or more identities.

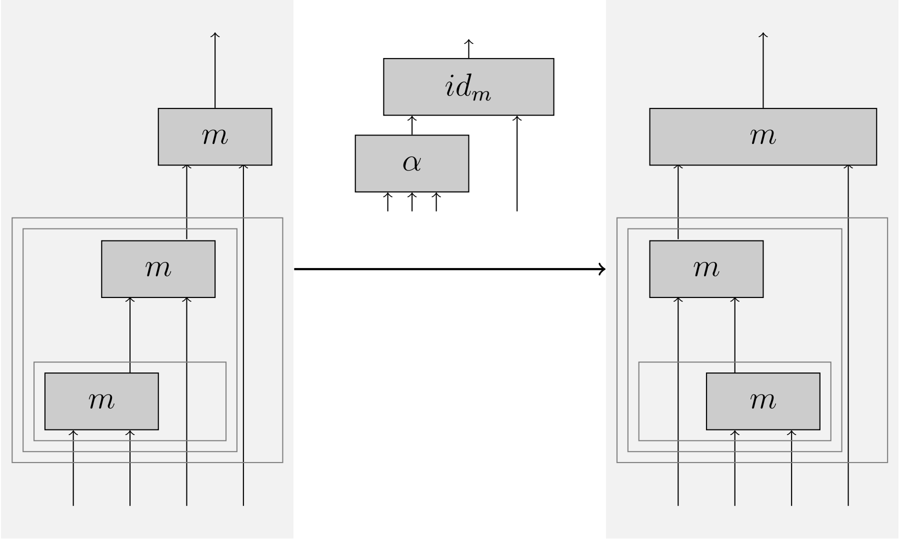


Figure 9: pentagon equation, path 1, step 1: $\alpha_{A,B,C} \otimes id_D$

Now, since both paths through the pentagon must be equal, we will have a string of diagrams for each path.

We we start with the longer three-legged path. Thanks to our fortuitous initial boxing, the diagram is in the right form for $\alpha_{A,B,C} \otimes id_D$ (as it is written from the 1-category perspective) to be applied—we can see embedded in the lower left of the diagram the correct type for the source of α as we boxed it in Figure 7. Thus, we apply α to that part and apply the 2-cell identity to the m at the top. The result is shown in Figure 9. In the destination 1-cell, we see that α was applied in the lower left, and the rest of the diagram remained unchanged. The 2-cell we applied, $\alpha_{A,B,C} \otimes id_D$ from the view of the 1-category, is shown drawn above the arrow. From the **2Vect** perspective, this is $id_m \circ (\alpha \boxtimes id_{id_{\mathbf{n}}})$, where $\mathbf{n}$ is the base 0-cell for this category. In this, we correspond the $\alpha_{A,B,C}$ with the α, the id_D with the $id_{id_{\mathbf{n}}}$ (that is, the identity 2-cell taken from the identity 1-cell on the base 0-cell of the category), and the $\otimes$ with the id_m. Thus, we can begin to see that by writing this equation within **2Vect**, we will be able to test the equation for *all* simple objects (and thus all objects) of the MTC. We are no longer concerned with arbitrary objects A, B, C, and D, but instead will test for every combination by operating at the 2-cell level of **2Vect**.

The part of the diagram we enter into *Mathematica* is the middle diagram that sits over the arrow. Naïvely, we would write this as:

```
TwoCellHorizontalComposition[GetIdentityOneCell[m],
    TwoCellTimes[α, GetIdentityTwoCell[GetIdentityOneCell[n]];
```

However, this is cumbersome, due to the long names of the functions and due to the need to write expressions such as $id_{id_{\mathbf{n}}}$ and id_m. We can solve the former problem as with the $\odot$ operation. We define our functions to automatically promote n-cells to $(n+1)$-cells using **GetIdentityOneCell** or **GetIdentityTwoCell**, as appropriate. Thus, instead of id_m and $id_{id_{\mathbf{n}}}$, we may simply write m and n, and *Mathematica* is smart enough to figure out what we mean. Furthermore, we define $\circ$ to implement 2-cell horizontal composition (which, N.B., is certainly *not* associative, so pay careful attention to the boxing of the 1-cell part of the diagram). Additionally, we also define $\circ$ to be 1-cell composition (which

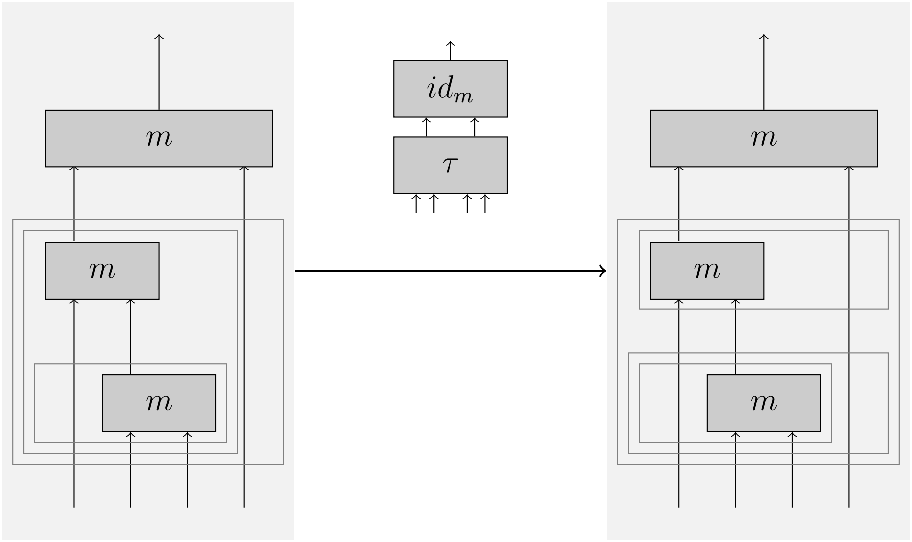


Figure 10: pentagon equation, path 1, step 2: τ reboxing

is very much related to 2-cell horizontal composition).[10] The operation is chosen based on the highest cell type in the arguments. Having said all that, instead of that long mess, we enter:

```
longpath = m ∘ (α⨀n);
```

Now, let us move on to the next leg on the path. We want to apply (according to the pentagon diagram) $\alpha_{A,B\otimes C,D}$—however, the type is not right. In fact, we need to make two reboxing moves. First, as we see the source types match, we need to apply τ. This is shown in 10. In order to apply τ, we implicitly used a box in the right-most line of the source 1-cell of the diagram to conform with the input source type for τ. This can be added since there is an implied boxing of the many identities that the line consists of (q.v. footnote 9).[11] The code for this line is:

```
longpath = longpath · (m ∘ Getτ[m, n⨀m, n, n]);
```

Here, we introduce a new function and a new operation. The $\cdot$ operation implements **TwoCell VerticalComposition**. This operation is associative, and can be used as an n-ary operation (which the primitive function **TwoCellVerticalComposition** cannot). Furthermore, the **Get**τ function returns the τ reboxing 2-cell. The order of the arguments is defined in Figure 2. Ultimately, the arguments ought to be 1-cells, but if a 0-cell is entered, it is automatically promoted to a 1-cell as described previously. This makes the notation and the coding very compact.

Now, we will apply ω to the outer two boxes, which will finally align the top part of the diagram so that the next step in the actual pentagon equation may be applied. This

[10] For 1-cells, this is also not n-ary. Even though $a \circ (b \circ c) = (a \circ b) \circ c$ for 1-cells in our representation, they are of different type, and it is essential to remember how they are bracketed.

[11] For the pentagon equation, all of the τ's will take at least two identities as arguments—rendering the τ equal to the identity 2-cell—and thus could be omitted. However, they have all been included in all the equations we will present, for completeness, and so the reader can gain an understanding of how they ought to be applied. In many cases they will not be identities.

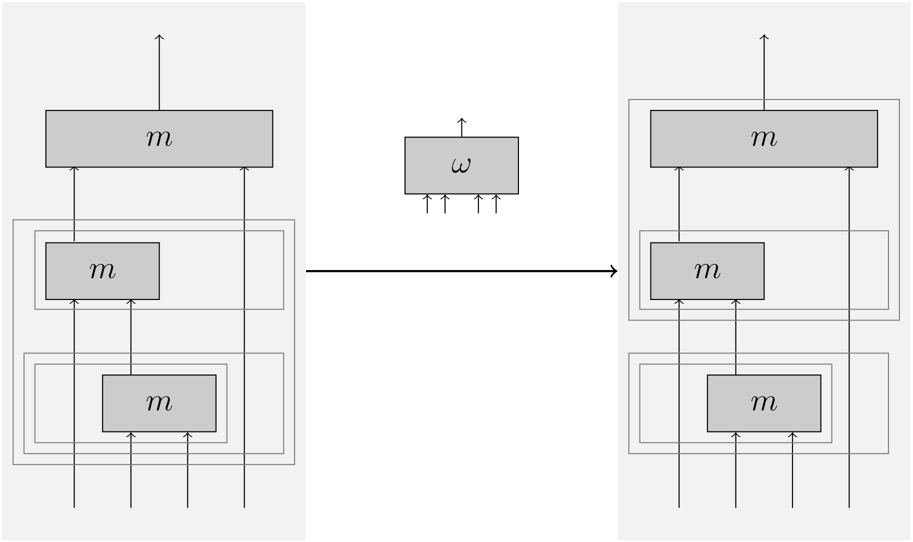


Figure 11: path 1, step 3: ω reboxing.

is shown in Figure 11, where we have used an implicit box around the top m in the source 1-cell—any single element can be considered boxed—to make the input consistent with the source 1-cell of ω. The code for this line is:

```
longpath = longpath · Getω[m, m⊙n, nmn];
```

Here, we introduce a new function **Get**ω, which returns the ω reboxing 2-cell. As with **Get**τ, the arguments ought to be 1-cells, but if they are 0-cells they are automatically promoted. The order of the arguments for **Get**ω is defined in Figure 3.

