Anatomy of a Scientific Research Paper (Part 1)

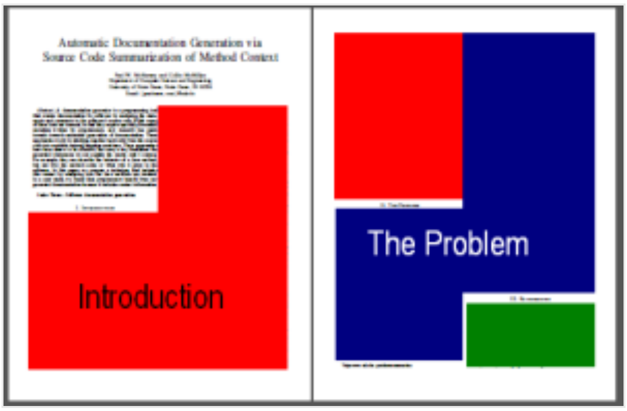
I was in school for 23 years. And the hardest thing I ever learned to do was write a measly 10 page research paper. I had my research results. I had built my tools. I had conducted all the necessary experiments. And yet the writing seemed impossible. Sentences came slowly, if ever. Every word was painful. I would do anything else, and chores became a blessing: clean my apartment, scrub the dishes, go to the dentist, anything but write.

I thought I might not be cut out for research. Maybe all this school was not for me after all. It's fashionable to quit nowadays.

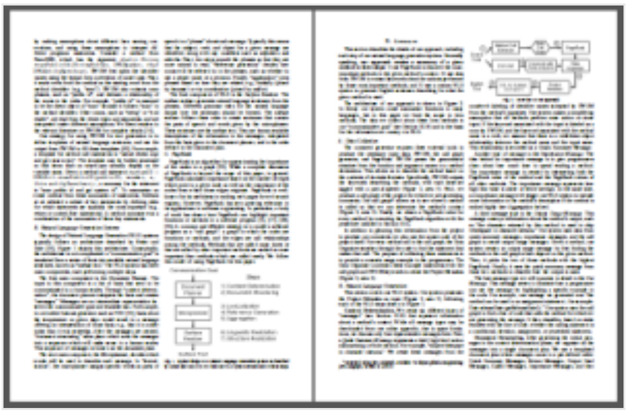
Nonsense. What was wrong was that I didn't know the anatomy of a scientific research paper. I didn't even know where to begin. If this sounds familiar to you, read on. Over the next few posts I will spell it out for you one \section{} at a time.

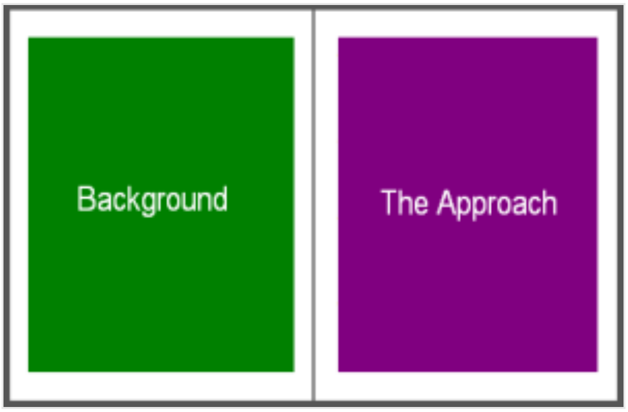
Let's start with a bird's eye view of a typical 10 page technical paper.

Automatic Documentation Generation via Source Code Summarization of Method Context


Automatic Documentation Generation via
Source Code Summarization of Method Context
Introduction
The Problem

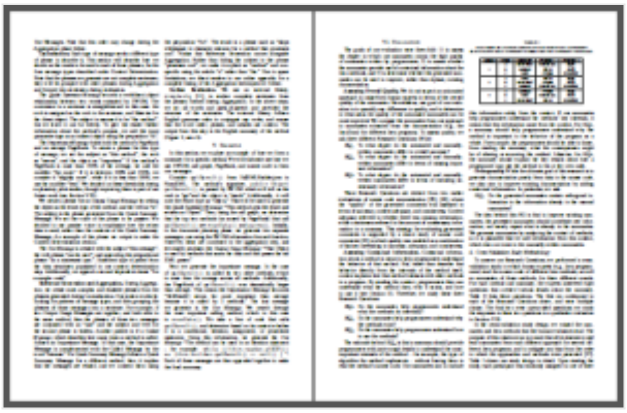
Pages 1 and 2. This is where you describe the problem you are solving. This is **not** where you describe your solution to the problem. You may hint at the solution in the last paragraph or so of the introduction. But a vast majority of the space on the first two pages should be dedicated to the problem. The introduction should give a general description of the problem domain. The "The Problem" section should give background, such as the source of the problem, the negative consequences of the problem, and the potential benefits to solving the problem. In other words: why is the problem important?

