Complexity Theory

Daniel Nikpayuk

November 22, 2016

Overview

There is no absolute definition of what complexity means; the only consensus among researchers is that there is no agreement about the specific definition of complexity.[1]

We begin with the above quote, noting first of all it is taken from Wikipedia which does not always hold up to academic rigour, but it's also worth noting—interestingly enough—if you do take the above quote at face value, that the very definition of *complexity* seems to be not so simple. You might even be tempted to say the very definition of complexity is in fact "complex".

As far as I can tell the lack of unity for a definition stems from the need to study complexity within different academic disciplines, each trying to frame the concept from its own perspective, each with its own intentions and expectations.

I attempt to do the same here.

The perspective I am approaching complexity from is that of mathematics. I seek to describe a complexity of *patterns* in the abstract, using versions of complexity that show up in the various branches of mathematics (including computing science) themselves.

philosophy: a comparative lens

The underlying intuitive narrative approach I have chosen to take comes from my view that humans are *toolmakers*, where our tools are the mitigators and mediators of the complexities we are required to navigate time and again throughout our lives.

As human life is such a broad experience, I in fact turn to the humanities as an inspiration for a universal language to describe complexity.[2] Why? Of all the tools humans have made, *language* itself is our most sophisticated one. As humans we intake information from our senses. We intake too much information to process it all. Our sensory input is so intense, with so many patterns we need tools to mitigate this complexity. *This is language.* And yet, as any humanist will protest, language is so much more than just a descriptive tool. I do not dispute this claim, and in another setting would be more than happy to discuss the humanist aesthetic of it with you; but on a more practical level I also observe language is in fact used by writers and poets and artists as well as others to express themselves. To express the complex experiences of their beautiful lives, and that humanists in particular have specialized their skillset in way of a form of language to access and navigate all these texts and works of literature. This is to say: Humanists have a specialized academic language to describe the ways in which humans in general use language to express the complexity they experience through life.

Now, I can't just take the humanists' approach unfiltered: The reality is I have borrowed from the humanists' wisdoms—even their terminology—but I have otherwise translated and sanitized my understanding of their discipline into a more formal mathematical setting. If you're a humanist reading this, feel free to protest, though I do offer you a mind game as distraction: I as a *mathematician* am trying to describe to you the *humanist*, what the humanities is. Since you already know what the humanities are, by reading my description of it, it will in fact teach you more about what mathematics actually is. Does such a thought experiment make sense? Is it of interest?

[1]https://en.wikipedia.org/wiki/Complexity; November 1, 2016

[2]If you've read my previous works, you will be familiar with this much, so thank you for bearing with it for the sake of the uninitiated.

In the direction of translating the humanities, I first offer my definition[3] of a *technology space*:

- **context**: Context is the data and sensory input of discussion. It is assumed to already exist, no justification is required as such. It is the raw experience of life itself. It is *memory.*
- **semiotics**: Semiotics are the signs and signifiers we use to represent this raw context we experience and inevitably need to describe. It forms the socially constructed realities and collective protocols of shared communication within a group. It starts with primitive forms—atomic patterns—and as building blocks combines to form more complicated patterns to recognize, access, and navigate a given context.
- **media**: The media is the infrastructure which relates how the semiotics are applied to the context of interest, and how that context of interest is constrained by its representation.

I offer this definition as a starting point, as it seems to form one of the most common paradigms to critically analyse texts and works within the humanist realm. And of the humanist paradigms, it is the one most readily translated into a formal mathematical setting—especially in attempt to describe something as elusive as *complexity* itself.

Why? Because complexity at its core, intuitively, does not exist objectively. It is not an attribute that can be applied to something by assessing that something's inner standard, its *essence.* Complexity is *relational.* This is to say, complexity is *comparative.* Something is only complex in comparison to something else. To this end, a technology space within its framework already provides a prerequisite comparison. A context space is (or can be) complex only relative to the semiotic space and media space used to represent it.

Tools are only useful for a purpose if they are simpler (in some way or another) than whatever it is they are meant to replace. If you are looking to meet with your friends, a phone is relatively simpler than navigating your social sphere in person just to coordinate when and where you will meet.