Now, we will apply the next actual step in the pentagon equation $\alpha_{A,B\otimes C,D}$. We see that the top of the 1-cell destination diagram from our last step is in the correct form for the application of α, and this corresponds to exactly the next step in the pentagon equation. The application of this step is shown in Figure 12. The code for this line is:

```
longpath = longpath · (α ∘ nmn);
```

Now, again, before we can apply the final leg of this path through the pentagon, we need to rebox—this time to effectively undo the reboxing from before. First we apply a ω^{-1}, Figure 13, and then we apply a τ^{-1}, Figure 14. However, notice the difference in the bottom row boxing from the destination 1-cell from Figure 13 and the source 1-cell from Figure 14, which, ostensible, should be the same picture. However, using the freedom given to us because the associator for the 2-monoidal structure of **2Vect**, ⊠, is the identity, we can associate parallel levels without applying any structural isomorphism 2-cell. Also, for clarity, we have omitted the box around the left-most line in the destination 1-cell of Figure 14.[12] The code for these two lines is:

```
longpath = longpath · inv[Getω[m, n⊙m, nmn]];
longpath = longpath · (m ∘ inv[Getτ[n, n, m, m⊙n]]);
```

[12] If this was not true (for instance, if we used a different 2-monoidal structure on **2Vect**), then this reboxing would come at the cost of applying another structural isomorphism 2-cell).

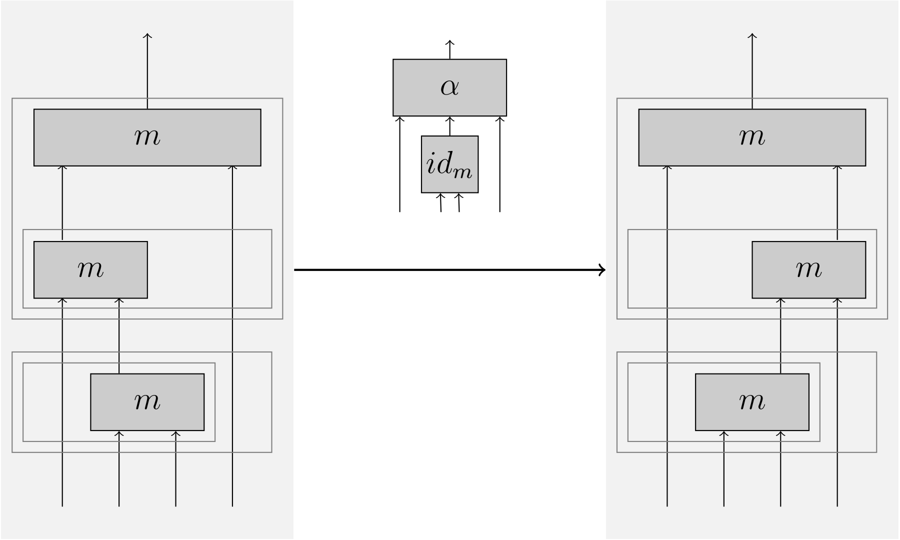


Figure 12: pentagon equation, path 1, step 3: $\alpha_{A,B\otimes C,D}$

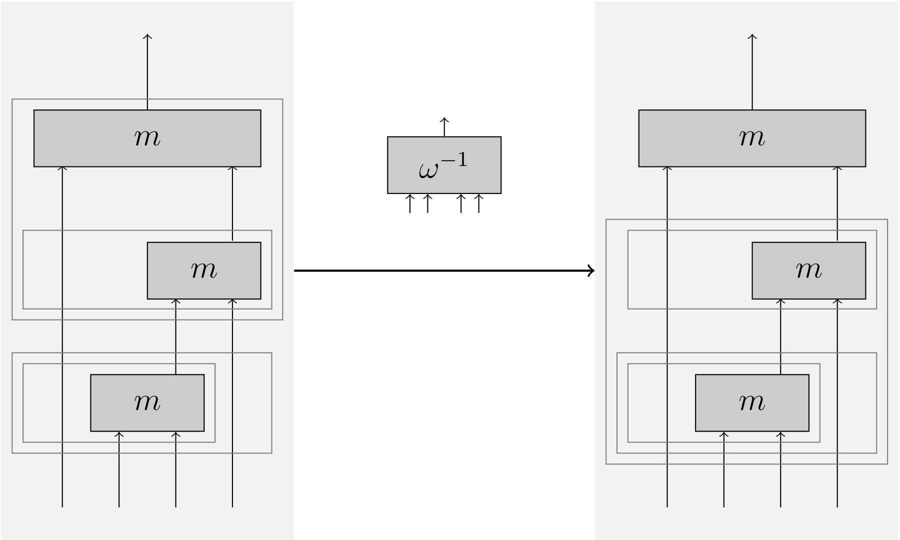


Figure 13: pentagon equation, path 1, step 4: ω^{-1} reboxing

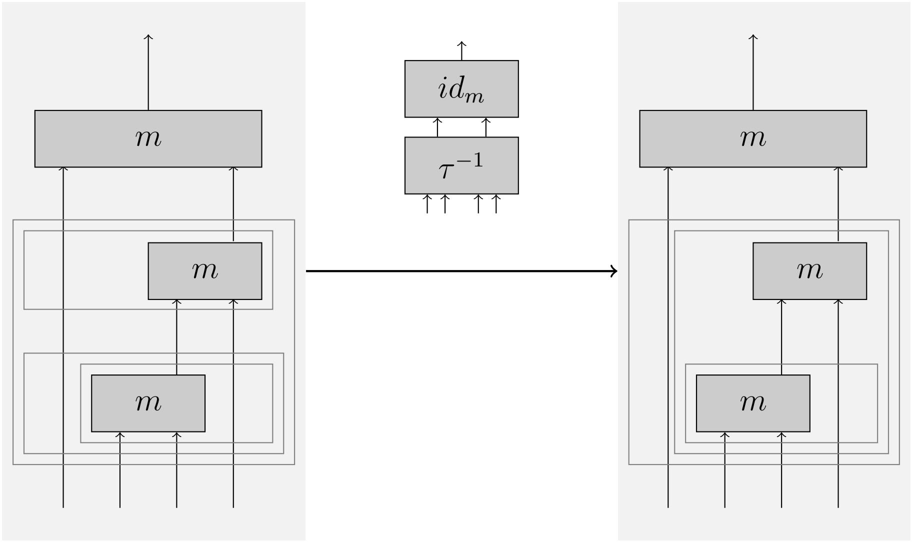


Figure 14: pentagon equation, path 1, step 5: τ^{-1} reboxing

The only new function here is the **inv** function, which takes either a 1-cell or a 2-cell and returns the inverse 1-cell or 2-cell, respectively. This can also be done using the *primitive* function, **TwoCellInverse**.

Finally, we are in exactly the correct form to apply the last leg of the pentagon, which is (from 1-category perspective) $id_A \otimes \alpha_{B,C,D}$. This is shown in Figure 15. The code for this line is:

```
longpath = longpath · (m ∘ (n⊙α));
```

The destination 1-cell of this Figure is required result; the diagram represents a particular boxing (although many are possible) of $A \otimes (B \otimes (C \otimes D))$.

Now, we must diagram and then program the shorter path of the pentagon equation. The short path through the pentagon proceeds in an identical fashion, with two caveats.

First, since we started with an arbitrary boxing of the first 1-cell state, $((A \otimes B) \otimes C) \otimes D$—which happened to be convenient for application of the first leg of the first path—we cannot simply apply the first leg of the new path. Almost always, some reboxing will be necessary. In this case, to apply $\alpha_{A\otimes B,C,D}$, we first need to apply τ, followed by ω.

Second, similar to the first caveat, just because the end state, $A \otimes (B \otimes (C \otimes D))$, is reached, does not mean that the process is complete. For the first path, we ended with an arbitrary boxing of that state. For the second path, most likely we will end with a different boxing. Thus, reboxing steps will need to be taken to ensure that the final diagrams for both paths match—i.e. so they are the same type and fit for comparison. Thus, the construction of the second path is not any different from the construction of the first path, except that it requires both reboxing at the beginning and at the end, which is necessitated by the boxing we chose to start and end with in the first path.

The code for the entire second path is:

```
shortpath = m ∘Getτ[m, m⊙n, n, n];
shortpath = shortpath · Getω[m, m⊙n, mnn];
shortpath = shortpath · (α ∘ mnn);
shortpath = shortpath · inv[Getω[m, n⊙m, mnn]];
```

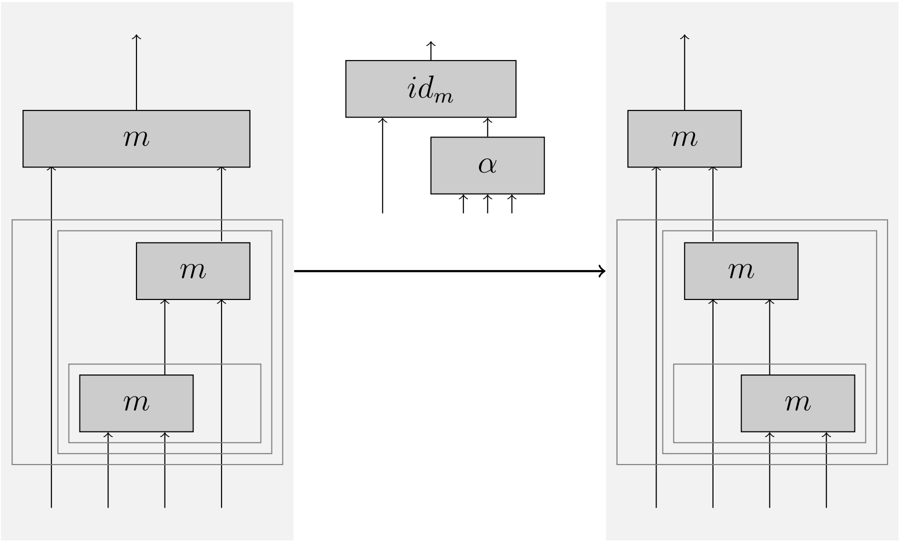


Figure 15: pentagon equation, path 1, step 6: $id_A \otimes \alpha_{B,C,D}$

```
shortpath = shortpath · (m ∘ inv[Getτ[n, m, m, n⊙n]]);
shortpath = shortpath · (m ∘ Getτ[m, n⊙n, n, m]);
shortpath = shortpath · Getω[m, m⊙n, (n⊙n)⊙m];
shortpath = shortpath · (α ∘ ((n⊙n)⊙m));
shortpath = shortpath · inv[Getω[m, n⊙m, (n⊙n)⊙m]];
shortpath = shortpath · (m ∘ inv[Getτ[n, n, m, n⊙m]]);
```

Ironically, while the second path was shorter from the 1-category perspective (and thus got the designation *shortpath*), it is in fact longer from the **2Vect** perspective due to our choice of boxing (we made it convenient to box the *longpath* requiring pre-reboxing and post-reboxing on the *shortpath*).