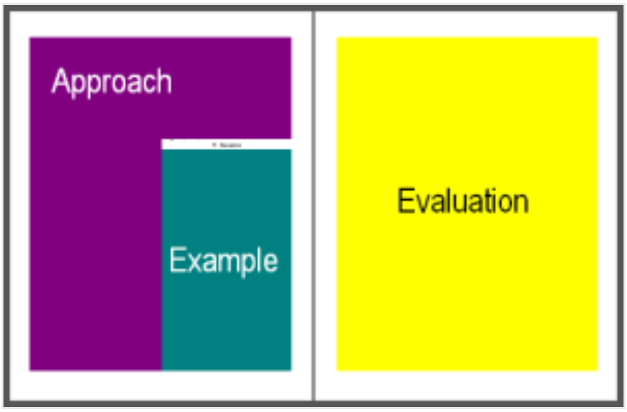




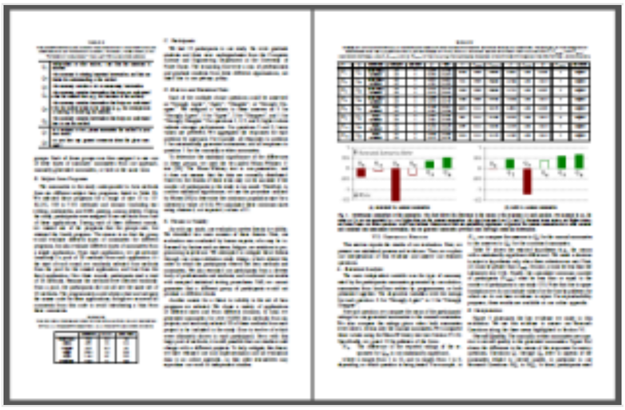
Next up are pages 3 and 4. Page 3 is the background. This means any tools and technologies that you rely on for your solution. This is **not** where you describe your solution. This is only where you give an overview of the most-important technologies you use that have been published before. If you are proposing a new feature location technique that combines Latent Semantic Indexing with Program Slicing, you should give a brief description of these in the background section. The background is not for rehashing the problem. And I repeat, the background is **not** for the solution. It is for the supporting technologies of the solution.

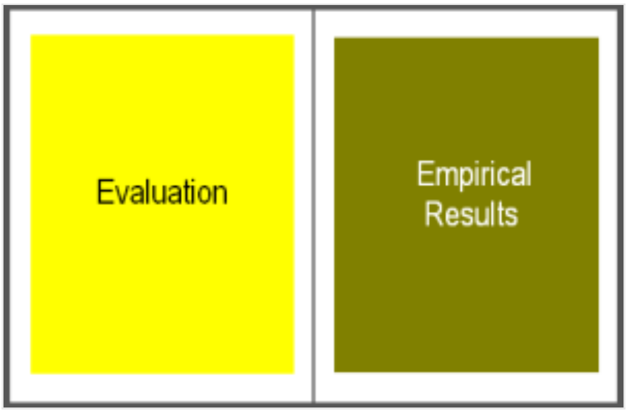
Finally on page 4 is where you start explaining your brand new, enlightened solution. Provide a picture to show the architecture. Make it 100% clear how you combined LSI and slicing. What, precisely, are the inputs and outputs of your solution? What parameters did you use? What design decisions did you make and what was the rationale behind each decision? Etc.



