A brief overview of the SPECTRALSEQUENCES code

Hood Chatham
hood@mit.edu

2017/7/26

---

1 Introduction

The basic idea is comparable to a more elaborate version of `tikzcd`. We plan to draw multiple copies of a spectral sequence, so the commands are not issued inside of a picture environment. Instead, all information must be stored to draw later. The `\sseq@savedpaths` is responsible for storing the collection of objects of the spectral sequence.

The user defines "classes" (which correspond roughly to generators of the spectral sequence E_2 / E_1 page). From the point of view of the programmer, a class is an object with a death page (either infinite or finite) signalling on which page it is the source or target of a differential and some pile of information on how to draw it. When the user calls the `\class` command, it makes a new class object with an appropriate name and stores the information to tell SPECTRALSEQUENCES how to draw the class later, and sets the page to be infinite. A macro informing spectralsequences about the class is added to `\sseq@savedpaths` so that it can be drawn later if appropriate.

The `\d` command is similar – we set the page of the source and target to be the current page and also the pages of any relevant structlines, store the page of the differential and the data needed to draw it, and add the differential to `\sseq@savedpaths`.

The `\structline` command saves code to draw a line, and also figures out what page it should die on based on the smaller of the two death pages of the source and target. If the source and target haven't been killed yet, it hooks into them so that `\d` knows to set the page of this structline too.

Now to draw a page of the spectral sequence, we start up a `{tikzpicture}` environment, draw some axes, walk across `\sseq@savedpaths`, and for each class, we check whether the death page of the class is greater than or equal to the current page and if so we draw a corresponding pgf/tikz node. Similarly, we draw the differentials that lie on exactly the current page, and the subset of the structure lines with death page later than the current one.

This is the core idea of the package. In addition to these core features are many important reusability features. For instance, we need a way to define reusable user macros that draw collections of classes and we need looping constructs.

So, having heard this simple description, a reasonable question to ask is: why does it take in excess of 8000 lines of TeX code to accomplish these simple tasks? There are a few reasons. Firstly, the package handles a lot of use cases and has a quite a few different features and there is a large amount of intrinsic complexity to getting these things right. Another huge reason is that TeX is a completely unreasonable coding environment. This is responsible for a good deal of the complexity in `sseqparsers.code.tex`, for example.

Perhaps the biggest single contributor to code complexity is performance. Ti*k*Z is very robust, but it is not fast. SPECTRALSEQUENCES draws roughly cubically many classes in large diagrams, and my larger test examples have as many as 5,000 classes. If each class takes an extra millisecond a second to draw, that comes out to an extra five seconds of compilation time. Because of this, unless the performance critical drawing code is super optimized, the package will be way too slow on large diagrams. Most of the added complexity due to performance considerations is contained in `sseqdrawing.code.tex`, but the hard parts of the key value code are all related to performance too.

The last major source of complexity is the robust error handling code. Most LaTeX packages have crappy error handling and result in inscrutable errors even in relatively reasonable use conditions. There are a two main reasons for this as far as I can tell: LaTeX doesn't have good error handling tools and most people who write LaTeX packages are not primarily programmers and have other priorities than ensuring that their packages have clean failure modes. The files `sseqmessages.code.tex` and `sseqforeach.code.tex` are entirely devoted to error handling, and almost all of the complexity of `sseqmacromakers.code.tex` is tied up in error handling too.

2 Load Store

The file `sseqloadstore.code.tex` defines two storage systems. The point of these systems is to ensure that unrelated spectral sequences do not interfere with each other. One system is faster and relatively storage intensive, and it is for macros that contain global information about the spectral sequence. These macros have default values which can be changed using `\sseqset`. The other system is for storing the data about the spectral sequence objects (classes, differentials, structlines, etc). This file is short and simple, but the seemingly innocuous features provided here lead to huge simplifications in much of the rest of the code.