Overall, the code for the completed function is:

```
PentagonEquation[m_, α_] :=  Module[{longpath, shortpath, n, nmn, mnn},

    n = GetDest[m];
    nmn = (n⊙m)⊙n;
    mnn = (m⊙n)⊙n;

    longpath = m ∘ (α⊙n);
    longpath = longpath · (m ∘ Getτ[m, n⊙m, n, n]);
    longpath = longpath · Getω[m, m⊙n, nmn];
    longpath = longpath · (α ∘ nmn);
    longpath = longpath · inv[Getω[m, n⊙m, nmn]];
    longpath = longpath · (m ∘ inv[Getτ[n, n, m, m⊙n]]);
    longpath = longpath · (m ∘ (n⊙α));

    shortpath = m ∘Getτ[m, m⊙n, n, n];
    shortpath = shortpath · Getω[m, m⊙n, mnn];
    shortpath = shortpath · (α ∘ mnn);
    shortpath = shortpath · inv[Getω[m, n⊙m, mnn]];
```

```
        shortpath = shortpath · (m ∘ inv[Getτ[n, m, m, n⊙n]]);
        shortpath = shortpath · (m ∘ Getτ[m, n⊙n, n, m]);
        shortpath = shortpath · Getω[m, m⊙n, (n⊙n)⊙m];
        shortpath = shortpath · (α ∘ ((n⊙n)⊙m));
        shortpath = shortpath · inv[Getω[m, n⊙m, (n⊙n)⊙m]];
        shortpath = shortpath · (m ∘ inv[Getτ[n, n, m, n⊙m]]);

        Return[longpath == shortpath];

    ]
```

4.2.2 Solving the Equation

Let's try our new equation using the following very simple fusion rules:

```
m = OneCell[{{1, 0, 0, 0}, {0, 1, 1, 0}}];
```

In general, we need to create variable versions of the structures we are interested in finding. Thus, for each 2-cell structure we create some helper functions:

```
GenerateαSource[m_] := m ∘ (m ⊙ GetDest[m]);
GenerateαDest[m_] := m ∘ (GetDest[m] ⊙ m);
GenerateVariableα[m_] := GenerateVariableTwoCell[GenerateαSource[m],
     GenerateαDest[m]];
```

For m to be a valid product functor, we require both the source and the destination of our α to be the same in our representation (i.e. even though they are of different type and thus have different diagrams—in fact, Figure 7 is exactly the 2-cell that mediates between these diagrams/types/boxings). Thus, first we test this:

```
GenerateαSource[m] == GenerateαDest[m]
```

For our fusion rules, this is *True*. It is clear that this property is entirely a function of the fusion rules m. Checking for this is handled automatically when using the built-in function **IsPentagon**[m, α] from the sub-package *MTCategories.m*.

Now we generate and display our associator, which will have the correct form and structure, but variable entries:

```
(αTest = GenerateVariableα[m]) // TwoCellForm
```

The output of this expression should be:

$$\{\begin{pmatrix} (a3) & 0_{0\to0} & 0_{0\to0} & 0_{0\to0} & 0_{0\to0} & 0_{0\to0} & 0_{0\to0} & 0_{0\to0} \\ 0_{0\to0} & (a4) & (a5) & 0_{0\to0} & (a6) & 0_{0\to0} & 0_{0\to0} & 0_{0\to0} \end{pmatrix},$$

$$\left\{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}_{,(8\to2)}\right\}\to\left\{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}_{,(8\to2)}\right\}\}$$

which is our variable 2-cell associator. Now let's get the pentagon equation:

```
Equation = PentagonEquation[m, αTest]
```

The output is really ugly (mostly due to a lot of 0-dimensional matrices). [**JAMIE, I SORT OF HAVE HELPER FUNCTIONS THAT CAN MAKE THIS DISPLAY NICE —i.e. the machinery behind the DisplayForm optional variable—but that would involve modifying the PentagonEquation function that I just showed them how to write. Does this needlessly complicate things, or is it necessary?**]

Let's solve it:

```
Solve[Equation]
```

Great! We have a solution. Four, in fact:

```
{{a3->1, a5->0}, {a3->1, a5->1}, {a3->0, a4->0, a6->0}, {a3->1,a4->0, a6->0}}
```

However, if we are trying to form a monoidal category, we generally demand that α is invertible. So, in fact, not all of these solutions are valid. What we should have written was:

```
Solve[Equation && IsTwoCellInvertible[αTest]]
```

which ensures that our α is invertible. Now, we only have one solution:

```
{{a3->1, a5->1}}
```

In many cases the equations generated will be very complicated systems of equations, and *Mathematica* will not be able to simply solve them. Then, clever tricks on the part of you, the human, are required.

4.2.3 Additional Function Testing

Rather than using the very simple fusion rules in the last section, let's test our function using **Fib**.

First we load the sub-package that lets us work within a single category. Then, we set the current category to **Fib**.

```
Needs["MTCategory`"]
SetMTCFib[]
```

Mathematica prints out all of the structures of **Fib**, including a non-trivial associator. Now, let's use the loaded fusion rules and associator as inputs to our function:

```
PentagonEquation[Getm[], Getα[]]
```

The output is *True*, meaning the function we coded successfully implements the pentagon equation.

5 List of Functions by Package

The following is a list of all the functions in the three packages. To see information about any of these functions from within *Mathematica* please enter:

```
?FunctionName
```

where *FunctionName* is the name of the function you want to display info about.

5.1 TwoVect.m

5.1.1 Matrix and 2-Matrix Functions

Dim[x] gives the dimensions of the matrix x and can handle 0D matrices.

Is0D[A] determine whether the matrix (not a 1-cell or a 2-cell) A is in 0D form.

TensorProduct[x, y] gives the tensor product of matrices x and y. 0D matrices should be entered in the form $\{-1, \{n \to m\}\}$, where at least one of m or n must be a 0. $1 \times n$ dimensional matrices must be entered in the form $\{\{a, b, c, \ldots\}\}$, and $n \times 1$ dimensional matrices must be entered in the form $\{\{a\}, \{b\}, \{c\}, \ldots\}$.

DirectSum[x, y] gives the direct sum of matrices x and y. 0D matrices should be entered in the form $\{-1, \{n \to m\}\}$, where at least one of m or n must be a 0. $1 \times n$ dimensional matrices must be entered in the form $\{\{a, b, c, \ldots\}\}$, and $n \times 1$ dimensional matrices must be entered in the form $\{\{a\}, \{b\}, \{c\}, \ldots\}$.

Dot0D[A, B] gives a matrix dot product, AB, that works with 0D matrices.

IdentityMat[n] generates the $n \times n$ identity matrix and can handle $n = 0$.

MatrixInverse[A] if it exists, computes A^{-1}, the inverse of the matrix A, and can handle 0D matrices."

IsMatrixInvertible[A] returns *True* if the matrix A is invertible and *False* if it is not. Only square matrices are candidates to be invertible. Handles 0D matrices.

GetAdjointMatrix[A] returns $A^{\dagger}$, the adjoint (conjugate transpose) of the input matrix, A. Handles 0D matrices.

GenerateVariableMatrix[*InputDim*] generates a matrix (with dimensions specified by the list
InputDim) filled entirely with new unique variables (i.e. to generate an $m \times n$ dimensional matrix with variable entries, $InputDim = \{m, n\}$). Will also work for $m = 0$ and/or $n = 0$.

CellByCellMatrixEquation[A, B] takes two matrices to be equated and equates them cell by cell (i.e. it creates a matrix of $A[[i, j]] == B[[i, j]]$ rather than just $A == B$).

CellByCellTwoMatrixEquation[A, B] takes two 2-matrices to be equated and equates them
element-matrix cell by cell (i.e. it creates a 2-matrix of $A[[i, j]][[k, m]] == B[[i, j]][[k, m]]$ rather than just $A == B$).

5.1.2 Generic 2Vect Functions for 1-cells and 2-cells

GetSource$[A]$ returns the source n-Cell for either 2-cells or 1-cells.

GetDest$[A]$ returns the destination n-Cell for either 2-cells or 1-cells.

GetCellType$[A]$ returns 0 if A is a 0-Cell, 1 if A is a 1-cell, 2 if A is a 2-cell, and an error otherwise (although if something is neither it may not always catch the error).

5.1.3 1-cell Functions

OneCell$[A]$ takes a matrix A and converts it into a 1-cell.

OneToMat$[A]$ takes a 1-cell A and converts it into a matrix.

OneCellComposition$[f, g]$ composes the 1-cells f and g as $f \circ g$, with the convention of right to left (i.e. g followed by f).

OneCellTimes$[f, g]$ combines the 1-cells f and g in parallel as $f \boxtimes g$ using the tensor product 2-functor $\boxtimes$.

GetIdentityOneCell$[n]$ returns the identity 1-cell of n, the input 0-Cell.

OneCellInverse$[f]$ takes the input 1-cell f and returns the inverse 1-cell, if it exists.

IsOneCellInvertible$[f]$ returns *True* if the 1-cell f is invertible and *False* if it is not. Only 1-cells that go between the same 0-Cells (i.e. represented by square matrices) are candidates to be invertible.

GetAdjointOneCell$[A]$ returns the adjoint (conjugate transpose) of the input 1-cell.

GetOneCellSwap$[m, n]$ will return the 1-cell $S_{m,n}$. If 0-Cells m and n (i.e. 2 natural numbers) are combined using the tensor product 2-functor as $m \boxtimes n$, then $S_{m,n}$ is the swap 1-cell such that $S_{m,n}$ acting on $m \boxtimes n$ will give $n \boxtimes m$.

5.1.4 2-cell Functions

TwoCell$[A]$ takes a 2-matrix (i.e. a matrix of matrices) A and converts it into a 2-cell.

TwoToTwoMat$[A]$ takes a 2-cell A and converts it into a 2-matrix (i.e. a matrix of matrices).