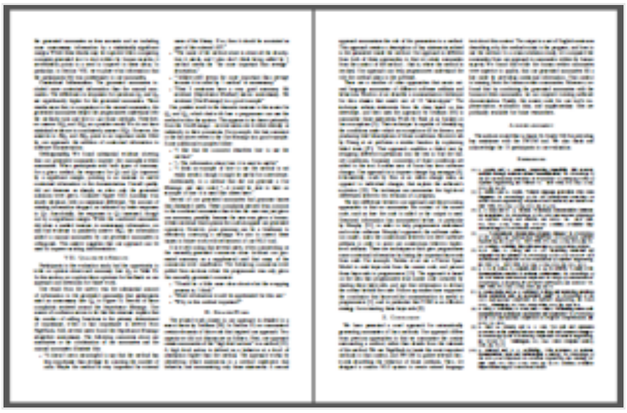


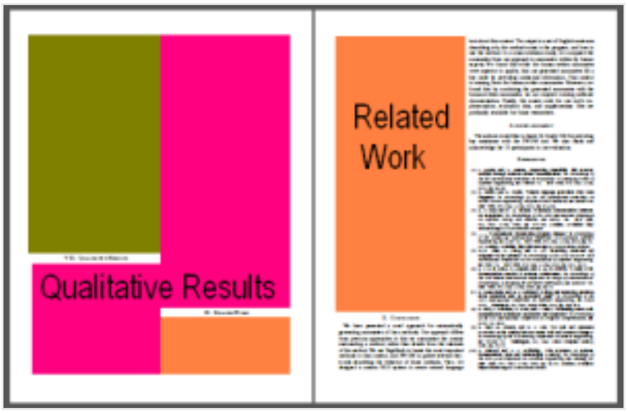
Now we come to pages 5 and 6. Page 5 is a continuation of the approach section. This is also a great place for a concrete example. Pick out one example and explain it thoroughly. Show exactly what the input and output of your approach. This example will lead nicely into the evaluation on page 6. Now, here you **do not** include **anything** about your research results. You **only** talk about your evaluation methodology. I repeat: you only talk about your evaluation methodology, NOT THE RESULTS YET. By methodology I mean, what gold sets did you obtain? Did you have human evaluators? If so, how many and who were they? What metrics did you use? And above all: what were your Research Questions?





Page 7 continues the evaluation section. Remember to end the evaluation section with a subsection called Threats to Validity, to let everyone know what you perceive as the weaknesses of your study, and what you tried to do about those weaknesses. On page 8, *at last*, we get to the Empirical Results. These are the results of your evaluation that can be quantified, such as metrics you measured or time taken. This is also where statistical test results go, and is usually where you answer most of your research questions. This is **not** where you describe the problem or the solution, or your evaluation methodology. It is only where you describe your quantifiable results.





Page 9 might be some spillover results, and also any qualitative results you have. Qualitative results being subjective results such as comments from human evaluators or unbiased observer opinions. Page 9 and 10 will usually also have your Related Work section, though some papers include this up front. And of course page 10 closes with a short conclusion, acknowledgements, and your references.

Sometimes conferences allow the references to fall onto an 11th or 12th page, provided you don't put any content on those pages.

Observations

Whew! That's a lot of writing. But it is also doable if you break it down into these digestible nuggets. Work a little bit each day, and try to write one section per week. That will get you a first draft within two months. If you are smart, you will be writing **while doing the research**, so you will have results just in time to write the results section.

One more thing to notice here: we only spent about 2 pages on the approach. We spent two pages on the problem, one and a half on the new idea, and then four pages on the evaluation and results. Got it? *That's just 20% of the paper on your new idea*. The other 80% is spent talking about the problem, supporting technologies, related work, and evaluation.

Anatomy of a Scientific Research Paper (Part 2): The Dreaded Introduction

The best place to start writing is the introduction. Many people dread writing the introduction though — it is the first place that reviewers will decide to reject your paper, so you have to get it right. It takes three things. One is a strong understanding of your research project: what problem are you solving and why. Your advisor should help your with this, and I will write another post on how to best understand your own research. Another "thing" is a strong understanding of your audience, so you know what you need to explain and what you can skip. This understanding comes in time after reading many papers in your field.

The last "thing" it takes to write a good introduction is to organize your thoughts in the way a reader is expecting. This organization is rather mysterious in the sciences, so I am going to explain it now.

Your reader is expecting the following:

1. A description of the high-level domain / environment
2. A key problem that happens in this high-level domain
3. Current solutions to that key problem
4. A problem with the current solutions
5. A glimpse at how you solve the problem in (4)
6. An overview of your evaluation / experiment results

That's it! If you write a few sentences on each of these, you will have your 1-page introduction in an acceptable format. Now, the "problem" may be a real problem practitioners have, an unsolved technical problem, or even a "knowledge problem" (such as an issue that has not been studied and explained in the literature yet). But the principle is almost always the same. Let's look at each item.