The first system works by making a list `\sseq@storelist` of the commands that need to be stored. It defines commands `\sseq@storecmds` and `\sseq@getcmds` which iterate over `\sseq@storelist` and copy each command between `\sseq@whatevercommand` and `\sseq@whatevercommand@@@`⟨sseqname⟩. Each command also has a default value stored in `\sseq@whatevercommand@@default`. This default can be changed by the user using `\sseqset` outside of a spectral sequence. There is also a command `\sseq@storelist@setdefaults` which loads the default values of all of the commands in `\sseq@storelist`. At the beginning of a `{sseqdata}` or `{sseqpage}` environment, if the spectral sequence is new then `\sseq@storelist@setdefaults` is run, and otherwise `\sseq@getcmds` is run. At the end of the `{sseqdata}` environment, `\sseq@storecmds` is run.

The second system basically is just semantic sugar for indexing into a command that has the name of the current spectral sequence inserted into it: the command `\sseq@obj{some name}` just expands to `\csname sseq.\sseq@thename.some name\endcsname`. This allows different spectral sequences to have different name spaces. A large number of commands to manipulate this are provided. All changes to objects are made global, so at the end of a `{sseqpage}` environment, any changes that have been made must be reverted. The command `\sseq@cleanup@obj` checks if we are in a context where changes will have to be reverted, checks if an old value of the current object has been stored yet, and if neither of these are true, it adds reverting the current object to a "cleanup" command to be run at the end of the current environment.

3 Messages

The file `sseqmessages.code.tex` defines the error handling code. It uses the delightful l3msg infrastructure which separates message content from the position the errors are triggered in and makes much more attractive error messages. One deficiency for my purposes in l3msg though is that it doesn't have very many hooks. My package knows a lot about the circumstances under which most errors trigger and wants to tell the user about this. The basic idea for this approach came from `tikzcd`. At the beginning of the `{tikzcd}` environment, the `\errmessage` command is patched to add an annotation explaining what cell in the matrix the error occurred in. This is because `{tikzcd}` has a two-pass algorithm, and so errors can occur far away from the code that triggered them.

Anyways, I have a fairly complicated pile of extra information to add into the picture. Firstly, I reconstruct what the user said as closely as possible. This is done inside of `sseqmacromakers.code.tex` and `sseqmain.code.tex`. I also assemble a stack trace that indicates all loops and user macros and all loop variable values.

I assemble all this context information and a little more, and then make an annotation message accordingly. All of the many cases here are handled using a pile of horrible spaghetti code.

The annotation setup code is not expandable, so it cannot happen as the error is written to stderr. We need a setup hook before the error is triggered, we need a hook inside the body of the message, and we need a cleanup hook after the error is done to remove any local changes the setup code made. I make a copy of `\__msg_error_code:nnnnnn` and add these two hooks to the beginning and end. The annotation itself is prepended to `\__sseq_hooked_msg_see_documentation_text:n` which is conveniently located inside of the message text.

The other interesting thing in this file is my code for handling `\end` mismatches. Because of the need for nested environment definitions, LaTeX only checks that the environment being `\end`ed matches the current environment after running the `\end` code. However, the code for `\end{sseqpage}` tries to draw a picture, and drawing this picture depends on a large amount of state. If the current environment isn't `{sseqpage}`, none of this state is actually present and so hundreds of incomprehensible low level errors happen before the check end code is run and LaTeX gives the environment mismatch error. The fix for my package is to run my own checkend code, which is pretty simple. See my tex stack exchange question for more details.

4 Loops

The file `sseqforeach.code.tex` is responsible for defining looping constructs. It's mostly pretty simple. The first half defines `\DoUntilOutOfBounds`, which repeats some block of stack based code until the top of the stack is out of bounds. The most complicated part of this is the descent check – it periodically checks that the top of the stack has moved towards the boundary of the page to prevent infinite loops.

The other part of this file patches the `\foreach` command from Ti*k*Z in order to make it inform SPECTRALSEQUENCES what the user said for the foreach loop (as best as I can approximate) and what the loop variables are. This is for error messages – the "what the user said" part will go into the stack trace, and the loop variables will be listed.