TwoCellVerticalComposition$[F, G]$ composes the 2-cells F and G vertically as $F \circ G$, with the convention of right to left (i.e. G followed by F). The source of F must equal the destination of G.

$$\begin{tikzcd}
\mathbf{N} \arrow[r, bend left=70, "f"] \arrow[r, "g"] \arrow[r, bend right=70, "h"'] \arrow[r, phantom, bend left=35, "\Downarrow G"] \arrow[r, phantom, bend right=35, "\Downarrow F"] & \mathbf{M}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{N} \arrow[r, bend left, "f"] \arrow[r, bend right, "h"'] \arrow[r, phantom, "\Downarrow F \circ G"] & \mathbf{M}
\end{tikzcd}$$

TwoCellHorizontalComposition$[G, F]$ composes the 2-cells G and F horizontally as $G \bullet F$, with the convention of right to left (i.e. F followed by G). The destination 0-Cell of the source and destination 1-cells of F must be the same as the source 0-Cell of the source and destination 1-cells of G.

$$\begin{tikzcd}
\mathbf{N} \arrow[r, bend left, "f_1"] \arrow[r, bend right, "f_2"'] \arrow[r, phantom, "\Downarrow F"] & \mathbf{M} \arrow[r, bend left, "g_1"] \arrow[r, bend right, "g_2"'] \arrow[r, phantom, "\Downarrow G"] & \mathbf{P}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{N} \arrow[r, bend left, "g_1 \circ f_1"] \arrow[r, bend right, "g_2 \circ f_2"'] \arrow[r, phantom, "\Downarrow G \bullet F"] & \mathbf{P}
\end{tikzcd}$$

TwoCellTimes$[F, G]$ combines the 2-cells F and G in parallel as $F \boxtimes G$ using the tensor product 2-functor $\boxtimes$.

GetIdentityTwoCell$[f]$ returns the identity 2-cell (under vertical composition) of f, the input 1-cell.

TwoCellInverse$[F]$ takes the input 2-cell F and returns the inverse 2-cell (under vertical composition), if it exists.

IsTwoCellInvertible$[F]$ returns *True* if the 2-cell F is invertible (under vertical composition) and *False* if it is not. Only 2-cells whose element-matrices (i.e. inner matrices) are all square are candidates to be invertible.

GetAdjointTwoCell$[A]$ returns the adjoint (conjugate transpose) of the input 2-cell.

WhiskerRight$[f, G]$**,** for input 1-cell f and 2-cell G, horizontally combines the 2-cell G followed by the 2-cell given by **GetIdentityTwoCell**$[f]$ (i.e. the 2-cell $id_f \bullet G$). The order is a little unintuitive.

WhiskerLeft$[G, f]$**,** for input 2-cell G and 1-cell f, horizontally combines the 2-cell given by

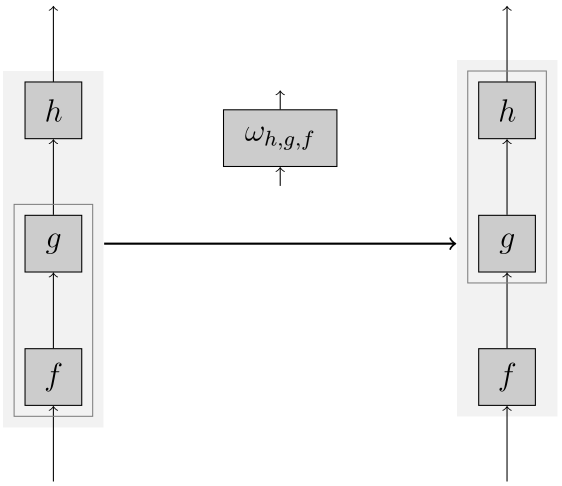
GetIdentityTwoCell$[f]$ followed by the 2-cell G (i.e. the 2-cell $G \bullet id_f$). The order is a little unintuitive.

GenerateVariableTwoCell$[TwoCellSource, TwoCellDest]$ returns a 2-cell whose element-matrix elements (i.e. the elements of the inner matrices) are filled entirely with new and unique variables. The inputs *TwoCellSource* and *TwoCellDest* are the 1-cell source and destination, respectively, of the new 2-cell to be generated.

5.1.5 2-cell Structural Isomorphisms

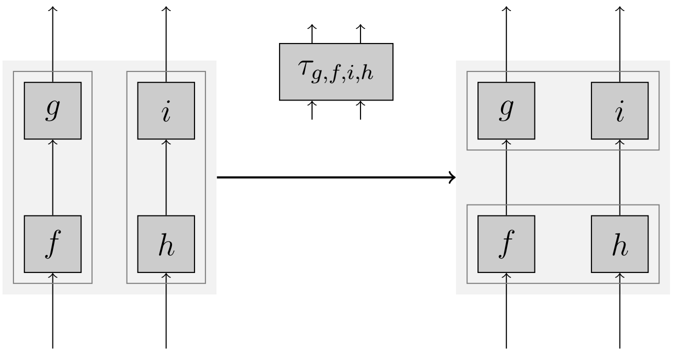
Getω$[a, b, c]$**,** with 1-cell inputs a, b, and c, is used to generate the structural isomorphism horizontal associator, ω, which is used to correct for horizontally composing three 2-cells in a row. If any of the arguments are 0-cells, they are automatically promoted to 1-cells using **GetIdentityOneCell**.
Getω[GetDest$[A]$**,GetDest**$[B]$**,GetDest**$[C]]\circ(A \bullet (B \bullet C))$
$== ((A \bullet B) \bullet C)\circ$**Getω[GetSource**$[A]$**,GetSource**$[B]$**,GetSource**$[C]]$.

$$\begin{tikzcd}
a_1 \circ (b_1 \circ c_1) \arrow[r, Rightarrow, "\omega_{a_1,b_1,c_1}"] \arrow[d, Rightarrow, "A\bullet(B\bullet C)"'] & (a_1 \circ b_1) \circ c_1 \arrow[d, Rightarrow, "(A\bullet B)\bullet C"] \\
a_2 \circ (b_2 \circ c_2) \arrow[r, Rightarrow, "\omega_{a_2,b_2,c_2}"'] & (a_2 \circ b_2) \circ c_2
\end{tikzcd} \tag{6}$$



Getτ$[a, b, c, d]$**,** with 1-cell inputs a, b, c, and d, is used to generate the structural isomorphism, τ, which is used to correct for the fact that horizontally composing two pairs of 2-cells and then monoidally combining them is not the same as monoidally combining the two pairs and then horizontally composing them. If any of the arguments are 0-cells, they are automatically promoted to 1-cells using **GetIdentityOneCell**.
Getτ[GetDest$[A]$**,GetDest**$[B]$**,GetDest**$[C]$**,GetDest**$[D]]\circ(A\bullet B)\boxtimes(C\bullet D) ==$
$(A\boxtimes C)\bullet(B\boxtimes D)\circ$**Getτ[GetSource**$[A]$**,GetSource**$[B]$**,GetSource**$[C]$**,GetSource**$[D]]$.

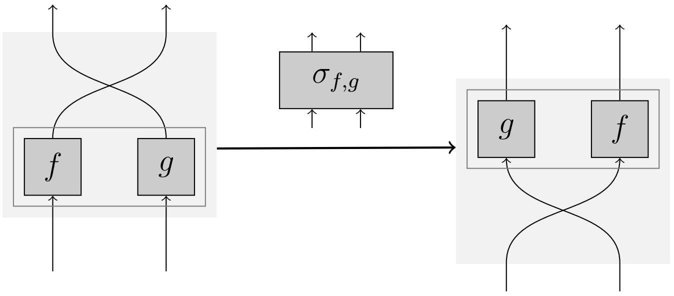
$$\begin{tikzcd}
(a_1 \circ b_1) \boxtimes (c_1 \circ d_1) \arrow[r, Rightarrow, "\tau_{a_1,b_1,c_1,d_1}"] \arrow[d, Rightarrow, "(A\bullet B)\boxtimes(C\bullet D)"'] & (a_1 \boxtimes c_1) \circ (b_1 \boxtimes d_1) \arrow[d, Rightarrow, "(A\boxtimes C)\bullet(B\boxtimes D)"] \\
(a_2 \circ b_2) \boxtimes (c_2 \circ d_2) \arrow[r, Rightarrow, "\tau_{a_2,b_2,c_2,d_2}"'] & (a_2 \boxtimes c_2) \circ (b_2 \boxtimes d_2)
\end{tikzcd} \tag{7}$$



Get$\sigma[A,B]$**,** with 1-cell inputs A and B, is used to generate the structural isomorphism, σ, a 2-cell swap (i.e. an interchange between a 1-cell swap and 1-cells on each leg of the swap). If any of the arguments are 0-cells, they are automatically promoted to 1-cells using **GetIdentityOneCell**.

Thus, at the 2-cell level, if $\mu : F_1 \rightarrow F_2$ and $\nu : G_1 \rightarrow G_2$ (with 1-cells $F_i : \mathbf{A} \rightarrow \mathbf{B}$ and $G_i : \mathbf{C} \rightarrow \mathbf{D}$), then **Get**$\sigma[F_2, G_2] \circ (id_{S_{B,D}} \bullet (\mu \boxtimes \nu)) == ((\nu \boxtimes \mu) \bullet id_{S_{A,C}}$**Get**$\sigma[F_1, G_1]$.

$$\begin{tikzcd}
S_{\mathbf{B},\mathbf{D}} \circ (F_1 \boxtimes G_1) \arrow[r, Rightarrow, "\sigma_{F_1,G_1}"] \arrow[d, Rightarrow, "id_{S_{\mathbf{B},\mathbf{D}}} \bullet (\mu \boxtimes \nu)"'] & (G_1 \boxtimes F_1) \circ S_{\mathbf{A},\mathbf{C}} \arrow[d, Rightarrow, "(\nu \boxtimes \mu) \bullet id_{S_{\mathbf{A},\mathbf{C}}}"] \\
S_{\mathbf{B},\mathbf{D}} \circ (F_2 \boxtimes G_2) \arrow[r, Rightarrow, "\sigma_{F_2,G_2}"'] & (G_2 \boxtimes F_2) \circ S_{\mathbf{A},\mathbf{C}}
\end{tikzcd} \tag{8}$$



5.1.6 Display Functions

TwoMatrixForm[F] display 2-matrices (i.e. matrices of matrices, or, 2-cells without their 1-cell source and destination) in a form equivalent to **MatrixForm**[F] for regular matrices.

OneCellForm[f] display 1-cells in a form equivalent to **MatrixForm**[F] for regular matrices.

TwoCellForm[F] display 2-cells in a form equivalent to **MatrixForm**[F] for regular matrices.

5.1.7 Overloaded Operations (that work for 1-cells and 2-cells)

A ∘ B or **SmallCircle**$[A, B]$ computes 1-cell composition or 2-cell horizontal composition, depending on the cell type of the arguments. This operation is *not* associative. The operation is chosen based on the highest cell type of the arguments, and the other argument is promoted to an $(n+1)$-cell by the operation **GetIdentityOneCell** or **GetIdentityTwoCell**.