Tools are only effective if they are relatively less complex than that which they are trying to access or navigate or manipulate. Language as a tool, as a technology space, is usually only of interest if the semiotics and medium of expression are relatively less complex than the life experience they are meant to model.

This is the underlying philosophy and motivation towards a definition of complexity.

philosophy: a reductive lens

Expanding upon the above philosophical motivation, I would say complexity has everything to do with *compression.* Why? It is in our attempt to compress a context space (by means of a semiotic and media space) that we even become exposed to and aware of complexity in the first place.

One intuitive descriptor of complexity then is that no matter how you try to simplify it, it just won't. This is not to say it cannot be simplified at all, rather no matter how you *do* simplify it, there always ends up something left over which fundamentally cannot simplify.

I should mention there is a competing intuitive essentialism toward complexity: "The whole is greater than the sum of its parts". This is a well known adage, describing otherwise the term *emergence.* This is to say, a complex *system* may be partially described by simpler parts, but that no "proper" subsystem of such parts is able to describe all of the behaviours of the system as a whole. Each subsystem, or each combination of subsystems may be able to reveal certain aspects of the whole, but some behaviours are only present when looking it in its entirety. This is to say, some behaviours are emergent within this system which—given this understanding—may now be described as *irreducible.*

I would say this competing intuitive descriptor comes from an additive lens. I myself have taken the stance that in fact these are not competing essentialisms but rather one follows from the other. In particular I have chosen to begin with the reductive approach because the negative has higher entropy than the positive (as it assumes less). Better to start with weaker assumptions. This much of course is debatable.

Finally, I should mention there is also an intuitive distinction between "organic" or *natural* complexity and "artificial" or *formal* complexity. I would suggest natural complexity as an academic approach looks to study a complex system which iterates over time and evolves—it is organic. Such an approach would be more productive toward the biological or social sciences for example. The aim here is to study formal complexity which like any classical branch of mathematics assumes a complete existence of the system of study in advance. There is nothing to evolve because everything is already known (in theory at least). Formal complexity is existential and descriptive rather than constructive and prescriptive.

With that said, it is my own personal belief that defining complexity in a formal realm is not a zero-sum game either—it is not somehow in competition with natural complexity. Many mathematical models of the natural world afterall begin as formal descriptions, which are then turned into prescriptive data. Real life observed data tends to be messy, but is still usefully

[3]an informal version of it.

approximated by such models. At the very least, even if the formal model I present here is not directly translatable as a whole, I would like to think some of the embedded constructs used in its explication may themselves be launchpads for translatable ideas and lines of future research.

Design

At this stage we're more interested in building up an intuition for complexity with actual examples before we try to derive any formal definition. We do this by means of case studies.

case study: regular languages

We start with an example from automata theory called *regular languages*:

$$\{a, b, aa, ab, ba, bb, aaa, \ldots\}$$

here our context space is this set of all strings of combinations of the letters 'a','b'. We then ask: Is this context space complex?

Trick question! Is it complex relative to *what*? In such a case I would delve even further into our compiler theory and introduce an alternate representation called a *regular expression*:

$$(a|b)^+ \quad \to \quad \{a, b, aa, ab, ba, bb, aaa, \ldots\}$$

So what are we looking at here? If you're not familar with regular languages and regular expressions, the easiest way to say it is we have a context of infinitely many strings of the letters 'a','b' and we have a shortform way to describe the patterns within. This is to say, we have the semiotic space starting with atomic elements: $\{a, b\}$, as well as a combinatorial space (of all possible combinations) of n-tuples of these atomic elements:

$$\text{Seq}(\{a, b\}) \quad := \quad \bigcup_{n \in \mathbb{N} \setminus \{0\}} \{a, b\}^n$$

reiterating this, we have a set of "primitives" along with effective grammatical rules for combining these primitives to form our semiotic space, which is a good start, but our interpretation is unfortunately as of yet incomplete. We still have the media space to look after, so how do we interpret it here? We have our paradim pattern:

$$(\underset{1}{\cdot}|\underset{2}{\cdot})^+$$

which reads: apply set union to $(\underset{1}{\cdot})$ and $(\underset{2}{\cdot})$ and then non-deterministically apply concatenation to repetitions of itself (which is to say create arbitrary length strings of its elements). We have to massage our interpretation a little in the presence of our semiotic space: A collection of arbitrary length tuples rather than arbitrary length strings. In this case the solution is simple, we just map our tuples onto the corresponding strings: And although we've made no real assumptions regarding the internal data structure of our strings, it's very likely we will be simply mapping our tuples componentwise onto the string positions which match.