5 Macro Makers

The file `sseqmacromakers.code.tex` defines commands to define commands. It is built on top of xparse but adds a large number of features specific to my package. Unfortunately, it depends on xparse internals and xparse is not yet stable. None of these dependencies are mission critical, but they significantly increase usability. I have attempted to make fallback code that will work even if incompatible xparse changes are made, but this is incomplete.

Like Ti*k*Z, SPECTRALSEQUENCES defines a bunch of commands that only make sense to use inside of certain environments. Thus, we define the command `\class` as `\sseq@class` and at the beginning of the `{sseqdata}` or `{sseqpage}` environments, we run a command `\sseq@installmacros` that does a bunch of things like `\let\class\sseq@class`. I want to use xparse to define `\sseq@class`, but the problem I run into is that xparse stores the name of the command being defined and uses that name in certain error messages. These error messages would refer incorrectly to `\sseq@class` instead of `\class`. The fix for this is pretty boring – it just involves redefining macros back and forward. Anyways, I make a command `\sseq@DeclareDocumentCommandAs` that defines a command using xparse that gives errors as if it were named a specific other command. I use this for defining most of the user-facing macros.

The rest of the file is responsible for defining `\NewSseqCommand` and `\NewSseqGroup`. These commands are relatively simple in principle – the point of `\NewSseqCommand` is almost already accomplished by xparse already. Most of the complexity is introduced in order to improve the error messages that SPECTRALSEQUENCES produces. In particular, I want to reconstruct as closely as possible what the user said when they ran the command. This way, if an error occurs nested several commands deep, or inside of a loop, the user can still get a decent idea about where the error is coming from.

6 Keys

There are different options that make sense for the `\class` command versus for the `\d` command, and some options that do make sense for both need distinct handling. However, there are also overlaps. Thus, the key value system is naturally a tree structure. pgfkeys has support for “directories” which proves extremely valuable. Each directory has an unknown option handler, which by default throws an error. I have a command `\sseq@passoptionto` that hands the keys to the parent directory.

SPECTRALSEQUENCES is a two-pass algorithm. On the first pass, it creates the data structures necessary to encode the spectral sequence, and on the second pass, the spectral sequence is actually drawn. The second basic design feature of the SPECTRALSEQUENCES key value system is that keys need to be able to do different things on the first and second pass. This is accomplished very simply – I use two wrapper macros `\sseq@options@firstpass` and `\sseq@options@secondpass` for code that should only happen respectively on the first and second pass. To use them, I basically just define `\sseq@options@firstpass` as a no-op and `\sseq@options@secondpass` to add its argument to a macro to be stored for later use at draw time. However, in some circumstances I need to redefine these two hooks, either to delete their argument or to evaluate there arguments.

Once we have this basic design, we can implement many of the basic features of SPECTRALSEQUENCES. There is a problem, which is that I want to mix Ti*k*Z keys with keys that my package defines gracefully. However, Ti*k*Z keys execute code that assumes that we are in the middle of drawing something. One early design approach I used was to define my package's unknown option handler to do validation only on Ti*k*Z keys on the first pass, and then store the keys and run `\pgfkeys` again to evaluate them at draw time.

The third major design goal is to allow “styles” that change the display of logical collections of classes. That is, the user should be able to say something “permanent cycles = red” and change all permanent cycles (as identified by SPECTRALSEQUENCES) to red. Originally I implemented this by storing the keys that the user provided and using `\pgfkeys` at draw time to evaluate them each time I made a class. Here I ran into a problem: `\pgfkeys` is very slow. When I used this design and compiled a 5,000 class spectral sequence, saying `classes = {red, red, red}` took over a second longer than not saying it. This was obviously an unsatisfactory state of affairs. The solution to this problem sped up the package tremendously, but at the cost of a large increase in design complexity of the key value system.