A ⊙ B ⊙ ... or **CircleDot**$[A, B, \ldots]$ computes the n-cell tensor-product. For 0-cells, this is simple multiplication. For 1-cells, this is the matrix tensor-product. For 2-cells, this is the 2-matrix tensor product. If one of the arguments is a different cell type than the other, the one that is lesser is automatically promoted to an $(n+1)$-cell by the operation **GetIdentityOneCell** or **GetIdentityTwoCell**. This is an n-ary operation as the tensor product is associative.

A · B· ... or **CenterDot**$[A, B, \ldots]$ computes 2-cell vertical composition. This operation is associative, and can be used as an n-ary operation.

id$[A]$ returns either the identity 1-cell or the identity 2-cell, depending on whether the argument is a 0-Cell or 1-cell, respectively.

inv$[A]$ returns either the inverse 1-cell or the inverse 2-cell, depending on whether the argument is a 1-cell or 2-cell, respectively.

Compose2Cells$[list]$ vertically composes a list of 2-cells. The first element of the list is the first 2-cell applied, which is usually written as the right-most 2-cell (i.e. if A and B are 2-cells and you want to compute $A \circ B$, then you would enter $\{B, A\}$ as the argument to this function). This is an alternative to the many ways of vertically composing 2-cells.

5.2 MTCategories.m

Within this section, there are two optional arguments that appear in many of the functions.

DisplayForm is optional argument that controls how the output is displayed. If *True*, it will display the structures using *TwoCellForm* (as opposed to the standard form which can be used for input or solving).

WithErrors is an optional argument that controls whether errors are outputted. If *False*, the errors will be suppressed. This option is essentially intended to be used when the 2-cell arguments have variable entries and you are trying to find the equations that the axioms require (and perhaps send them to a solver). If *True*, the output will let the user know which axioms fail (if any).

5.2.1 Generating Functions

GenerateMonoidalStructures$[m, u, DisplayForm]$ generates a complete set of monoidal category 2-cell structures, α, ρ, and λ (with the correct 1-cell source and destination), for a given product functor 1-cell m and unit 1-cell u. N.B. *DisplayForm* is set to *False* by default.

GenerateMTCStructures$[m, u, DisplayForm]$ generates a complete set of modular tensor category 2-cell structures (including the full set of monoidal category structures),

α, ρ, λ, β, and θ (with the correct 1-cell source and destination), for a given product functor 1-cell m and unit 1-cell u. N.B. *DisplayForm* is set to *False* by default.

GenerateHandle$[m, u]$ generates the handle (which is a 1-cell object) of the MTC given by product functor m and unit u. The handle is the sum over all the simple objects of that simple object fused with its dual: $\sum A \otimes A^*$, for all simple objects A.

5.2.2 Monoidal Categories

IsPentagon$[m, \alpha, WithErrors, DisplayForm]$ is a function that calculates and tests the pentagon equation, an axiom of monoidal categories, where the 1-cell m specifies the fusion rules, and the 2-cell α specifies the associator. **IsPentagon**$[m, \alpha]$ tests the form and consistency of m and α, whether α is invertible, and the pentagon equation itself. The last two arguments are optional (and default to *True* and *True*, respectively).

$$\begin{tikzcd}
 & ((A \otimes B) \otimes C) \otimes D \arrow[dl, "\alpha_{A,B,C} \otimes id_D"'] \arrow[dr, "\alpha_{A \otimes B,C,D}"] & \\
(A \otimes (B \otimes C)) \otimes D \arrow[d, "\alpha_{A,B\otimes C,D}"'] & & (A \otimes B) \otimes (C \otimes D) \arrow[d, "\alpha_{A,B,C\otimes D}"] \\
A \otimes ((B \otimes C) \otimes D) \arrow[rr, "id_A \otimes \alpha_{B,C,D}"'] & & A \otimes (B \otimes (C \otimes D))
\end{tikzcd} \tag{9}$$

IsTriangle$[m, \alpha, \rho, \lambda, u, WithErrors, DisplayForm]$ is a function that calculates and tests the triangle equation, an axiom of monoidal categories, where the 1-cell m specifies the fusion rules, the 1-cell u specifies the unit, the 2-cell ρ specifies the right unitor, and the 2-cell λ specifies the left unitor, and the 2-cell α specifies the associator. **IsTriangle**$[m, \alpha, \rho, \lambda, u]$ tests the form and consistency of the arguments, whether ρ and λ are invertible, and the triangle equation itself. The last two arguments are optional (and default to *True* and *True*, respectively). Note: this function should only be called after the category is shown to satisfy the pentagon equation.

$$\begin{tikzcd}
(A \otimes u) \otimes B \arrow[rr, "\alpha_{A,u,B}"] \arrow[dr, "\rho_A \otimes id_B"'] & & A \otimes (u \otimes B) \arrow[dl, "id_A \otimes \lambda_B"] \\
 & A \otimes B &
\end{tikzcd} \tag{10}$$

IsMonoidal$[m, \alpha, \rho, \lambda, u]$ tests if the input structures form a monoidal category, where m is the 1-cell tensor product functor specifying the fusion rules, u is the 1-cell unit, α is the 2-cell associator, ρ is the 2-cell right unitor, and λ is the 2-cell left unitor.

5.2.3 Modular Tensor Categories

IsBraided$[m, \alpha, \beta]$ tests if a monoidal category is braided with the input 2-cell β. For this function to be used, m andα (and other structures, of course) must be shown already to form a monoidal category.

IsHexagon1$[m, \alpha, \beta, \mathit{DisplayForm}]$ is a function that calculates if a monoidal category satisfies the first hexagon equation for braided monoidal categories (an axiom of MTCs), where the 1-cell m specifies the fusion rules, the 2-cell α specifies the associator, and the 2-cell β specifies the braid. This function is essentially intended to be used when α and/or β have variable entries (as opposed to simply calling **IsBraided**$[m, \alpha, \beta]$. The last argument is optional (and defaults to *True*).

$$\begin{tikzcd}
 & A \otimes (B \otimes C) \arrow[r, "\beta_{A,B\otimes C}"] & (B \otimes C) \otimes A \arrow[dr, "\alpha_{B,C,A}"] & \\
(A \otimes B) \otimes C \arrow[ur, "\alpha_{A,B,C}"] \arrow[dr, "\beta_{A,B}\otimes id_C"'] & & & B \otimes (C \otimes A) \\
 & (B \otimes A) \otimes C \arrow[r, "\alpha_{B,A,C}"'] & B \otimes (A \otimes C) \arrow[ur, "id_B\otimes\beta_{A,C}"'] &
\end{tikzcd} \tag{11}$$

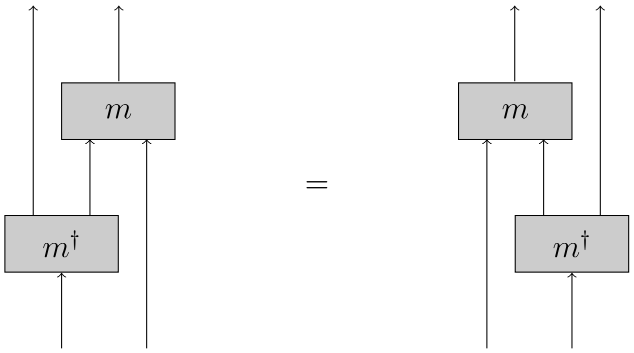
IsHexagon2$[m, \alpha, \beta, \mathit{DisplayForm}]$ is a function that calculates if a monoidal category satisfies the second hexagon equation for braided monoidal categories (an axiom of MTCs), where the 1-cell m specifies the fusion rules, the 2-cell α specifies the associator, and the 2-cell β specifies the braid. The last argument is optional (and defaults to *True*). This function is essentially intended to be used when β has variable entries (as opposed to simply calling **IsBraided**$[m, \alpha, \beta]$). However, it MUST NOT be used when α has variable entries since α^{-1} is required (since a variable α cannot be inverted).

$$\begin{tikzcd}
 & (A \otimes B) \otimes C \arrow[r, "\beta_{A\otimes B,C}"] & C \otimes (A \otimes B) \arrow[dr, "\alpha^{-1}_{C,A,B}"] & \\
A \otimes (B \otimes C) \arrow[ur, "\alpha^{-1}_{A,B,C}"] \arrow[dr, "id_A\otimes\beta_{B,C}"'] & & & (C \otimes A) \otimes B \\
 & A \otimes (C \otimes B) \arrow[r, "\alpha^{-1}_{A,C,B}"'] & (A \otimes C) \otimes B \arrow[ur, "\beta_{A,C}\otimes id_B"'] &
\end{tikzcd} \tag{12}$$

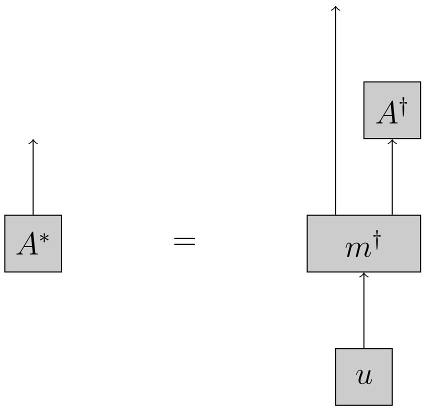
IsTwisted$[m, \beta, \theta, \mathit{WithErrors}, \mathit{DisplayForm}]$ is a function that calculates if a braided monoidal category has a twist (an axiom of MTCs), where the 1-cell m specifies the fusion rules, the 2-cell β specifies the braid, and the 2-cell θ specifies the twist. **IsTwisted**$[m, \beta, \theta]$ tests the form and consistency of the arguments, whether θ is invertible, and the twist equation itself. The last two arguments are optional (and default to *True* and *True*, respectively). Note: this function should only be called after m and β (with additional structures, of course) are shown form a braided monoidal category.

$$\begin{tikzcd}
A \otimes B \arrow[r, "\theta_{A\otimes B}"] \arrow[d, "\theta_A\otimes\theta_B"'] & A \otimes B \\
A \otimes B \arrow[r, "\beta_{A,B}"'] & B \otimes A \arrow[u, "\beta_{B,A}"']
\end{tikzcd} \tag{13}$$

IsFrobenius$[m]$ tests the Frobenius condition, which is necessary but not sufficient for confirming the rigidity property of MTCs.