High-level domain / environment

This is where you connect your research to a real-world problem. Do not get flowery here, this is not Shakespeare and it is not a speech. Start with a definition of your problem. If you are describing a new technique for growing drought-resistant corn, a good starting sentence might be "Drought-resistant corn is..." If you are describing a study of lightning bolts and tree mortality, a good starting sentence might be "Tree mortality from lightning strikes is..." Etc. Answer questions which are obvious to you, but non-obvious to others, such as whether tree mortality has to take place immediately after the strike, or within 2 days, or what.

Take a look at this paper I co-authored this year:

Panichella, A., ***McMillan, C.****, Moritz, E., Palmieri, D., Oliveto, R., Poshyvanyk, D., and De Lucia, A., "Using Structural Information and User Feedback to Improve IR-based Traceability Recovery", in*

Proceedings of 17th European Conference on Software Maintenance and Reengineering (CSMR'13), Genova, Italy, March 5-8, 2013, pp. 199-208. [PDF]

From paragraph 1:

> *Traceability recovery is a key software maintenance activity in which software engineers extract the relationships among software artifacts. These relationships (called "traceability links") are a valuable resource during software maintenance because they provide a connection from high-level software documents such as use cases to low-level implementation details, such as source code and test cases [3].*

"Traceability recovery is…" The reader immediately knows what this paper is about. Even if you don't know what traceability recovery is, you now know what you need to know in order to understand the paper.

Problem in high-level domain

So you've explained the high-level domain. Great. So what's the problem in that domain? Spend some space explaining it *in simple language*. Perhaps the problem in creating drought-resistant corn is that water evaporates quickly from the leaves. So you would say "Unfortunately, drought-resistant corn is difficult to breed because water evaporates quickly from the leaves." Here's the example from my paper:

> *Unfortunately, traceability links are notoriously difficult to extract from software [3, 13, 28]. Software engineers must read and understand different artifacts to determine whether a link exists between two artifacts. Meanwhile, the artifacts are constantly being modified in the midst of an evolving software system. Maintaining a list of up-to-date traceability links inevitably becomes an overwhelming, error-prone task. Automated tools for traceability recovery offer an opportunity to reduce this manual effort and increase productivity.*

Traceability links are hard to extract. Don't believe us? Check out these related papers. It is extremely expensive to do by hand.

Current solutions

The problem is well-known, so there must be current techniques to address it. You may think of these current solutions as your competition, or against what you might evaluate your new technique. Even if the only solution is manual. In the case of a literature "knowledge problem" paper, the current solutions may be papers that partially answer the question, or address similar questions. In the corn example, maybe the current strategy is to breed plants with thicker leaves that dry out less quickly. Or, in our paper about traceability links:

Information Retrieval (IR) [5] has gained wide-spread acceptance as a method for automating traceability recovery [3, 13, 21, 28]. The IR-based methods, such as those based on Vector Space Model (VSM) [5] or probabilistic Jensen and Shannon (JS) model [1], identify traceability links using the textual information from the software artifacts. For example, the keywords from documents describing use cases may match keywords in the comments of a source code file. Textual information has the advantage of being widely available…

Problems with current solutions

The current solutions ain't all ice cream and lollipops. They have their own problems. Sometimes severe problems. Talk about them. Maybe breeding corn plants with thick leaves causes the corn ear to be smaller, because energy is used by making leaves instead of grain. Say so. From our traceability paper:

…but it is unfortunately also highly subjective. Words may have multiple meanings, identifiers from software are often misleading if taken out of the context, and comments are frequently out of date [2]. Different strategies have been successful in improving IR-based methods, including text pre-processing (e.g., [37, 39]), smoothing filters [12], and combinations of these approaches [17]. Nevertheless, imprecision remains a major barrier to using IR for traceability link recovery in practice.

Notice the pattern of problem, solution, problem-with-solution, solution-to-problem-with-solution, etc. This can continue a couple times until you get to your point:

Structural information contained in source code (e.g., function calls or inheritance relationships) has been proposed in solutions to increase the precision of IR-based traceability recovery [31]. In general, a combined approach will use an IR-based method to locate a set of candidate links, and then either augment or filter the set of links based on the structural information. However, combined approaches tend to be sensitive to the IR method. If the candidate links are correct, then the structural information can help locate additional correct links. Otherwise, the structural information offers little help, or will even pollute the results with incorrect links.

Your solution to the problem-in-the-current-solution

Finally you describe what you did. Take a paragraph to tell your reader the *key idea* behind your approach. Maybe you bred corn plants with deeper roots instead of thicker leaves. Use simple language to tell the world. Here's our key idea:

Our conjecture is that the traceability links recovered by IR-methods should be verified by software engineers prior to expanding the set of links with structural information.

