THESIS CHAPTER 4: COMPUTATE

CALEB HILL

Contents

1. Mathematica hacking

A first reference for Mathematica programming is [**mathematica˙programming**]. For the those with, for instance, C-style programming background, Mathematica will seem strange. There are ways of doing things that are "correct" and "incorrect." One should start recognizing this pattern from StackExchange posts. The questioner will show their initial attempt using, e.g., a `For[]` statement, and someone more savvy will respond with a `Table[]` statement. This comes down to Mathematica's preference for functional programming

1.1. Functional programming basics.

While calling Mathematica "functional" might technically be incorrect, the mathematician learning to code in the language should keep in mind that Mathematica is what might be described as *function-centric*. That is, the basic objects, data types, etc. can be represented as function values. For example, one might want to represent a list of elements, say `{1,2,3}`. In Mathematica, one may type this in two equivalwent ways: `{1,2,3}` or `List[1,2,3]`. Thus a list can be thought of as the output of the function `List[]` on any finite number of inputs. Similarly one may represent `a+b` equivalently by `Plus[a,b]`. A debugging tool the new Mathematica programmer ought to keep in mind is use of the `Head[]` function. This function answers the question, "This object may be represented as the output of which function?" For example, `Head[1]` returns `Integer`, `Head[1/7]` returns `Rational`, `Head[a+b]` returns `Plus`, and `Head[{1,2,3}]` returns `List`. In the sequel, for example, we use the `Head` function to parse products by inspecting for type `NCMult` (product of noncommutative factors) or `Integer`. As a final example if the function-centric nature of Mathematica, we note that we can extract elements of a list (or other `Head` types) using the `Part[]` function. That is, in order to obtain the second element of the list `someList` we may equivalently write `Part[someList,2]` or `someList[[2]]`.

A more typical example of how Mathematica's function-centric nature manifests is looping. The programmer familiar with, for instance, C++ might default to using a `For[]` which involves manually specifying the iterator and how it iterates. The "correct" way to accomplish the same task in Mathematica is usually to use a `Table[]` statement. Using `Table[]` takes the work of specifying iteration out of the programmer's hands. It has the additional benefit of giving the ability to swap `Table[]` for `ParallelTable[]` where applicable. The `Table[]` function also has a built in way to iterate over multiple dimensions, streamlining workflow and readability.

1.2. Symbols.

Functional or function-centric programming streamlines some aspects of the programmer's task when using Mathematica. The platform's real power comes from its ability to do symbolic manipulations. Mathematica's symbolic abilities make it a natural companion to, and really an extension of, the mathematician's blackboard. Powerful symbolic manipulation has consequences for both algorithmic problems (SAMs) and numerical problems (equation solving). We discuss examples of both, beginning with the algorithmic case.

1.2.1. *SAMs.*

We begin by discussing where Mathematica's symbolic power helps us solve an algorithmic problem. Recall from the definition of $\mathrm{GPA}(\Gamma)$ that the defining basis for

$$\mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(m \to n)$$

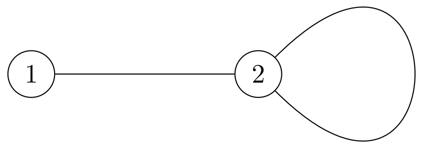


FIGURE 1. The tadpole graph.

is given by pairs (p, q) of parallel paths of length m and n, respectively. In this definition the possibility of double edges requires that we take an edge-centric view of paths, not a vertex-centric one. That is, there may be two distinct edges

$$f, f' : v \to w$$

which, if taking a vertex centric view would both be specified by the pair (v, w) of vertices. However, we must distinguish these paths. The solution is to use a *symbolic adjacency matrix (SAM)* [2]. The graphs we use are not sparse, so even a dogmatic computer scientist would concede that an adjacency matrix is warranted.

Second, it was crucial to the problem to keep track of which direction a path traversed an edge in. The SAM accomplished this by its lack of symmetry; every "half" of every edge of the graph was given a unique symbol. That is, the unoriented edge $u \leftrightarrow v$ is split into two antiparallel edges $u \to v$ and $v \to u$.