GenerateDual$[x, m, u]$ returns the dual candidate of the 1-cell x (i.e. tentatively x^*). Note: this can only be called if **IsFrobenius**$[m]$ is True.



IsRigid$[m, \alpha, u]$ tests if a monoidal category satisfies the rigidity property (i.e. has duals). For this function to be used, m and α must form a monoidal category.

$$\begin{tikzcd}
A \arrow[rr, "id_A"] \arrow[d, "\eta_A \otimes id_A"'] & & A \\
(A \otimes A^*) \otimes A \arrow[rr, "\alpha_{A,A^*,A}"'] & & A \otimes (A^* \otimes A) \arrow[u, "id_A \otimes \varepsilon_A"']
\end{tikzcd} \tag{14}$$

$$\begin{tikzcd}
A^* \arrow[rr, "id_{A^*}"] \arrow[d, "id_{A^*} \otimes \eta_A"'] & & A^* \\
A^* \otimes (A \otimes A^*) \arrow[rr, "\alpha^{-1}_{A^*,A,A^*}"'] & & (A^* \otimes A) \otimes A^* \arrow[u, "\varepsilon_A \otimes id_{A^*}"']
\end{tikzcd} \tag{15}$$

IsRibbonCoherent$[m, \theta, u, DisplayForm]$ tests whether the 2-cell twist θ is compatible with duals. Essentially, this tests whether the twisted braided monoidal category is a ribbon category. The last argument is optional (and defaults to *False*). For this function to be used, m, θ, and u (with additional structures, of course) must form a twisted braided monoidal category, and **IsRigid**$[m, \alpha, u]$ must be *True*.

$$\begin{tikzcd}
u \arrow[r, "\eta_A"] & A \otimes A^* \arrow[r, bend left, "id_A \otimes \theta_{A^*}"] \arrow[r, bend right, "\theta_A \otimes id_{A^*}"'] & A \otimes A^*
\end{tikzcd} \tag{16}$$

IsModular$[m, u, \beta]$ tests if a braided twisted monoidal category that is rigid satisfies the modular property, where m is the 1-cell fusion rules, u is the 1-cell unit, and β is the 2-cell braid.

IsMTC$[m, \alpha, \rho, \lambda, u, \beta, \theta]$ tests if the input structures form a modular tensor category, where where m is the 1-cell product functor, α is the 2-cell associator, ρ is the 2-cell right unitor, λ is the 2-cell left unitor, u is the 1-cell unit, β is the 2-cell braid, and θ is the 2-cell twist.

5.2.4 Monoids (Algebras)

IsMonoidAssociative$[m, A, \mu, \alpha, DisplayForm]$ determines whether the 2-cell multiplication μ satisfies the monoid associativity equation for the 1-cell A to be a monoid. The last argument is optional (and defaults to *True*). For this function to be used, the 1-cell fusion rules m and the 2-cell associator α (with additional structures, of course) must form a monoidal category.

$$\begin{tikzcd}
& (A \otimes A) \otimes A \arrow[dl, "\mu \otimes id_A"'] \arrow[dr, "\alpha_{A,A,A}"] & \\
A \otimes A \arrow[d, "\mu"'] & & A \otimes (A \otimes A) \arrow[d, "id_A \otimes \mu"] \\
A & & A \otimes A \arrow[ll, "\mu"]
\end{tikzcd} \tag{17}$$

IsMonoidRightUnit$[m, A, \mu, \eta, u, \rho, DisplayForm]$ determines whether the 2-cell multiplication μ and the 2-cell unit η satisfies the monoid right unit equation for η to be the unit of the 1-cell monoid A. The last argument is optional (and defaults to *True*). For this function to be used, the 1-cell fusion rules m, the 1-cell unit u, the 2-cell associator α, and the 2-cell right unitor ρ (with additional structures, of course) must form a monoidal category.

$$\begin{tikzcd}
A \otimes A \arrow[dr, "\mu"'] & & A \otimes u \arrow[ll, "id_A \otimes \eta"'] \arrow[dl, "\rho_A"] \\
& A &
\end{tikzcd} \tag{18}$$

IsMonoidLeftUnit$[m, A, \mu, \eta, u, \lambda, DisplayForm]$ determines whether the 2-cell multiplication μ and the 2-cell unit η satisfies the monoid left unit equation for η to be the unit of the 1-cell monoid A. The last argument is optional (and defaults to *True*). For this function to be used, the 1-cell fusion rules m, the 1-cell unit u, the 2-cell associator α, and the 2-cell left unitor λ (with additional structures, of course) must form a monoidal category.

$$\begin{tikzcd}
u \otimes A \arrow[rr, "\eta \otimes id_A"] \arrow[dr, "\lambda_A"'] & & A \otimes A \arrow[dl, "\mu"] \\
& A &
\end{tikzcd} \tag{19}$$

IsMonoid$[m, u, \alpha, \rho, \lambda, A, \mu, \eta]$ determines whether A, μ, and η form a monoid (i.e. an algebra). A is the 1-cell object the algebra is over, μ is the 2-cell multiplication, and η is the 2-cell unit. The structures m, u, α, ρ, λ must form a monoidal category with m the 1-cell product functor, u the 1-cell unit, α the 2-cell associator, ρ the 2-cell right unitor, and λ the 2-cell left unitor. Note: this function does not specify the error if the function returns $False$.

IsMonoidDaggerFrobenius$[m, \alpha, A, \mu, \eta, \mathit{DisplayForm}]$ determines whether the monoid given by A, μ, and η is †-Frobenius. The last argument is optional (and defaults to *False*). For this function to be used, the 1-cell fusion rules m and the 2-cell associator α (with additional structures, of course) must form a monoidal category, and **IsMonoid**$[m, u, \alpha, \rho, \lambda, A, \mu, \eta]$ must be *True* (i.e. A, μ, and η must already be shown to form a monoid).

IsMonoidCommutative$[m, A, \mu, \beta, \mathit{DisplayForm}]$ determines whether the monoid given by A and μ is Commutative. The last argument is optional (and defaults to False). For this function to be used, the 1-cell fusion rules m, the 2-cell associator α, and the 2-cell braid β (with additional structures, of course) must form a braided monoidal category, and **IsMonoid**$[m, u, \alpha, \rho, \lambda, A, \mu, \eta]$ must be *True* (i.e. A, μ, and η must already be shown to form a monoid).

$$\begin{tikzcd}
A \otimes A \arrow[rr, "\beta"] \arrow[dr, "\mu"'] & & A \otimes A \arrow[dl, "\mu"] \\
& A &
\end{tikzcd} \tag{20}$$

5.2.5 Monoidal Functors

IsMonoidalFunctorStructureEquation1$[F, \phi, m_1, \alpha_1, m_2, \alpha_2, \mathit{WithErrors}, \mathit{DisplayForm}]$ is a function that tests if the 1-cell functor F and 2-cell structure ϕ (and the structures of the two monoidal categories, labeled by subscripts 1 and 2, respectively) satisfy the first structure equation for F to be a strong monoidal functor. This equation shows that $(FA \odot FB) \odot FC$ goes to $F(A \otimes (B \otimes C))$ in two different ways, where here, in this notation, $\otimes$ is used to represent the action of m_1 and $\odot$ is is used to represent the action of m_2. The last two arguments are optional (and default to *True* and *True*, respectively). This function tests the form and consistency of the arguments, whether ϕ is invertible, and the structure equation itself. Note: this function should only after the categories given by subscripts 1 and 2 in the input are shown form monoidal categories.

$$\begin{tikzcd}
(FA \bullet FB) \bullet FC \arrow[r, "\alpha_{2\,FA,FB,FC}"] \arrow[d, "\phi_{A,B} \bullet id_{FC}"'] & FA \bullet (FB \bullet FC) \arrow[d, "id_{FA} \bullet \phi_{B,C}"] \\
F(A \otimes B) \bullet FC \arrow[d, "\phi_{A \otimes B, C}"'] & FA \bullet F(B \otimes C)) \arrow[d, "\phi_{A, B \otimes C}"] \\
F((A \otimes B) \otimes C) \arrow[r, "F\alpha_{1\,A,B,C}"'] & F(A \otimes (B \otimes C))
\end{tikzcd} \tag{21}$$

IsMonoidalFunctorStructureEquation2and3$[F, \phi, \phi_u, m_1, \alpha_1, \rho_1, \lambda_1, u_1, m_2, \alpha_2, \rho_2, \lambda_2, u_2,$

WithErrors, DisplayForm] is a function that the 1-cell functor F and 2-cell structures ϕ_u and ϕ (and the structures of the two monoidal categories, labeled by subscripts 1 and 2, respectively) satisfy the second and third structure equations for F to be a strong monoidal functor. The second equation shows that $FA \odot u_2$ goes to FA in two different ways, and the third equation shows that $u_2 \odot FB$ goes to FB in two different ways, where here, in this notation, $\otimes$ is used to represent the action of m_1 and $\odot$ is is used to represent the action of m_2. The last two arguments are optional (and default to *True* and *True*, respectively). This function tests the form and consistency of the arguments, whether ϕ_u is invertible, and the structure equation itself, and then will specify any errors. Note: this function should only after the categories given by subscripts 1 and 2 in the input are shown form monoidal categories and the first monoidal structure equation is shown to be satisfied.

$$\begin{tikzcd}
FA \bullet u_2 \arrow[r, "\rho_2"] \arrow[d, "id_{FA} \bullet \phi_u"'] & FA \\
FA \bullet Fu_1 \arrow[r, "\phi_{A,u_1}"'] & F(A \otimes u_1) \arrow[u, "F\rho_1"']
\end{tikzcd} \tag{22}$$

$$\begin{tikzcd}
u_2 \bullet FB \arrow[r, "\lambda_2"] \arrow[d, "\phi_u \bullet id_{FB}"'] & FB \\
Fu_1 \bullet FB \arrow[r, "\phi_{u_1,B}"'] & F(u_1 \otimes B) \arrow[u, "F\lambda_1"']
\end{tikzcd} \tag{23}$$

IsStrongMonoidalFunctor$[F, \phi, \phi_u, m_1, \alpha_1, \rho_1, \lambda_1, u_1, m_2, \alpha_2, \rho_2, \lambda_2, u_2]$ tests if proposed structures (the 1-cell F and the 2-cells ϕ and ϕ_u) form a strong monoidal functor from the category given by product functor m_1 to the category given by product functor m_2. The structures labeled by 1 are from the category given by m_1 and structures labeled by 2 are from the category given by m_2.