The solution of course is to only run `\pgfkeys` on any given key once and store the result, rather than repeatedly using `\pgfkeys` on the same input. In order to do this, I needed to hook into `\pgfkeys` and change it so that every time that it would normally evaluate code, it should just store it into `\sseq@savedoptioncode`. Conveniently enough, there are only two places where `\pgfkeys` emits code, so this basic idea is easy enough. The problem is that many keys execute recursive calls to `\pgfkeys` that are state-dependent. Normally, the key code is executed instantly, so it can reasonably expect the state it needs to be preserved. However, if I defer the body of a key that makes such a recursive call, pieces of the necessary state are lost before the code is executed and it breaks. Thankfully, all such state-dependent recursive calls are made using the command `\pgfkeysalso`. So the fix is to first walk across the key code and find any instances of `\pgfkeysalso` that occur in it, and run these `\pgfkeysalso`s. Once this recursive expansion has taken place, the resulting code is independent of the state of the key value system's state and is safe to store for later use.

7 Main

The `sseqmain.code.tex` defines the main environments and user commands. The logic is pretty much as described in the overview section, and nothing super notable happens beyond that. There are a lot of details to get right, in particular a lot of the mess in this file is due to the complicated interfacing with `sseqkeys.code.tex`.

8 Parsers

The file `sseqparsers.code.tex` is a bit of a grab bag. The code is grouped by structure more than by content, hence the name. Most importantly, it has all of the coordinate parsers that define the coordinate syntax. This file uses some amount of LaTeX3 programming, though it is a bit of a chimera. There are three main reasons I did not use LaTeX3 programming more broadly or consistently. The first is inexperience: I simply haven't used LaTeX3 much. The second is that LaTeX3 and TikZ do not get along all that well. This is fixable but a pain. The third is that LaTeX3 has a lot of fancy convenient robust macros, but I generally need sloppy fast macros. If I can write a faster macro that adds a large number of weird preconditions, for the purposes of SPECTRALSEQUENCES that is probably better.

One very good reason to use `\ExplSyntax` in this file is that it gets rid of a lot of space hazards. In this file, I really just make use of the variants of `\exp_args` and `\exp_last_unbraced`. These macros are tremendous innovations that greatly improve expandable macro writing.

One major annoyance is the need for `\sseq@ifintexpr`. This is a sanitizer for `\numexpr`. I don't understand why the ε-TeX source doesn't contain an input sanitizer! TeX does not support catch blocks or exceptions, so because we can't ask forgiveness we have to ask for permission. Anyways `\sseq@ifintexpr` was difficult to write and is slow. Together, `\sseq@ifintexpr` and `\numexpr` are still much faster than `\pgfmathparse` though, and the input sanitization only has to occur at the user input stage, so all of the internal uses of `\numexpr` don't pay this cost. Contrast the case for `\sseq@ifpgfmathexpr` which is my sanitizer for `\pgfmathparse`. All it does is redefine `\pgfmath@error` to throw away the error, set a flag, and quit out of `\pgfmathparse`. It was orders of magnitude easier to write.

There is a parser that does some manipulation on TikZ commands to process the options and the coordinates. The options processing is primarily for speed purposes, and the coordinate parsing I do because I want to make several changes to the way that tikz processes coordinates for the sake of better interoperability with the rest of the SPECTRALSEQUENCES feature set (these changes are unfortunately quite involved...). A third thing gained by this is that SPECTRALSEQUENCES can know that the path looks reasonably close to being syntactically correct at compile time (though I don't run much of the tikz parser, so there's only so much it can tell).

There are a few tricks in here that I think are relatively interesting. The one that I find the most interesting is as follows: suppose you have an expandable macro that needs error handling. The problem with errors in expandable macros is that the errors themselves are not expandable. My solution is to write the macro so that it either a) satisfies whatever the postcondition of the macro was supposed to be or b) is of the form `\protecterror{\someerror}`. Before the `\edef` occurs, the `\protecterror` is defined to expand to `\noexpand\protecterror\unexpanded{#1}`, so that it hangs around after being `\edef`'d. Suppose we are writing `\edef\theoutput{\somemacro{some input}}`. Then when you want to check for the error, redefine `\protecterror` to globally set some error flag, then say `\setbox0=`*`\hbox{\theoutput}`* in a local scope. Because `\hbox` evaluates its argument, this throws an error if `\theoutput` contains an error, and also `\protecterror` sets the error flag. However, `\theoutput` can have other random junk in it, which won't leak into the output. This solution doesn't depend on the error being in the top level of `\theoutput`, it can be in as deeply nested braces as necessary and doesn't require `\scantokens` hacking. It doesn't rely on any particular structure features other than the presence or absence of the error. Anyways, this is how I manage to have robust error messages with expandable macros. Obviously this isn't applicable to many circumstances, but in my code, my need for expandable macros is always in situations where I have a macro whose evaluation depends on temporary state, but whose result needs to be stored inside of a command (not an `\hbox`) for later handling.