Definition 1 (Symbolic Adjacency Matrix)**.** *Let Γ be a fintie graph. If Γ is undirected, split every edge of Γ into a pair of antiparallel directed edges. Label every edge $e : v_i \to v_j$ of Γ by a noncommutative variable a_e. The* **symbolic adjacency matrix** *(SAM) of Γ is the matrix S_Γ whose (i, j) entry is*

$$\sum_{e:v_i \to v_j} a_e$$

Take as an example the tadpole graph in Figure 1. Its symbolic adjacency matrix is

$$\begin{bmatrix} 0 & a \\ b & c \end{bmatrix}$$

where a, b, and c are noncommutative symbolic variables. Recall that the usual adjacenecy matrix (whose entries are nonnegative integers counting edges between vertices) can be raised to the n-th power to count the number of paths of length n between pairs of vertices. In a similar way, the degree-n symbolic polynomials which are the entries in the n-th power of the SAM explicitly give the paths between vertices. For instance, the $(2, 2)$ entry in the square of the above SAM is $c^2 + ba$.

This tells us that the two loops at vertex 2 are[1]

$$c^2 = 2 \to 2 \to 2$$
$$ba = 2 \to 1 \to 2$$

For small graphs, this offers the mathematical programmer a more intuitive way to compute loops in Mathematica than the usual BFS or DFS. We pair paths in the following manner.

Definition 2. *Let A and B be $m \times n$ matrices with noncommutative polynomial entries*

$$p_{i,j;A} =, \quad \text{and} \quad p_{i,j;B} =$$

Define $A \boxtimes B$ to be the $m \times n$ matrix whose (i,j) entry is the formal sum

$$\sum(...,...)$$

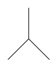
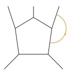
where

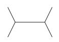
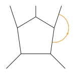
The following Proposition is a straightforward generalization of the results of [2] with an application to the context of graph planar algebras.

Proposition 1. *Let A be the SAM for the graph Γ. Then the terms of the nonzero entries of $A^m \boxtimes A^n$ form a basis for $Hom_{\mathrm{GPA}(\Gamma)}(m \to n)$.*

Proposition 1 is the theoretic backing for the function `GetHomBasisList[]` discussed below.

1.2.2. *Symbolic Equation Solving.* My work also requires finding exact solutions to large systems of equations. Despite the low order of these equations, solving them can require large amounts of computational resources due to their symbolic nature.

1.3. **Algorithms and Data Structures.** The computations required to describe the quantum subgroups of type G_2 are good test cases for a computational framework. The sizes of the hom-spaces grow fast enough that the largest computations like finding (decPentagon) relations becomes intractable with algorithms that are too naive. For instance, building from the GPA embedding of the trivalent generator  to the embedding of the pentagon  requires composing an element of $\mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(2 \to 4)$, an element of $\mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(4 \to 2)$, an element of $\mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(2 \to 4)$, an element of $\mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(4 \to 3)$, and an element of $\mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(4 \to 3)$. At level 4, these hom-spaces are of dimensions 7,776, 7,776, 7,776 and 34,528, respectively. Doing this with the naive $\mathcal{O}(N^2)$ algorithm would require at least 16,234,547,147,440,128 operations, which is obviously infeasible. Instead, one might hope to find a way to compose morphisms with a lower time complexity. Mathematica's Association data structure offers some hope of at least drastically lowering the complexity coefficient.

Regardless of implementation, one trick one should employ is precomputing values that arise frequently. For example in the example above with (decPentagon), we can spot  in the lower part of the figure. Computing this morphism earlier obviously saves us the time of computing it as part of  and decreases the number of operations needed by $\frac{207}{7776^2}$.

[1]Paths are read left to right.

2. Computational Algebraic Number Theory

Here we explain some of the ways we do computational algebraic number theory (CANT), both explicity and implicitly.

2.1. **CANT setup.** In the course of finding the projection coefficients and the $\mathbb{Z}_n$-like extension structure constants at level 4, it was necessary to work over the field

$$K := \mathbb{Q}(q, a_5, b_5, b_6, b_7, c_1),$$

where we have

$$q = e^{2\pi i/48}, \quad a_5 = \frac{2}{2}, \quad b_5 = \sqrt{\cdots}, \quad b_6 = \sqrt{\cdots}, \quad b_7 = \sqrt{\cdots}, \quad c_1 = .$$

While in theory we should be able to accomplish this directly by using Mathematica's built-in support for algebraic numbers with

```
ToNumberField[{q, a5, b5, b6, b7, c1}],
```

we found this to be too slow[2]. Instead, something more clever is required. We exploit the isomorphism $L(\gamma) \cong L[x]/(\min_\gamma(x))$. Practically, this involves combining Mathematica's lighting-fast (for low degree) `AlgebraicNumber` operations with its reasonably quick `PolynomialMod` function.