5.2.6 Solvers (prototype)

SolveMonoidal$[m, u]$ solves for the 2-cells α, λ, and ρ, for the candidate monoidal category with product functor 1-cell m and unit 1-cell u. This function is a prototype and tends to run out of memory.

5.2.7 Automatic Reboxing Functions

These functions take care of the "lower-level" work in **2Vect**. Essentially, they allow you to apply morphisms of a particular 2-vector space (i.e. a particular MTC) to specific objects in that 2-vector space (i.e. or MTC). Remember, the objects of a particular 2-vector space are 1-cells in **2Vect**, and morphisms between those objects are 2-cells in **2Vect**.

αABC$[m, \alpha, A, B, C]$ returns the 2-cell given by $\alpha_{A,B,C}$ (i.e. the natural transformation α applied to objects A, B, and C), where A, B, and C are 1-cell objects of the monoidal category, and m is the 1-cell product functor. The output 2-cell is automatically appropriately reboxed.

βAB$[m, \beta, A, B]$ returns the 2-cell given by $\beta_{A,B}$ (i.e. the natural transformation β applied to objects A and B), where A and B are 1-cell objects of the braided monoidal category, and m is the 1-cell product functor. The output 2-cell is automatically appropriately reboxed.

ϑA$[m, \theta, A]$ returns the 2-cell given by θ_A (i.e. the natural transformation θ applied to object A), where A is a 1-cell object of the twisted braided monoidal category, and m is the 1-cell product functor. The output 2-cell is automatically appropriately reboxed.

λA$[m, \lambda, u, A]$ returns the 2-cell given by λ_A (i.e. the natural transformation λapplied to object A), where A is a 1-cell object of the monoidal category, m is the 1-cell product functor, and u is the 1-cell unit object. The output 2-cell is automatically appropriately reboxed.

ρA$[m, \rho, u, A]$ returns the 2-cell given by ρ_A (i.e. the natural transformation ρ applied to object A), where A is a 1-cell object of the monoidal category, m is the 1-cell product functor, and u is the 1-cell unit object. The output 2-cell is automatically appropriately reboxed.

φAB$[m_1, m_2, F, \phi, A, B]$ returns the 2-cell given by $\phi_{A,B}$ (i.e. the natural transformation ϕ applied to objects A and B), where A and B are 1-cell objects of the monoidal category given by m_1. F is the 1-cell monoidal functor, m_1 is the 1-cell product functor of the first category, and m_2 is the 1-cell product functor of the second category. The output 2-cell is automatically appropriately reboxed.

5.3 MTCategory.m

SetMTC$[m, u]$ is used to set the modular tensor category you want to work with. The 2-cell structures are created in the appropriate form with variable entries. Please input m, the 1-cell fusion rules, and u, the 1-cell unit. **SetMTC**$[m, u, alphaSet, rhoSet, lambdaSet$ is used to set the modular tensor category with user inputted 2-cell structures. Please run **IsMTC**[] to ensure that the structures satisfy all the MTC axioms. As before, m, the 1-cell fusion rules, and u, the 1-cell unit, must be inputted as 1-cells.

SetMTCFib[] is used to set the modular tensor category you want to work with to **Fib**, the Fibonacci MTC.

$$m = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \tag{24}$$

$$u = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\tau = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\alpha = \begin{bmatrix} [1] & \emptyset_{0\times 0} & \emptyset_{0\times 0} & [1] & \emptyset_{0\times 0} & [1] & [1] & [1] \\ \\ \emptyset_{0\times 0} & [1] & [1] & [1] & [1] & [1] & [1] & \begin{bmatrix} \varphi^{-1} & \varphi^{-\frac{1}{2}} \\ \varphi^{-\frac{1}{2}} & -\varphi^{-1} \end{bmatrix} \end{bmatrix}$$

$$\rho = \begin{bmatrix} [1] & \emptyset_{0\times 0} \\ \\ \emptyset_{0\times 0} & [1] \end{bmatrix}$$

$$\lambda = \begin{bmatrix} [1] & \emptyset_{0\times 0} \\ \\ \emptyset_{0\times 0} & [1] \end{bmatrix}$$

$$\beta = \begin{bmatrix} [1] & \emptyset_{0\times 0} & \emptyset_{0\times 0} & \left[e^{-\frac{4}{5}i\pi}\right] \\ \\ \emptyset_{0\times 0} & [1] & [1] & \left[e^{\frac{3}{5}i\pi}\right] \end{bmatrix}$$

$$\theta = \begin{bmatrix} [1] & \emptyset_{0\times 0} \\ \\ \emptyset_{0\times 0} & \left[e^{\frac{4}{5}i\pi}\right] \end{bmatrix}$$

SetMTCIsing[] is used to set the modular tensor category you want to work with to the Ising MTC.

$$m = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{25}$$

$$u = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{26}$$

$$\sigma = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{27}$$

$$\psi = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{28}$$

$$\rho = \begin{bmatrix} [1] & \emptyset_{0\times 0} & \emptyset_{0\times 0} \\ \emptyset_{0\times 0} & [1] & \emptyset_{0\times 0} \\ \emptyset_{0\times 0} & \emptyset_{0\times 0} & [1] \end{bmatrix}$$

$$\lambda = \begin{bmatrix} [1] & \emptyset_{0\times 0} & \emptyset_{0\times 0} \\ \emptyset_{0\times 0} & [1] & \emptyset_{0\times 0} \\ \emptyset_{0\times 0} & \emptyset_{0\times 0} & [1] \end{bmatrix}$$

$$\alpha_{\sigma,\sigma,\sigma} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\beta = \begin{bmatrix} [1] & \emptyset_{0\times 0} & \emptyset_{0\times 0} & \emptyset_{0\times 0} & \left[e^{-\frac{1}{8}i\pi}\right] & \emptyset_{0\times 0} & \emptyset_{0\times 0} & \emptyset_{0\times 0} & [-1] \\ \emptyset_{0\times 0} & [1] & \emptyset_{0\times 0} & [1] & \emptyset_{0\times 0} & [-i] & \emptyset_{0\times 0} & [-i] & \emptyset_{0\times 0} \\ \emptyset_{0\times 0} & \emptyset_{0\times 0} & [1] & \emptyset_{0\times 0} & \left[e^{\frac{3}{8}i\pi}\right] & \emptyset_{0\times 0} & [1] & \emptyset_{0\times 0} & \emptyset_{0\times 0} \end{bmatrix}$$

$$\theta = \begin{bmatrix} [1] & \emptyset_{0\times 0} & \emptyset_{0\times 0} \\ \emptyset_{0\times 0} & \left[e^{\frac{1}{8}i\pi}\right] & \emptyset_{0\times 0} \\ \emptyset_{0\times 0} & \emptyset_{0\times 0} & [-1] \end{bmatrix}$$

Setα[αSet] sets the 2-cell α, the associator of the MTC. Input must be the appropriate 2-cell. Please run **IsMTC**[] to ensure that the new structure satisfies all the MTC axioms.

Setρ[ρSet] sets the 2-cell ρ, the right unitor of the MTC. Input must be the appropriate 2-cell. Please run **IsMTC**[] to ensure that the new structure satisfies all the MTC axioms.

Setλ[λSet] sets the 2-cell λ, the left unitor of the MTC. Input must be the appropriate 2-cell. Please run **IsMTC**[] to ensure that the new structure satisfies all the MTC axioms.

Setβ[βSet] sets the 2-cell β, the braid of the MTC. Input must be the appropriate 2-cell. Please run **IsMTC**[] to ensure that the new structure satisfies all the MTC axioms.

Setϑ[θSet] sets the 2-cell θ, the twist of the MTC. Input must be the appropriate 2-cell. Please run **IsMTC**[] to ensure that the new structure satisfies all the MTC axioms.

PrintMTC[] is used to print all the structures of the current MTC. It is not checked for consistency or form. Please run **IsMTC**[] to ensure correct form and to ensure that the structures satisfy all the MTC axioms.

IsMTC[] tests if the currently set MTC satisfies all the axioms. **IsMTC**[$m, \alpha, \rho, \lambda, u, \beta, \theta$] tests if the input structures form a modular tensor category, where where m is the 1-cell product functor, α is the 2-cell associator, ρ is the 2-cell right unitor, λ is the 2-cell left unitor, u is the 1-cell unit, β is the 2-cell braid, and θ is the 2-cell twist.

GenerateHandle[] generates the handle (which is a 1-cell object) of the currently set MTC.
GenerateHandle[m, u] generates the handle of the MTC given by product functor m and unit u. The handle is the sum over all the simple objects of that simple object fused with its dual: $\sum A \otimes A^*$, for all simple objects A.

IsMonoid[A, μ, η] tests if A, μ, and η form a monoid (i.e. an algebra), where A is a 1-cell object the algebra is over (and is an object of the currently set MTC), μ is the 2-cell multiplication, and η is the 2-cell monoid unit. **IsMonoid**[$m, u, \alpha, \rho, \lambda, A, \mu, \eta$] tests whether A, μ, and η form a monoid in the monoidal category given by m, u, α, ρ, and λ (where m is the 1-cell product functor, u is the 1-cell unit, α is the 2-cell associator, ρ is the 2-cell right unitor, and λ is the 2-cell left unitor). Note: does not specify the error (i.e. whether there is a problem with the form of A, μ, or η, or whether the equation fails) if the function returns *False*.

IsMonoidDaggerFrobenius[A, μ, η] determines whether the monoid given by A, μ, and η is †-Frobenius, where A is a 1-cell in the current category (and is the object the monoid is over), μ is the 2-cell multiplication, and η is the 2-cell monoid unit. Note: for this function to be called, A, μ, and η must already be shown to form a monoid.

IsMonoidCommutative[A, μ] determines whether the monoid given by A and μ (η is unneeded for this function) is Commutative, where A a 1-cell object in the current category (and is the object the monoid is over), and μ is the 2-cell multiplication. Note: for this function to be called, A, μ, and η must already be shown to form a monoid.

Getm[] returns the 1-cell m, the fusion rules of the MTC.

Getu[] returns the 1-cell u, the unit of the MTC.

Getα[] returns the 2-cell α, the associator of the MTC.

Getρ[] returns the 2-cell ρ, the right unitor of the MTC.

Getλ[] returns the 2-cell λ, the left unitor of the MTC.