Then we went on to give a brief example to illustrate our answer. An example is one way, citing supporting literature is another way.

Summary of experimental results

Now explain how you evaluated what you did and what results you saw. Remember you only have a few sentences here. Maybe you planted both your variety of corn and a competitor's corn in quarter-acre test plots in 5 different counties. Say so briefly. Then say what your results were. Such as if you found corn ears 5% larger, or whatever.

Follow my simple formula here and you will find your introductions much easier to write, and much easier for reviewers to accept! Of course there are many factors to a good paper, but my formula will get you started.

Write the Introduction FIRST

In writing the Anatomy posts, I realized an important ingredient to successful research: writing the introduction first. **First.** First, as in **before** you "do the research." Before you design the experiment. Before you write the related work. Before you tap out a single line of code. Write the introduction first.

Most students recoil at the idea. I did. After all, why write anything before you have results to write about?

Because you do not understand your research until you have written the introduction. Recall the six points that you need to tell your reviewers in your introduction:

1. A description of the high-level domain / environment
2. A key problem that happens in this high-level domain
3. Current solutions to that key problem
4. A problem with the current solutions
5. A glimpse at how you solve the problem in (4)
6. An overview of your evaluation / experiment results

If you cannot fill in these six points **right now**, it means you do not understand your research. You may think you understand it, but you don't. Here's a typical situation. A graduate student starts in June on a summer project. After two months it's August 1st and the student has downloaded 3 gigantic datasets, written clever scripts to organize them as a SQL database, and, with great effort, written a C program to compute a new metric on the datasets via the SQL tables. In a couple days the student loads all the metrics into an Excel spreadsheet. After struggling with XLStat for a few hours the student manages to calculate a statistical hypothesis test comparing the new metric to some old metrics. And tada! The new metric provides statistically-significant improvement to all previous metrics! Hurray! A Publishable Result!

Then comes the start of the academic year. Someone asks the student, "what did you do this summer?" And the student responds:

> *Well, I worked with Dr. Soandso and programmed a tool that extracts data from SQL database and processes it using a math library along with some supporting functions I had to write.*

You can hear it now:

> *Oh, so what problem did you solve?*
>
> *Hmm, the definitely the hardest problem I solved was that the SQL library I was using at*

> *first only could access MySQL databases, but the database had to be in PostgreSQL, so we had to … blah blah blah*

I did this *all the time*. I didn't understand my own research! How embarrassing!

Zoom out and ask yourself: what if you were talking to a reporter? What would you tell him or her that you work on? How would your research affect the public, even in a very small way? Maybe the metric you calculated was a way of measuring some qualities of corn stocks (protein and fat content, perhaps). And your new metric is more-correlated to price than all other metrics (the statistically-significant result over 3 datasets). That means you created a tool that helps farmers grow better crops!

Now the reporter is interested. The public hears of your discovery, and is excited enough to pay you its hard-earned money via taxes to continue your research. The public, the farmer, and you all benefit. You can now be a happy part of the virtuous cycle of scientific research described in a book by Pietra Rivoli.

Well, you *could* be a happy part of the virtuous cycle, if you understood your own research. Understood **the point** of your own research, that is. Quick exercise, answer in three sentences max: what did the student I mentioned above do this summer for research? How about:

> *The student created a tool to help farmers grow better corn. The tool is a better way for farmers to determine the quality of the corn growing in their fields. It works by measuring the protein and fat content in a sample of the corn, and comparing these to historical data.*

Makes total sense!

If you write a good introduction first, you will be forced to answer these questions up front. You will be forced to understand how your research fits into the big picture. You will know who benefits and why. You will feel much more confident about your work, and will be able to work more quickly.

*Now for practice. :-) What did you do for **your** research?*

Anatomy of a Scientific Research Paper (Part 3): What's YOUR Problem?

It's the first question every reviewer will ask when reading your paper: *what problem are you solving?*

If you do not answer it, your paper will be rejected. If you answer it in a way that is too complicated, your paper will be rejected. And if you **can not** answer it, your advisor will probably not let you submit the paper in the first place. In fact, I would say this is probably the number one reason that papers are rejected. Papers usually start talking about the solution before talking about the problem.

The good news is that you can make it easy for reviewers to find out what problem you address. You do this with a section titled *The Problem*. This is the section we'll look at today.