The first useful observation to make is that we have the following field diagram:

$$\begin{tikzcd}
\mathbb{F}(q) & \mathbb{F}(a_5) & \mathbb{F}(b_5) & \mathbb{F}(b_6) & \mathbb{F}(b_7) & \mathbb{F}(c_1) \\
& & \mathbb{F} := \mathbb{Q}(g) \arrow[ull, dash, "4"] \arrow[ul, dash, "4"] \arrow[u, dash, "2"] \arrow[ur, dash, "2"] \arrow[urr, dash, "2"] \arrow[urrr, dash, "2"] & & & \\
& & & \mathbb{Q} \arrow[ul, dash, "4"] & &
\end{tikzcd}$$

where $g = \sqrt{2 + \sqrt{3}}$[3]. This observation allows us to reduce high-order expressions in $\mathbb{Q}(q, a_5, b_5, b_6, b_7, l)$ using the minimal polynomials for q, a_5, b_5, b_6, b_7, and c_1 to lower-order expressions. We gain additional speed by working over $\mathbb{F}$ instead of $\mathbb{Q}$. The minimal polynomials for q, a_5, b_5, b_6, b_7, and c_1 over $\mathbb{F}$ are:

$$\begin{gathered}
p_q(\eta) = \eta^4 - g\eta^2 + 1 \\
p_{a_5}(\alpha) = \alpha^4 + (6g - 2g^3)\alpha^2 + (5 - 8g - 1g^2 + 2g^3) \\
p_{b_5}(\beta_1) = \beta_1^2 + (1 - 1g) \\
p_{b_6}(\beta_2) = \beta_2^2 + (2g - g^2) \\
p_{b_7}(\beta_3) = \beta_3^2 - (2 - 8g + 2g^3) \\
p_{c_1}(\lambda) = \lambda^2 - (4 + 10g + 2g^3)
\end{gathered}$$

[2]We naively attempted this direct approach and terminated the computation after about three weeks.

[3]Note that $\mathbb{Q}(g) = \mathbb{Q}(\sqrt{2}, \sqrt{3})$. We use g as a primitive element to unlock algebraic number functionalities.

where we represent, e.g., the algebraic number $5 - 8g - 1g^2 + 2g^3 \in \mathbb{F}$ by

```
AlgebraicNumber[Root[1-4#^2+#^4&,4], {5,-8,-1,2}]
```

Thus we are able, without knowing its precise value, to deduce that, for example,

$$\frac{a_5 q}{(1+q^2)b_5} = a_5 b_5 q(g_1 + g_2 q^2)$$

where

$$g_1 = -1 - g + g^2, \quad \text{and} \quad g_2 = \frac{1}{2}(1 + g + g^2 + g^3)$$

by reducing modulo the minimal polynomials.

We implement this logic in Mathematica by giving g as

```
AlgebraicNumber[Root[1-4#^2+#^4&,4], {0, 1, 0, 0}]
```

and reducing the expression $\frac{\alpha\eta}{(1+\eta^2)\beta_1}$ modulo the minimal polynomials; reducing the polynomials we encoutner modulo quadratic and quartic minimal polynomials with `AlgebraicNumber` coefficients is quite fast.

2.2. **CANT for embeddings.**

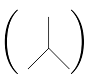
This approach derives efficiency in the two main tasks we have to accomplish: finding GPA embeddings, and validating equations among generators.

The process of finding a GPA embedding is a dance back and forth between approximations and exact solutions, with a heavy theme of making strong assumptions. Any assumption we make, however, will eventually be verified precisely. Many times we initially find a numerical approximation to an embedding, and use this to inform our guesses for exact solutions. A close approximation of a solution gives strong hypotheses for questions of the following type:

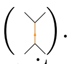
- Which coordinates are zero?
- Which pairs of coordinates are the same?
- Which pairs of coordinates are conjugate?
- Which coordniates have good immediate guesses? (For example, $-0.4999 + 0.866025i$ is likely $e^{2\pi i/3}$.)