Getβ[] returns the 2-cell β, the braid of the MTC.

Getϑ[] returns the 2-cell θ, the twist of the MTC.

GetPentagonEquation[*DisplayForm*] returns the pentagon equation, an axiom of monoidal categories (and thus MTCs). *DisplayForm* is an optional argument that defaults to *True*. In general, this function is intended to be used when α has variable entries.

$$\begin{tikzcd}
& ((A\otimes B)\otimes C)\otimes D \arrow[dl, "\alpha_{A,B,C}\otimes id_D"'] \arrow[dr, "\alpha_{A\otimes B,C,D}"] & \\
(A\otimes(B\otimes C))\otimes D \arrow[d, "\alpha_{A,B\otimes C,D}"'] & & (A\otimes B)\otimes(C\otimes D) \arrow[d, "\alpha_{A,B,C\otimes D}"] \\
A\otimes((B\otimes C)\otimes D) \arrow[rr, "id_A\otimes\alpha_{B,C,D}"'] & & A\otimes(B\otimes(C\otimes D))
\end{tikzcd} \tag{29}$$

GetTriangleEquation[*DisplayForm*] returns the triangle equation, an axiom of monoidal categories (and thus MTCs). *DisplayForm* is an optional argument that defaults to *True*. In general, this function is intended to be used when α, λ, and/or ρ have variable entries.

$$\begin{tikzcd}
(A\otimes u)\otimes B \arrow[rr, "\alpha_{A,u,B}"] \arrow[dr, "\rho_A\otimes id_B"'] & & A\otimes(u\otimes B) \arrow[dl, "id_A\otimes\lambda_B"] \\
& A\otimes B &
\end{tikzcd} \tag{30}$$

GetHexagonEquation1[*DisplayForm*] returns the first hexagon equation for braided monoidal categories, an axiom of MTCs. *DisplayForm* is an optional argument that defaults to *True*. In general, this function is intended to be used when β and/or α have variable entries.

$$\begin{tikzcd}
& A\otimes(B\otimes C) \arrow[r, "\beta_{A,B\otimes C}"] & (B\otimes C)\otimes A \arrow[dr, "\alpha_{B,C,A}"] & \\
(A\otimes B)\otimes C \arrow[ur, "\alpha_{A,B,C}"] \arrow[dr, "\beta_{A,B}\otimes id_C"'] & & & B\otimes(C\otimes A) \\
& (B\otimes A)\otimes C \arrow[r, "\alpha_{B,A,C}"'] & B\otimes(A\otimes C) \arrow[ur, "id_B\otimes\beta_{A,C}"'] &
\end{tikzcd} \tag{31}$$

GetHexagonEquation2[*DisplayForm*] returns the first hexagon equation for braided monoidal categories, an axiom of MTCs.*DisplayForm* is an optional argument that defaults to *True*. In general, this function is intended to be used when β has variable entries, but MUST NOT be used when α has variable entries since α^{-1} is required (since a variable α cannot be inverted).

$$\begin{tikzcd}
& (A\otimes B)\otimes C \arrow[r, "\beta_{A\otimes B,C}"] & C\otimes(A\otimes B) \arrow[dr, "\alpha^{-1}_{C,A,B}"] & \\
A\otimes(B\otimes C) \arrow[ur, "\alpha^{-1}_{A,B,C}"] \arrow[dr, "id_A\otimes\beta_{B,C}"'] & & & (C\otimes A)\otimes B \\
& A\otimes(C\otimes B) \arrow[r, "\alpha^{-1}_{A,C,B}"'] & (A\otimes C)\otimes B \arrow[ur, "\beta_{A,C}\otimes id_B"'] &
\end{tikzcd} \tag{32}$$

GetTwistEquation[*DisplayForm*] returns the twist equation, an axiom of MTCs. *DisplayForm* is an optional argument that defaults to *True*. In general, this function is intended to be used when β and/or θ have variable entries.

$$\begin{tikzcd}
A \otimes B \arrow[r, "\theta_{A\otimes B}"] \arrow[d, "\theta_A \otimes \theta_B"'] & A \otimes B \\
A \otimes B \arrow[r, "\beta_{A,B}"'] & B \otimes A \arrow[u, "\beta_{B,A}"']
\end{tikzcd} \tag{33}$$

GetRibbonCoherenceEquation[$DisplayForm$] returns the equation of coherence between the 2-cell twist θ and the dual structure of the MTC. This is a required axiom of MTCs.
DisplayForm is an optional argument that defaults to *True*.

$$\begin{tikzcd}
u \arrow[r, "\eta_A"] & A \otimes A^* \arrow[r, bend left, "id_A \otimes \theta_{A^*}"] \arrow[r, bend right, "\theta_A \otimes id_{A^*}"'] & A \otimes A^*
\end{tikzcd} \tag{34}$$

GetMonoidAssociativityEquation[$A, \mu, DisplayForm$] returns the associativity equation for A and μ to form a monoid, where A is the 1-cell object the algebra is over, and μ is the 2-cell monoid multiplication. *DisplayForm* is an optional argument that defaults to *True*.

$$\begin{tikzcd}
 & (A \otimes A) \otimes A \arrow[dl, "\mu \otimes id_A"'] \arrow[dr, "\alpha_{A,A,A}"] & \\
A \otimes A \arrow[d, "\mu"'] & & A \otimes (A \otimes A) \arrow[d, "id_A \otimes \mu"] \\
A & & A \otimes A \arrow[ll, "\mu"]
\end{tikzcd} \tag{35}$$

GetMonoidRightUnitEquation[$A, \mu, \eta, DisplayForm$] returns the right unit equation for A, μ, and η to form a monoid, where A is the 1-cell object the algebra is over, μ is the 2-cell monoid multiplication, and η is the 2-cell monoid unit. *DisplayForm* is an optional argument that defaults to *True*.

$$\begin{tikzcd}
A \otimes A \arrow[dr, "\mu"'] & & A \otimes u \arrow[ll, "id_A \otimes \eta"'] \arrow[dl, "\rho_A"] \\
 & A &
\end{tikzcd} \tag{36}$$

GetMonoidLeftUnitEquation[$A, \mu, \eta, DisplayForm$] returns the left unit equation for A, μ, and η to form a monoid, where A is the 1-cell object the algebra is over, μ is the 2-cell monoid multiplication, and η is the 2-cell monoid unit. *DisplayForm* is an optional argument that defaults to *True*.

$$\begin{tikzcd}
u \otimes A \arrow[rr, "\eta \otimes id_A"] \arrow[dr, "\lambda_A"'] & & A \otimes A \arrow[dl, "\mu"] \\
& A &
\end{tikzcd} \tag{37}$$

GetMonoidDaggerFrobeniusEquation$[A, \mu, \eta, DisplayForm]$ returns the †-Frobenius equation, the conditions required for the monoid formed by A, μ, and η to be †-Frobenius (where A is the 1-cell object the algebra is over, μ is the 2-cell monoid multiplication, and η is the 2-cell monoid unit). *DisplayForm* is an optional argument that defaults to *True*.

GetMonoidCommutativityEquation$[A, \mu, DisplayForm]$ returns the commutativity equation, the conditions required for the monoid formed by A, μ, and η to be commutative (where A is the 1-cell object the algebra is over, μ is the 2-cell monoid multiplication, and η is the 2-cell monoid unit). *DisplayForm* is an optional argument that defaults to *True*.

$$\begin{tikzcd}
A \otimes A \arrow[rr, "\beta"] \arrow[dr, "\mu"'] & & A \otimes A \arrow[dl, "\mu"] \\
& A &
\end{tikzcd} \tag{38}$$

α$[A, B, C]$ returns the 2-cell associator given by $\alpha_{A,B,C}$ (i.e. α applied to objects A, B, and C), where A, B, and C are 1-cell objects of the monoidal category. The output 2-cell is automatically appropriately reboxed.

β$[A, B]$ returns the 2-cell braid given by $\beta_{A,B}$ (i.e. β applied to objects A and B), where A and B are 1-cell objects of the braided monoidal category. The output 2-cell is automatically appropriately reboxed.

ϑ$[A]$ returns the 2-cell twist given by θ_A (i.e. θ applied to object A), where A is a 1-cell object of the twisted braided monoidal category. The output 2-cell is automatically appropriately reboxed.

λ$[A]$ returns the 2-cell left unitor given by λ_A (i.e. λ applied to object A), where A is a 1-cell object of the monoidal category. The output 2-cell is automatically appropriately reboxed.

ρ$[A]$ returns the 2-cell right unitor given by ρ_A (i.e. ρ applied to object A), where A is a 1-cell object of the monoidal category. The output 2-cell is automatically appropriately reboxed.

$\mathbf{A} \otimes \mathbf{B}$ or **CircleTimes**$[A, B]$ monoidally combines in the underlying MTC. If A and B are 1-cells (i.e. objects in the underlying MTC), then this is given by **OneCellComposition**$[m$, **OneCellTimes**$[A, B]]$. If A and B are 2-cells (i.e. morphisms in the underlying MTC), then this is given by **TwoCellHorizontalComposition[id**$[m]$**, TwoCellTimes**$[A, B]]$. For both, m represents the fusion rules of the underlying category. If the arguments are *all* 1-cells, this is an n-ary operation $\mathbf{A} \otimes \mathbf{B} \otimes \ldots$, if the arguments are 2-cells the operation is only binary (so please use appropriate bracketing).

References

[1] J.C. Baez. Higher-dimensional algebra II: 2-Hilbert spaces. *Arxiv preprint q-alg/9609018*, 1996.

[2] B. Coecke. Quantum picturalism. *Arxiv preprint arXiv:0908.1787*, 2009.

[3] J. Elgueta. A strict totally-coordinatized version of Kapranov and Voevodsky's 2-category 2Vect. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 142, pages 407–428. Cambridge Univ Press, 2007.

[4] A. Joyal and R. Street. The geometry of tensor calculus i. *Advances in Mathematics*, 88(1):55–112, 1991.

[5] M. Kapranov and V. Voevodsky. Braided monoidal 2-categories and Manin-Schechtman higher braid groups. *Journal of Pure and Applied Algebra*, 92(3):241 – 267, 1994.

[6] S. Mac Lane. *Categories for the Working Mathematician.* Springer, 1998.

[7] Daniel A. Roberts. Representing modular tensor categories: A computer algebra system for topological quantum computing. Master's thesis, Department of Computer Science, University of Oxford, 2011.