9 Drawing

The most notable aspect of the `sseqdrawing.code.tex` is that significant chunks of it need to be highly optimized. The command that is used the most times in a normal spectral sequence is the one that prints a class. It is important to be aware of differences between the efficiency of different low level pdf commands. It is worth pointing out that I have fallen into the trap in some places of optimizing code that didn't need optimization.

In order to maintain all the features of Ti*k*Z while being as fast as possible, all drawing code is done in two ways – one way that is fast and uses low level PGF, and a second way that is slow and uses Ti*k*ZĊertain options the user asks for cause the fallback slow code to be run instead of the fast code. It is actually surprising to me how easy it has actually been to maintain this two-method code. It's unusual for the class drawing code to have to resort to the fallback, but if an edge is bent or has a label on it, SPECTRALSEQUENCES uses Ti*k*Z to draw it. Both of these are reasonably standard use cases, but hopefully they will only ever apply to a small percentage of edges in any diagram.

In order to build the relevant chunks of the fast code, I copied the Ti*k*Z main loop and deleted stuff until I got to the minimal subset of code that works for my purposes. One of the main enemies of speed in this section (other than the Ti*k*Z parser and main loop which I avoid by using PGF) is the `\pgfmathparse` command, which comes up a lot even in PGF code and is quite slow. As much as possible, I try to get away with using the ε-TeX `\numexpr` and `\dimexpr` commands, both of which are implemented in the ε-TeX binary and so are very fast. These two commands both have annoying limitations, and it can be difficult to accomplish certain tasks with them. However, it is worth it in interest of speed. In fact, `\numexpr` is actually preferable to `\pgfmathparse` for most of the code outside of the drawing file, because SPECTRALSEQUENCES deals primarily with integers and basic arithmetic `+-*/()` that `\numexpr` handles, and because `\pgfmathparse` has the unfortunate property of turning all integers into floats.

The other difficulty that arises here is that SPECTRALSEQUENCES needs to be able to handle very large diagrams. In a normal `\tikzpicture`, the distance from `(0,0)` to `(1,0)` is 1cm. If you use `scale=0.5` for instance, and use a coordinate `(3,0)`, then the order that multiplications are performed in is $(3 \times 1\text{cm}) \times 0.5$. The issue is that 600cm is an overflow, and this overflow occurs even if you have a very small scaling factor that would make that coordinate fit onto the page (which is probably a bit less than 600cm long). The solution is instead of applying a scaling transformation, I adjust the position of the coordinate (1,0), say to be 0.01cm. Then $600 * 0.01\text{cm} = 6\text{cm}$ leads to no overflow. This leads to various complications in the drawing code.

There's a second more problematic issue though. Suppose that I want to zoom in on the range from 600 to 700 on a very large spectral sequence. The scale I want to use to fit this to the page will cause an overflow. To fix this I subtract an offset between the minimum and maximum value of x from every single coordinate. This adjusts the range so that it actually goes from say 0 to 100, and then no overflow occurs.

The issue then is how to make this offset also apply to Ti*k*Z primitives. In order to do this I patch into the Ti*k*Z coordinate parser and every time it parses a coordinate add in the offsets in the appropriate place. There are a few spots where this goes awry and offsets are added to coordinates that are calculated relative to a another coordinate with the offset already calculated in (the `to` keyword is the main offender here), so there are second order fixes to prevent this.