Suppose we have obtained an embedding of $\mathcal{G}_2(q)$:

$$F\left(\right) = \sum_{i=1}^{N} \alpha_i (p_i, q_i) \in \mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(2 \to 1).$$



and we are searching for the image of the second generator for $\mathcal{D}_4$: $F\left(\right)$. In practice we find that for each i, $\alpha_i \in \mathbb{R}$. Furthermore, the set of distinct magnitudes,

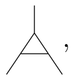


$$\{|\alpha_i|\}$$

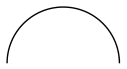
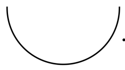
is quite small. At level 4, we have only 9 distinct magnitudes of trivalent coordinates, despite $\mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(2 \to 1)$ being 88. Upon analysis of these nine magnitudes, we find that each lies within the number field generated by one of four of them. These generators are

$$a_5 = \frac{2}{2}, \quad b_5 = \sqrt{\cdots}, \quad b_6 = \sqrt{\cdots}, \quad \text{and} \quad b_7 = \sqrt{\cdots}.$$

We therefore decide that we will work, in practice, with symbolic variables α, β_1, β_2, and β_3, and reduce any expressions we encounter modulo the minimal polynomials for the generators *in the variables α, β_1, β_2, and β_3*. Now, in order to express summands of the defining relations of $\mathcal{D}_4$ such as

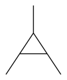
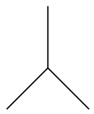
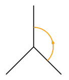
,

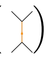
we need access to rigidity maps

 and .

By definition of GPA(Γ), we know the coordinates of these maps. In order to apply the same CANT techniques, we use symbolic variables for these coordinates: λ and $\frac{1}{\lambda} = \frac{\lambda}{4+10g+2g^3}$. In this way λ plays the role of c_1. The final symbol we need is η, which will play the role of q.

Now suppose we find a numerical approximation for the image of the projection in GPA(Γ). We may arrive at this point by solving a large linear system (i.e., the half-braid relation), by applying optimization to some nonlinear relations, or some other means. We can use this approximation to solve, for example, the linear system

 $= t_1$  $+ t_2$ 

to find guesses for the values of t_1 and t_2. Once we have our *guesses* for t_1 and t_2, we may turn this into an *assumption* by imposing the exact, symbolic relationship and returning equations which are linear in the coordinates of F ().

2.3. **CANT for equation validation.** Suppose we have an equation in the two generators: trivalent and projection (and cups, caps, and sticks). With symbolic GPA embeddings of the generators as above, this amounts to a linear combination of vectors, where the vectors have symbolic polynomial entries and are scaled by symbolic polynomials. In order to prove these equations are true, we can either specialize the symbols to their numeric values, or show that the equations hold purely symbolically. It turns out the latter approach is superior in the cases we are interested in.

We actually require very few relationships between the symbolic variables to hold in order to show that the cyclic extension equations are true. The following relations, in addition to the defining relations from the minimal polynomials, turned out to be sufficient to verify all equations at level 4:

$$\begin{aligned}
\beta_3 &= \beta_1\lambda(\frac{3}{2} - \frac{3}{2}g - \frac{1}{2}g^2\frac{1}{2}g^3) \\
\beta_1 &= -\alpha^3(\frac{1}{2} + \frac{1}{2}g) - \alpha(1 + \frac{3}{2}g - g^2 - \frac{1}{2}g^3) \\
\beta_3\lambda &== \alpha((-5 - 9g + 3g^2 + 3g^3) + \alpha^2(-2 - 5g + g^3)).
\end{aligned}$$

Sufficiency here means that the following process terminates. Suppose we have an expression $\mathcal{E}$ which we would like to show is equal to 0. Impose the known relations on $\mathcal{E}$: those given by the minimal polynomials in addition to those listed above. Expand the result $\mathcal{E}_1$ and combine all like terms. Once again impose the relations on $\mathcal{E}_1$ to obtain $\mathcal{E}_2$; combine like terms again. In the cases we worked with, $\mathcal{E}_2$ was identically 0 after combining like terms.

3. Documentation

As discussed above, we use the package `NCAlgebra`. We also use the convention that if $p = a_1 * \cdots * a_m$ and $q = b_1 * \cdots * b_n$ are two parallel paths from vertex i to vertex j, then we represent the multiple $\alpha(p,q) \in \mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(m \to n)$ of basis vector (p,q) by the Mathematica `Association`

```
<| scalar->alpha, p-> a1**...**am, q->b1**...**bn, s->i, t->j |>
```

and consider a list of such associations to represent a linear combination of basis vectors. Note that `**` denotes noncommutative product. Some of the following functions require that linear combinations of basis vectors exhibit certain properties, such as respecting the order of a certain basis. When this is the case, an appropriate `Assert` statement has been inserted.

3.1. Dagger, RealDagger, and SymDagger.

The graph planar algebra is given a $\dagger$ structure by defining the dagger $\left(\sum_i \alpha_i(p_i, q_i)\right)^\dagger = \sum_i \overline{\alpha_i}(q_i, p_i)$. It is important to note that this operation does not change the source and target of the paths p_i and q_i. This operation is captured in the function `Dagger`. When coefficients are known (or on an ad hoc basis assumed) to be real, one may use the `RealDagger` function which avoids the computational penalty of performing computations on elements of the form `Conjugate[x]`.