Okay, I admit I may have lost you here. I've taken something pretty simply and have explained it in a complicated way. Bear with me though, that's why it's a case study: We cut our teeth on the simple examples so when looking at more complicated ones we'll be ready. Besides, even though it's overkill, if our model can't handle something simple such as this, it's probably no good to begin with. In any case, if it hasn't made much sense yet, please continue onto the follow examples, they might.

Before that though, I should add: Another way of looking at this is to say we have *compressed* a much more complicated infinite set into a much simpler two element signifier set as well as a media pattern to show how those signifiers are related to the context. To reiterate, we have shown this regular language context space is not in fact complex relative to our regular expression—we are able to describe it perfectly in a much simpler way. In this case our semiotic + media space pair is called *equivalently complex* relative to their context space given we are able to represent it fully. At least at the intuitive level anyway.

case study: regular expressions

Regular expressions again? We already looked at them.

Yes, but our example was quite simple. Let's look at the slightly more complicated regular language:

$$\{b, ab, ba, aab, aba, baa, aaab, aaba, abaa, baaa, \ldots\}$$

if you'll notice we still have combinations from the atomic set $\{a, b\}$ but we now lack the full spectrum of the combinatorial space like before—here some such combinations are missing. Our corresponding regular expression is:

$$(a)^\star(b)(a)^\star \quad \to \quad \{b, ab, ba, aab, aba, baa, aaab, aaba, abaa, baaa, \ldots\}$$

which reads: A string of zero or more 'a's followed by exactly one 'b' followed again by zero or more 'a's.

So in this case, as with the previous case study, you can interpret us combinatorially creating our semiotic space starting with an atomic set $\{a, b\}$ along with grammatical rules of combination (n-tuples), but in this case before we map our semiotic space elements to our context space (as in the previous example) we need to *filter* our semiotic space, we need to restrict it. A word about terminology: In the case of the positive, I prefer to use the word **sifter** rather than "filter", as we then focus on what we're keeping rather than removing (the negative). This sifter + map combination is then our media space.

To reiterate: So far our complexity comparison requires:

1. semiotics:
   (a) a set of atoms,
   (b) a grammar of combinatorial rules (to build the full space),
2. media:
   (a) a predicate sifter (which narrows the semiotics),
   (b) a mapping (relating the remaining semiotics to the context).

case study: exceptional regular languages

Going back to our first case study, what if we complicate its regular language slightly by adding the symbol 'c' to its set:

$$\{c, a, b, aa, ab, ba, bb, aaa, \ldots\}$$

in that case our original regular expression no longer perfectly describes this set:

$$(a|b)^+ \quad \not\to \quad \{c, a, b, aa, ab, ba, bb, aaa, \ldots\}$$

because the character 'c' is inaccessible relative to our combined semiotic and media space.

This then is our first example of complexity.

My question to you the reader is: Does this *feel* right? Does this satisfy our intuitive understanding of complexity? Such a question is the motivation for the following two definitions:

Absolute Complexity (v0.1): A context space $\mathcal{C}$ with respect to a semiotic + media space pair $(\mathcal{S}, \mathcal{M})$ is absolutely complex if there is at least one object $o \in \mathcal{C}$ which is inaccessible by $(\mathcal{S}, \mathcal{M})$.

though I have not formally defined the relationships, it should seem clear that a complexity space would be a triple $(\mathcal{C}, \mathcal{S}, \mathcal{M})$ satisfying so far the above summary intuition.