The Problem will typically have three parts. The first part is a simple description of the problem. The second part explains the main reason why the problem is difficult to solve. The third part is a motivating example that demonstrates the problem. Let's look at these. Take a look at the first paragraph of this *The Problem* section:

> *The long-term problem we target in this paper is that much software documentation is incomplete [28], which costs programmers time and effort when trying to understand the software [11]. In Java programs, a typical form of this documentation is a list of inputs, outputs, and text summaries for every method in the software (e.g., JavaDocs). Only if these summaries are incomplete, do the programmers resort to reading the software's source code [37]. What they must look for are clues in the source code's "structure" about how the methods interact [16, 22, 48]. The term "structure" refers to both the control flow relationships and the data dependencies in source code. The structure is important because it defines the behavior of the program: methods invoke other methods, and the chain of these invocations defines how the program acts. In this paper, we aim to generate documentation that is more-complete than previous approaches, in that our generated documentation contains structural information in each method's summary.*

Now, you have probably never read this paper before. I'm dropping you at the beginning of Section 2, so you haven't even read the introduction or abstract. But you still know what the problem is. This is possible because the paragraph uses simple language such as "The long-term problem we target in this paper is…". You also know why the problem should be solved: it costs programmers time and effort. And notice that even though these statements may seem like common sense, they are cited — leave no question that your problem is real.

Moving on. Sentence two, the first of several definitions here. Many readers may have different ideas about common terms like "documentation." So we set up here that we're talking about JavaDoc-like documentation. I.e., documentation for programmers, not end users.

Sentence three and four describe what programmers get out of this documentation. They are looking for some kind of "structural" information. We know what this is, but readers may not, so we define it in sentence five, and say why this "structure" is important in sentence six. Let me point out again to use simple language. "The structure is important because...".

Sentence seven hints at what we do. **Notice**, there is **no** discussion of our solution. *The Problem* section should not talk about the solution. The most it should do is clarify what problem is being solved in the paper. Look at the next paragraph:

> *We include this structural information from the "context" surrounding each method in the program. A method's "context" is the environment in which the method is invoked [24]. It includes the statement which called the method, the statements which supplied the method's inputs, and the statements which use the method's output. Context-sensitive program slicing has emerged as one effective technique for extracting context [24]. Given a method, these techniques will return all statements in its context. However, some statements in the context are more-relevant to the method than other statements. This issue of relevance is important for this paper because we must limit the size of the text summaries, and therefore select only a small number of statements for use in generating the summaries.*

Sentence eight and nine give more definitions. All these definitions are important for understanding our solution. Do not obsess about citations here, but do cite the definitions whenever possible, and definitely when you use a non-standard definition.

In addition to these definitions, the point of this paragraph is to let your readers know what is so hard about your problem. You do not want your readers to react by thinking "this problem been solved already" or "the trivial solution is the best one for this problem". These reactions are unfortunately common for scientific readers, and you should be sure to make the difficulty clear. Or, if you prefer, to make your "niche" clear.

Next is a good time to reinforce the difficulty. A motivating example usually does the trick:

> *Consider the manually-written examples of method summaries from NanoXML, a Java program for parsing XML, below. Item 1 is an example method we selected. It demonstrates how the default summary from documentation can be incomplete. In isolation, the method summary leaves a programmer to guess: What is the purpose of reading the character? For what is the character used? Why does the method even exist? ...*

I've snipped the rest of the example for the sake of space, but you can look it up in the paper. Just remember: the point is to communicate *what problem you are solving*. Show the inputs and typical outputs. Contrast these to the outputs from your solution.

Now, another item you may want to discuss in *The Problem* is the benefit to a solution to the problem.

You may wish to do this if your problem is somewhat obscure. In the case of the example I used above, we take it for granted that better documentation generators would reduce programmer time and effort. But that may not be the best strategy depending on your problem or venue.

There is another time you will want to explicitly describe the benefits of a solution. That is when the problem you solve is a “knowledge problem.” It is a topic for another post, but basically a knowledge problem is a hole in current literature. For example, your paper may discuss an empirical study where you compared different tools/chemicals/specimens/etc. In this case, you will probably not have a motivating example, and instead explain exactly how other papers do not quite cover the problem you discuss. I’ll write a different post about this, but for now some information is in Write the Introduction FIRST.