When using the CANT framework, we need a suitable way to compute complex conjugates for the dagger operation. This is accomplished on a case-by-case basis. When q is a root of unity this is fairly easy. The symbolic variable η plays the role of q, and we impose the relation defined by the minimal polynomial on η:

$$\eta^4 = g\eta^2 - 1. \tag{1}$$

Thus if q is, in the case of G_2 at level 4, a 48-th root of unity, we know $\overline{q} = q^{47}$. To express $\overline{q}$ symbolically with η, then, we merely reduce η^{47} according to Equation 1.

We may observe these three distinct ways of computing daggers:

Function	Scalar computation
`Dagger`	`"scalar" -> Conjugate[inputList[[kindex]]["scalar"]]`
`RealDagger`	`"scalar" -> inputList[[kindex]]["scalar"]`
`SymDagger`	`"scalar" ->  symConj[inputList[[kindex]]["scalar"]]`

In the above, `SymConj` is defined by

```
symConj[expr_] := ApplyRelns[expr /. \[Eta] -> -\[Eta] (\[Eta]^2 -g )]
```

3.2. Compose.

Composing an element of $\mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(m \to n)$ with an element of $\mathrm{Hom}_{\mathrm{GPA}(\Gamma)}(l \to m)$ is defined on basis elements by

$$(p', q') \circ (p, q) = \delta_{q',p} \cdot (p', q)$$

This basis-vector-to-basis-vector composition is reflected in the function `LilCompose`. Composing a pair of linear combinations (i.e. lists) of basis vectors is taken care of by `BigCompose`. The function `Comp` handles arbitrary compositions of the form $f_1 \circ \cdots \circ f_r$. The mechanism of `LilCompose` is fairly straightforward and should present little trouble to understanding.

At the moment `BigCompose`, and therefore `Comp`, use the naive $\mathcal{O}(n^2)$ algorithm for computing compositions of lists. There is hope that a linear-time algorithm utilizing hash maps exists.

3.3. **Tensor.** Tensoring is a similar story to composition. The monoidal product on the GPA is given in terms of the defining basis by

$$(p,q) \otimes (p',q') = \delta_{s(p'),t(p)}(pp',qq').$$

The function `LilTens` performs this operation on pairs of basis vector associations: $(p,q) \otimes (p',q')$. The function `BigTens` performs this operation on pairs of lists of basis vector associations: $f \otimes g$. The function `Tens` performs this operation on an arbitrary list of lists of basis vector associations: $f_1 \otimes \cdots \otimes f_r$.

As with composition, we currently use the naive quadratic time algorithm, with a hope of optimizing to a linear time algorithm.

3.4. **Cup, Cap, and Stick.** The rigidity of the embedding $\mathcal{P}_{X;\mathcal{C}} \to \mathrm{GPA}(\Gamma)$ implies that the Cup and Cap maps are sent to corresponding rigidity maps in $\mathrm{GPA}(\Gamma)$. As defined in [1], we have

$$\mathrm{ev} = \sum_{(e,f)} \sqrt{\frac{\lambda_{s(f)}}{\lambda_{s(e)}}}((e,f),s(e)) \tag{2}$$

where the sum is across all pairs (e,f) of antiparallel edges in Γ, and where λ is the Frobenius eigenvector of Γ. For instance, when

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

we have

$$\lambda = \begin{bmatrix} \frac{1}{2}(-3+\sqrt{21}) \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

and corresponding positive eigenvalue $\frac{1}{2}(3+\sqrt{21})$

3.5. **InTermsOf.**

3.6. **GetHomBasisList.** See Proposition 1.

3.7. **Respecting an Ordered Basis and Equating.**

References

[1] Daniel Copeland and Cain Edie-Michell. *Cell Systems for* $\overline{\mathrm{Rep}(U_q(\mathfrak{sl}_N))}$ *Module Categories*. 2023. arXiv: `2301.13172 [math.QA]`.

[2] Prabhaker Mateti and Narsingh Deo. "On Algorithms for Enumerating All Circuits of a Graph". In: *SIAM Journal on Computing* 5.1 (1976), pp. 90–99. DOI: `10.1137/0205007`. eprint: `https://doi.org/10.1137/0205007`. URL: `https://doi.org/10.1137/0205007`.