Before moving onto our second definition, I should like to add some notation to aid in clarifying its concept. I have informally pointed to the idea that given a representative pair $(\mathcal{S}, \mathcal{M})$ there are expected to be elements of $\mathcal{C}$ which are accessible, as well as the fact that there may be elements which are inaccessible. To represent this in a shortform way, I will respectively write

$$\mathcal{C}_{\sigma(\mathcal{S},\mathcal{M})} \quad , \quad \mathcal{C}_{\phi(\mathcal{S},\mathcal{M})}$$

to respectively mean the accessible elements (σ), and the inaccessible elements (ϕ) of $\mathcal{C}$. Of course if the setting is clear, I will shortform it further as

$$\mathcal{C}_\sigma \quad , \quad \mathcal{C}_\phi$$

Relative Complexity (v0.1): A context space $\mathcal{C}$ with respect to a semiotic + media space pair $(\mathcal{S}, \mathcal{M})$ is relatively complex *under a given measure* m if

$$m(\mathcal{C}_\sigma) \quad < \quad m(\mathcal{C}_\phi)$$

It is this concept of *relative complexity* which is explored through the remainder of this essay. Although the reader will be quick to note it is already problematic as we have introduced an assumed *measure*. Keep in mind at the time of this writing, I use the term "measure" loosely—which is to say I do not specifically mean the known and well established measures such as *Lebesgue* or any others within real analysis.

case study: Cantor's diagonalization argument

For our final case study, we look at trying to model the real line using the natural numbers.

Here, if you know your set theory, our semiotic space being the natural numbers starts with the atomic set $\emptyset$ and the grammatical rule: $n + 1 := succ(n) := n \cup \{n\}$. The natural numbers then being the union of all such sets, identifying $0 := \emptyset$.

I will assume the reader has seen Cantor's diagonalization argument proving the difference in cardinality between the natural numbers and the real numbers. Skipping the proof here, I will summarize its interpretation in regards to our interests: We have our semiotic space (as described above), our sifter predicate is the identity (meaning we do not restrict our combinatorial semiotic space in any way), and then we show there does not exist any surjective mapping from the natural numbers to the reals. The complication here if you'll notice is that this deviates from our above intuitive understanding with respect to the mapping stage. In our above regular language / regular expression examples, we were only concerned with exactly one (natural) mapping, but here we allow for more than just one—and still show complexity exists—thus inducing a "boundary" level version of absolute complexity.[4]

How do we reconcile this case study with our previous intuitive summary? We extend: Instead of a "mapping" stage as before, we can now say we have a "bundling" stage, where the idea of a bundling is kind of like an inventory of possible mappings. More formally:

1. semiotics:
   (a) a set of atoms,
   (b) a grammar of combinatorial rules,
2. media:
   (a) a predicate sifter,
   (b) a bundle.

here our *bundle* is in the set theoretic sense, which is to say a collection of mappings all sharing the same domain and range—being the semiotic and context spaces respectively.

In this case our two definitions of complexity need to be updated just a little as well:

Absolute Complexity (v1.0): A context space $\mathcal{C}$ with respect to a semiotic + media space pair $(\mathcal{S}, \mathcal{M})$ is absolutely complex if there is at least one object $o \in \mathcal{C}$ which is inaccessible by $(\mathcal{S}, \mathcal{M})$.

Again, this is not formal, and the wording here hasn't changed from the previous, but the underlying assumption now is that our mapping stage is now a bundling stage. As well, no matter which function within the bundle is chosen to map, there will always still remain an object (even if it differs for each chosen function) which remains inaccessible under that mapping. Under this definition, the real line is complex relative to the natural numbers within Cantor's understanding.

Relative Complexity (v1.0): A context space $\mathcal{C}$ with respect to a semiotic + media space pair $(\mathcal{S}, \mathcal{M})$ is relatively complex *under a given measure* m if

$$\sup m(\mathcal{C}_\sigma) \quad < \quad \inf m(\mathcal{C}_\phi)$$

this is the definition which has changed a little.

Here our "sup" is the *supremum* and our "inf" the *infimum.* The interpretation being now with our bundle stage, each mapping within the bundle partitions the context into its accessible and inaccessible parts. Thus we have a collection of accessible sets as well as collection of inaccessible ones. Continuing this extension we then have a collection of measures of accessible sets as well as a collection of measures of inaccessible sets. In this case we then take the supremum and infimum—they form duals given that our accessible and inaccessible sets are compliments of each other. To summarize: If our best possible mapping within the bundle is such that it measures the accessible elements less than the inaccessible ones, our technology space should qualify as relatively complex.