Anatomy of a Scientific Research Paper (Part 4): Background

We are now starting to enter the heart of the paper. Like any project, your work is going to be partially built on existing components. In fact, it may be *entirely* built on existing components. Your contribution may be a new combination of the existing components that has not been tried before.

Now, some of these components will be necessary to understand your work. These components are what you should describe in your background section. That's what we're going to discuss today.

But what can you expect your audience to know? Which components need to be explained, and which can we assume that people know? The trick is to identify the components that you would need to tell someone else in your field, if that someone were to reimplement your approach. Try this exercise: pick a colleague, not in your lab, who you know personally. Then call him or her on the phone and explain how to implement your idea. Note whenever your colleague says "now, what is the XYZ again?" Those are the components you need to explain. You can even do this exercise in your head if you know your colleague well enough. You can anticipate the questions because you know what you know that he or she does not know. (Whew!)

Say you are developing a new treatment for an old skin disease. You propose that tiny doses of two different heart drugs will cure the disease. Your colleagues in dermatology might be aware of the heart drugs, but not how those drugs work. This is a good clue that you should explain it briefly — you want your audience to see your intuition about why your idea will work.

But watch out for the classic **trap** here. The trap is that you might start explaining your approach in the background. The temptation to do this is very strong: you have been working on the project for months or years. Your brain is in the details.

So, do NOT discuss ANYTHING about the approach yet. DO NOT DISCUSS THE APPROACH YET.

Instead, write down the 2-4 "must know" technologies. Then identify a few references that describe those technologies. Then start writing the background.

As usual, this is time for fancy talk. Use simple language. How about:

> *Section 3: Background This section describes two supporting technologies for our work: ACME Portable Rockets (APR) [23] and Paintable Trick Walls (PTW) [53]. These techniques were proposed elsewhere. We explain them here because we use them in our approach.*

And now, one sub-section for each of those technologies. Remember, this ain't Shakespeare:

> *Section 3.1: ACME Portable Rockets*
> *ACME Portable Rockets are…*
> *A typical application of APR is…*

Section 3.2: Paintable Trick Walls
Paintable Trick Walls are…
A typical application of PTW is…

Expect each subsection to be 2-4 paragraphs. In each subsection, answer these questions:

- What are the inputs to the tool / technique / technology?
- What are the outputs of the technology?
- What is the primary mechanism of action of the technology?
- Who invented the technology (reference the original paper)?
- What are the typical application for the technology?
- What is the key advantage of the technology over competitors? (e.g., is it safer? faster? cheaper? what?)

Notice that the questions have NOTHING to do with *your* approach. You do NOT say anything about why the technologies are the best ones for your application, why you chose them, etc.

There are two reasons you want to avoid discussing your approach in the background section. First, it is what readers expect. When the section header says "Background", your readers expect to hear about the background. Second, it makes the background much easier to write. You can write the background even if you have not finished the approach. In fact, you **should** write the background before the approach. It will help you better understand the tools you are using in your own work.

So, what are *your* supporting technologies?

Anatomy of a Scientific Research Paper (Part 5): The Approach

Today we'll talk about the approach. In a typical "build a better mousetrap" paper, this is the only section which contains details about your new idea. Yes, the only section. Everything else is setup or evaluation or related work.

And the approach section is at most 2 pages long. So how do we make effective use of it? How can we share our great new idea with the world?

Well, you can usually divide the approach into 4 parts:

1. A high-level overview / summary breakdown
2. Data collection / key implementation details
3. Detailed architecture and walkthrough
4. Example of the input and output

For the first part, you're looking for a 1 paragraph general summary. Break down how your system works from a very high level. Now, this can be difficult. You have had your nose in the details for months to years with your project, and you're going to want to start in on every tiny detail and nuance. Stop that now. Think of telling someone about how to jump start a car. You 1) open the hood of both cars, 2) connect the positive terminal of the bad battery to the positive terminal of the good battery, 3) connect cable to negative terminal of good battery, 4) connect cable to chassis of the dead car, and 5) start the dead car!

You could have said that "the crown detachment apparatus is operated to expose the electrical storage box before the X-marked-joint is established to the Electrical Carriage Line (ECL) while also established to an adjacent electrical storage box until an I-shaped-marked-joint established with the ECL is attached, or otherwise connected, to the I-shaped-marked-joint of the adjacent ESB via the metallic frame objects. However, pressure is not applied to the controller apparatus until this point. Furthermore, detachments may be necessary regarding controller output depending on feedback."

But that wouldn't make any sense at all.