The motivation behind the idea of relative complexity by the way comes from *energy conservation*: A truism of all life is that energy must be expended to acquire more energy. If the energy expended exceeds that of the energy acquired the effort was not generally worth it. Life is clever and hedges its bets—it stores previously saved energy in reserves to offset the above mentioned situation of overspending (which does happen time to time), but on average if life is to continue then the energy gained needs to exceed that of the energy expended.

Relative complexity then—within its definition—assumes energy conservation within the system as a whole. How does this relate to complexity? Again, it goes back to compression. I've noticed things which are considered beautiful are generally so

[4] not only is it complex relative to one specific mapping, it's complex relative to many, or in this case all.

because they tend to compress well. For example symmetry is often considered necessary for something to qualify as beautiful. Given the classification of symmetry groups, it is easy to see regardless of which type of symmetry (eg. reflective, translational) what they all have in common is they compress well. It takes less energy to represent more. If you look at other qualifiers of general aesthetics, it can be argued the recurring theme across their subjectivity throughout culture and other diversity is again the fact that in some way they compresses in a "nice" way.

As for its value in our definition of relative complexity: We try to compress a context using a technology space, and with it we may be able to compress part of the context, but if the energy lost due to the act of compression is greater than the energy gained, then our attempt in its simplest form—intuitively—is ineffectual. This is to say our context relative to our representation is sufficiently complex.

addendum

Before we move on, I would like to add a quick note on potential future terminology. In particular, we may need to make a distinction in technology spaces in regards to their media spaces. For example with our above regular expressions, notice how our media space stage has two parts: Sifting, then Bundling, but the expression used to represent this is implicitly used for both. This is to say we don't—and maybe we can't—separate these two parts. In the case that we can't, I would call the media space within such a technology space *intangible* and otherwise *tangible*.

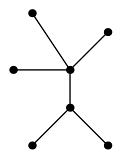
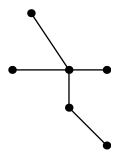
I mention this as nothing more than an addendum here because it's not the most pressing detail at this stage of research, but worth noting as a potential future path.

semantic generators

Now we are now ready to talk about *value systems*. As it turns out, complexity might be an appropriate way to model general semantics, the basis of this claim being the idea of a semantic generator.

A semantic generator then, intuitively to me, is kind of like: The scraping together and friction amassed by strong wills trying to unite. If a technology is relatively complex, it means its representation is not a perfect fit to its context—something is always left out—but if we continue anyway to try to make it match knowing it never fully will, new meaning will constantly be created.

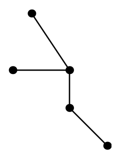
As for something closer to a practical example, let me explain with the following visuals:

For all intents and purposes you can consider these to be: Abstract patterns. Graph theory (the basis for these visuals) I suspect is suitably general enough for that purpose, as you can consider its *vertices* (dots) as objects and its *edges* (lines) as unnamed relationships between those objects.

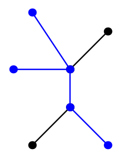
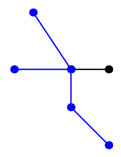
My own practical example actually comes from computing science, in the attempt to refactor or abstract existing code (to improve performance, maintainability, reduce errors, etc). You might look for common patterns within your own code, and for example consider blocks of code as vertices and the algorithmic instruction flow as relationships, or however you want to think about it. The point being, however such abstract patterns are interpreted, we would end up seeing two similar but not quite identical patterns, and would then be required to ask: Do we refactor?

In this case if we do, there are two possible ways. The first being kind of like the "greatest common divisor" (gcd) or even "set intersection" paradigm:



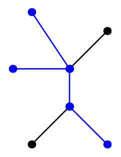
this is to say we look for what's common.

And upon determining what's common, maybe we can replace it with a single function or module and connect it to the remaining parts:

here the blue is what's common, the black being the remainder.

Or we can find a structure/function which extends both patterns. This is more along the lines of a "least common multiple" (lcm) or "set union" paradigm:



this is the approach which leads to a complexity of pattern abstraction.

How? Going back to relative complexity, you can think of it as: We use the gcd as the semiotics, and attempt to use it to represent the two patterns as context. Naturally this gcd has Naturally the intersection maps onto itself, so what's left are all the parts belonging to what we'll call the *symmetric difference* (borrowing terminology from set theory). Assuming we have a measure, we then test if the accessible parts are greater than or equal the inaccessible parts, and if so our abstraction was worth the cost.[5]

So how does this create complexity? We take this process to its limit: Finding two patterns and refactoring is one thing, but what if we had started with three similar patterns and wanted to refactor? What if we had started with one hundred? The obvious answer being so long as our "least common extension" is worth the cost: If what's common is greater than or equal to what's different, than it's worth it. But what if that's not the case, what if it is in fact relatively complex?

Then we have to choose which patterns to allow into the boundary of our *semantic module*. How do we decide that?

In the natural world, I think this often enough happens by first come first serve, until a spillover effect is reached at which point the module becomes stable. Though in an organic context politics and art and evolution also allow these *constructs* to disassemble and reassemble periodically until they're sufficiently efficient within their living environment, which is to say these modules remain stable in local time but evolve for improved performance relative to their ecologies.[6]

Otherwise, in order to decide which patterns make the cut—when conscious design is more active—we would need: A value system!

The complexity then of modular design requires we generate semantics (our value systems) to decide how to group things together beyond their basic composition. This leads to concepts such as *privileging*.

Application

As this is intended to be an introduction to complexity, our theory selection though short, is now over. As for applications, we will look at a single example from computing science: The *list map operator* within the functional programming paradigm.

My main motivation for developing this theory of complexity comes from the need for me to express all the patterns I see in math and the world which I as of yet have a language to express. Beyond that though, the list map operator is a clear representative application, a muse if you will, to further this research further still.

For those not in-the-know, the list map operator:

$$(\ \mathrm{map}\quad f\quad \mathrm{list}\ )$$

takes a function f and applies it to each component of a given list, returning a new and resultant list. As an abstract pattern itself it is seen in pretty much every facet of functional (and other) programming.

In my own application, I have been building a C++ library, and have great need of a low-level, well thought out, well designed map operator. Of itself, that's easy enough, but I have, in crafting my library, coded many small variant forms of this operator, and have reached a point wondering if I should refactor, and if so, how best to do so?

[5]Here the cost is measured in terms of what structurally differs rather than the energy required to create the compression in the first place—the assumption being the energy required to compress was offered by the coder as a sunk cost.

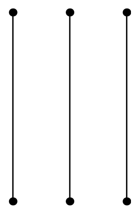
[6]Though only briefly carved out here, this is a serious future direction in research and application of this theory.

This is where complexity theory of this nature can help. The first best practice implied by our theory can be interpreted as saying: Even if I do refactor, there's only so many variants I can compress before we reach a point of *complex returns.*[7]

The coder's process here is actually pretty standard, so I'll go through the predictable steps with you:

option 1:

The first option is to do nothing. To simply leave these algorithms as is, having their different names, and different optimized contexts:



here you can think of each line segment as a list map operator algorithm similar (some overlapping code) but otherwise different from the other two.

One such example arising in practice is when you create two distinct map operators, one for a singly-linked list and one for a doubly-linked list. In such a case, the algorithms are identical in terms of the code used to iterate and map over the lists, only the details in node construction differ (singly-linked lists you allocate less space, and only link forward).

If these were the only two similar list map operators in my library, I might take this approach, but the exact design I've aimed for has resulted in hundreds/thousands of such operators. How so many?

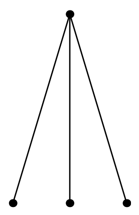
Batching. If you recall, batching as a best practice occurs when you have 4 people driving to the same place: Instead of each driving their own cars they all take the same one together. If a list is sufficently long, it becomes worthwhile to batch a few of the most common operations—it becomes worthwhile to reduce parse cycles. For example, given my code is in C++ and I have refined control over allocation and deallocation of memory, I may wish to delete the input list after applying the map operator to it. In that case I can reparse the list to delete each node, or if I know I'm deleting it immediately after the list map, why not delete as the list map itself is performed? It's already iterating over the list after all.