Take a look at a paper Ameer Armaly, Casey Ferris, and I wrote together:

> *Armaly, A., Ferris, C., McMillan, C., "Reusable Execution Replay: Execution Record and Replay for Source Code Reuse", in Proc. of 36th IEEE/ACM International Conference on Software Engineering, New Ideas and Emerging Results Track (ICSE'14 NIER), Hyderabad, India, May 31-June 7 2014.*

Here's the first paragraph-and-a-half from the approach section of that paper:

> *Our approach enables the reuse of functions from C and C++ programs. Given a function to*

> *reuse,* ***our approach works in four steps:*** *1) from a log file, restore the state of the program containing the function at a point just prior to the function's execution, 2) modify any parameters or global variables as instructed by the programmer, 3) pass control to the function so that it executes, and 4) catch the function return so that the programmer can read the function's output.*
>
> *In this section, we will elaborate on each of these steps. We will use the example in Figures 2 and3 to illustrate how these steps work in practice. These figures show how our approach can reuse the function nearestStar() from Section 2.1.*

Now, you may have no idea about the details here, but if you understand programming, you can understand the gist of our idea. That's because we started off with a high-level description of the most important steps.

Next you will need some preliminary details, to prepare for explaining the big details. These preliminary details are just pointing to some background sections you wrote earlier, how to set up your new gadgetry, and other important technical details. This is also where you would mention how you actually get the input data (e.g., through some pre-processing, or from a database), what kind of programming languages you're dealing with, etc. Take a look at the next three paragraphs from our paper:

> *3.1 Supporting Technology*
>
> *We have heavily modified the Jockey library [19] for our approach. The most important modification we made was to add the ability to "go live." Many approaches, such as the one implemented in the GNU debugger, do not actually re-execute the logged instructions. Instead, they log the output of each instruction and, during replay, restore the state as it was after the instruction. This restoration produces an identical result when the logs are reviewed for debugging. For our work in reuse, we alter the state before replay, which means that the instructions will need to be re-executed, rather than restored. We implement a "go live" system after the state is restored, inspired by an approach described by Laadan et al. [15].*
>
> *3.2 Preparation*
>
> *To prepare to reuse a function, a programmer must first record a checkpoint for that function. The checkpoint must be taken at a point just prior to the function's execution. We provide a recording utility based on Jockey's checkpointing feature. The utility takes a program and the name of a function in that program. The utility then executes the program. The programmer may interact with the program to ensure that some behavior is recorded, or run a test script. The utility monitors the process — whenever the function is*

called, the utility directs Jockey to record the state of the program to a checkpoint file. The function may be called several times, and there will be one checkpoint for each of these. The programmer can choose a checkpoint that he or she prefers, otherwise the default is the first checkpoint.

3.3 Reusing Functions

We implemented our approach as a userspace C/C++ library for 32-bit Linux 2.6.10. While implementation for different environments is possible, in this paper we limit the scope to one environment for clarity and reproducibility. Figure 2 shows an example program using our library. The remainder of this section will cover the steps of our approach, using this example for context.

Supporting Technology, Preparation, then gritty details about the kernel version our tool runs on and so forth. Everything important for reproducing our approach. Nothing extraneous.

At long last comes part 3 of the approach, which finally shows what you're system does internally. Check out page 3 from our paper. Here it is for convenience:

[illegible]

Essentially what you want to do here is show a figure or two with the either the architecture of your system, or the user interface to the system. Mark the figures with bubbles labeled 1, 2, 3, …, for each thing that happens. Then clarify in the text. Walk through what happens. At the top-right of the page above: "To make this feature available for reuse, we have provided flashback_load_scene() in our library (Figure 2, area 1)." Use as many gritty details are you need to explain what is happening. For example, we point out that we use libdwarf. That breakpoints are placed in pre-defined memory locations. Etc.

In part 4 of the approach, the example, you want to make sure that somewhere there is an explicit example of the input to the system, the output of the system, and a walkthrough of how input became output. In our short paper above, we rolled the example into the background as a motivating example, and into the approach as Figures 2 and 3. In other papers, you may want to create a separate section with an example if you didn't get a chance to put it elsewhere. Do not let your paper end without a clear example of the input and output to your system. Do NOT let your paper end without a CLEAR example of the INPUT and OUTPUT.

And that's how it's done! To recap:

1. A high-level overview / summary breakdown
2. Data collection / key implementation details
3. Detailed architecture and walkthrough
4. Example of the input and output