So basically, I have several "configuration variables" such as {apply deallocate, omit deallocate} for the input list, or {apply allocate, omit allocate} for the output list (in this case the output list might already exist and we're just mutating it). Or I can apply this map operator over 3 varieties of basic lists: {singly-linked, doubly-linked, array}. My own library actually has several other variant configuration parameters, so as you can see the small branch variations build up quickly.

This first option then, given my coding design situation, in this case becomes impractical.

option 2:

The second option is to keep the code for each variant separate as before, but to create an interface so one can "dispatch by configuration", this way we can semantically group these functions together, and reduce the namespace (which would not scale anyway) as we get to reuse the exact name for all:



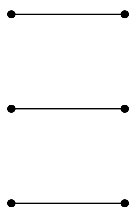
If you don't mind the increase in memory required to retain all possible variants, performance-wise this approach is ideal as it should have the least overhead while keeping a clean design. As well, given that this is a library, which in most programming contexts means "lazy" linking (add the code to the compiled program only if used), the memory increase may in fact be minimal.

Unfortunately in my case, my library is primarily a template library, and my trustworthy hardworking compiler still tends to choke on the several thousand template variants it has to preprocess (the vast majority of which it discards). I've found ways to minimize compile time by modularizing the template source code itself (the compiler only reads source code that's actively linked to through macros), and it helps, but in the long run it only scales to a point: It's not a permanent solution. In the world of industry, for deployment purposes, slower compile-time might be acceptable as you only need to do it once, but for prototyping purposes it's less than ideal.

[7]I allude here to the wisdom of *diminishing returns.*

option 3:

So, instead of vertical modularization (separate algorithmic streams), and instead of dispatched vertical modularization (separate algorithmic streams with a single interface), we can instead do horizontal modularization[8]:



here we compare algorithmic streams, refactor what's the same, and encapsulate what's different with control flow grammar. This approach is less performant, as the block modules within a single algorithm increase overhead and control flow calls which decrease performance overall, but is lightweight memorywise, and quick to compile. If nothing else, ideal for prototyping.

And yet, after having reached this satisfactory solution, we still have to ask: Do we refactor all variants? Or do we cluster the variants into a few representive classes? Such a design decision for me at first was more of an artistic decision, but with this theory of complexity now in my back-pocket becomes analytic in nature.

addendum

I admit, the weak point in my theory is this is still at a research level: I do not have precise best practice guidelines here. I suspect even if this way of doing things even takes hold at all in industry, different production teams will have slightly different analytic protocols anyway depending on their culturals, philosophies, and intentions at the time. There's no actual best performance measure in the general case to determine how many refactorings would be allowed before complexity ensues. There are many good metrics, but it varies on context after all.

I will add though, if you are at all curious, for my own library, I took a hybrid approach—slowing down my own production greatly I might add, though given the low-level importance of this module within the larger library as a whole it is worth it.

I ended up using both options 3 and 2:

With option 3 I can prototype, which is wiser given my library as a whole hasn't stabilized yet. When enough of my library has stabilized, I can take some frequency statistics, and use the option 2 inlined code to replace the most common variants throughout the library as a whole, thus reinforcing performance at frequent and/or critical points. Furthermore, regarding the prototyping version itself, I've reinforced its performance by vertically modularizing the internal loops (as you should not be making a lot of function calls within loops, that obviously adds up in cost). It's a worthy tradeoff: Although it's an ad-hoc optimization (it breaks with the clean narrative conceptual design), it is a terminal optimization at that.

Conclusion

That's about it. I hope the preceeding was sufficiently clear, if not yet rigourous.

As for the rigour, I have not formalized the definitions here because the reality is this theory of complexity is still in its early stages, and I should not like to fully commit to any one design until the larger landscape has materialized and stabilized. With that said, you should not expect the ideas here to generally change, only to be refined.

Then again, I haven't even mentioned other ideas I've been researching such as "recursively compressible" (related to information theory) or "modular divergence" (historical treatments of system evolution) with applications further to computing science, not to mention culture theory. I'll leave you with that.

Thank you. Pijariiqpunga.

[8]Here you can think of these horizontal lines as lines of source code, and so we are thus modularizing blocks of algorithmic instructions.