

# Practical T<sub>E</sub>X 2004 — program and information

**Sunday** 3–5 pm *registration*  
**July 18** 5–7 pm *reception*

---

■ **Track 2: Introduction to L<sup>A</sup>T<sub>E</sub>X class.** Starting at 10:30 am Monday, and 9 am Tuesday and Wednesday, and ending at lunchtime each day, Sue DeMeritt and Cheryl Ponchin will teach a continuing class on beginning and intermediate L<sup>A</sup>T<sub>E</sub>X, with no prerequisites. Participants can choose whether to attend this class or the morning talks.

■ **Mac OS X & T<sub>E</sub>X session.** Starting at lunch time Monday and continuing into the afternoon, a round-table discussion on Mac OS X will be held in the Napa room, led by Hans Hagen, Wendy McKay, and Ernest Prabhakar from Apple.

---

**Monday** 9 am Karl Berry, T<sub>E</sub>X Users Group *Welcome*  
**July 19** 9:15 am Peter Flynn, Silmaril Consultants *Keynote address: T<sub>E</sub>X and the interface*  
 10:15 am *break*  
 10:30 am Eitan Gurari, Ohio State University *TeX4ht: HTML production*  
 11:15 am Kaveh Bazargan, River Valley Technologies *L<sup>A</sup>T<sub>E</sub>X to MathML and back: A case study of Elsevier journals*  
 11:45 pm Hans Hagen, NTG, Pragma ADE *The pros and cons of PDF*  
 12:30 pm *lunch*  
 2:00 pm Jenny Levine, Duke University Press *Label replacement in graphics*  
 2:30 pm David Allen, University of Kentucky *Screen presentations, manuscripts, and posters from the same L<sup>A</sup>T<sub>E</sub>X source*  
 3:15 pm *break*  
 3:30 pm Baden Hughes, University of Melbourne *T<sub>E</sub>X and XML*  
 4 pm Hàn Thế Thành, University of Education, Ho Chi Minh City *Micro-typographic extensions of pdfT<sub>E</sub>X in practice*  
 4:45 pm q & a *moderator: Lance Carnes*

---

**Tuesday** 9 am Volker R.W. Schaa, Dante e.V. *pdfT<sub>E</sub>X and XML workflow for conference proceedings*  
**July 20** 9:45 am Anita Schwartz, University of Delaware *Paperless dissertations at the University of Delaware*  
 10:30 am *break*  
 10:45 am Hans Hagen *MetaPost: More than math and fonts*  
 11:45 am Brooks Moses, Stanford University *MetaPlot, MetaContour, and other collaborations with MetaPost*  
 12:30 pm *lunch*  
 1:30 pm Cheryl Ponchin, Ctr. for Comm. Research *L<sup>A</sup>T<sub>E</sub>X survey*  
 2:15 pm Steve Grathwohl, Duke University Press *What is ConT<sub>E</sub>Xt, that we should be mindful of it?*  
 3 pm *break*  
 3:15 pm William Richter, Texas Life Insurance Co. *T<sub>E</sub>X and scripting languages*  
 4 pm q & a *moderator: Karl Berry*  
**social events** (see next page)  
 5 pm *treasure hunt*  
 7:30 pm *banquet*

---

**Wednesday** 9 am Nelson Beebe, University of Utah *A bibliographer's toolbox*  
**July 21** 9:45 am David Jones, American Mathematical Soc. *The amsrefs package*  
 10:30 am *break*  
 10:45 am Steve Peter, Beech Stave Press *T<sub>E</sub>X and linguistics*  
 11:30 am Hans Hagen *ConT<sub>E</sub>Xt*  
 12:30 pm *lunch*  
 1:30 pm Steve Grathwohl *70 years of the Duke Mathematical Journal online*  
 2:15 pm q & a *moderator: Baden Hughes*  
 3 pm *break*  
 3:15 pm panel: Digital Publishing *moderator: Lance Carnes; Kaveh Bazargan, Karl Berry, Peter Flynn, David Fuchs, Hans Hagen.*  
 4 pm *end*

---

**Thursday** **additional courses**  
**July 22** Peter Flynn *Practical T<sub>E</sub>X on the Web*  
 Sue DeMeritt, Cheryl Ponchin *Intermediate and Advanced L<sup>A</sup>T<sub>E</sub>X*  
 Hans Hagen *Introduction to ConT<sub>E</sub>Xt*

## Treasure hunt

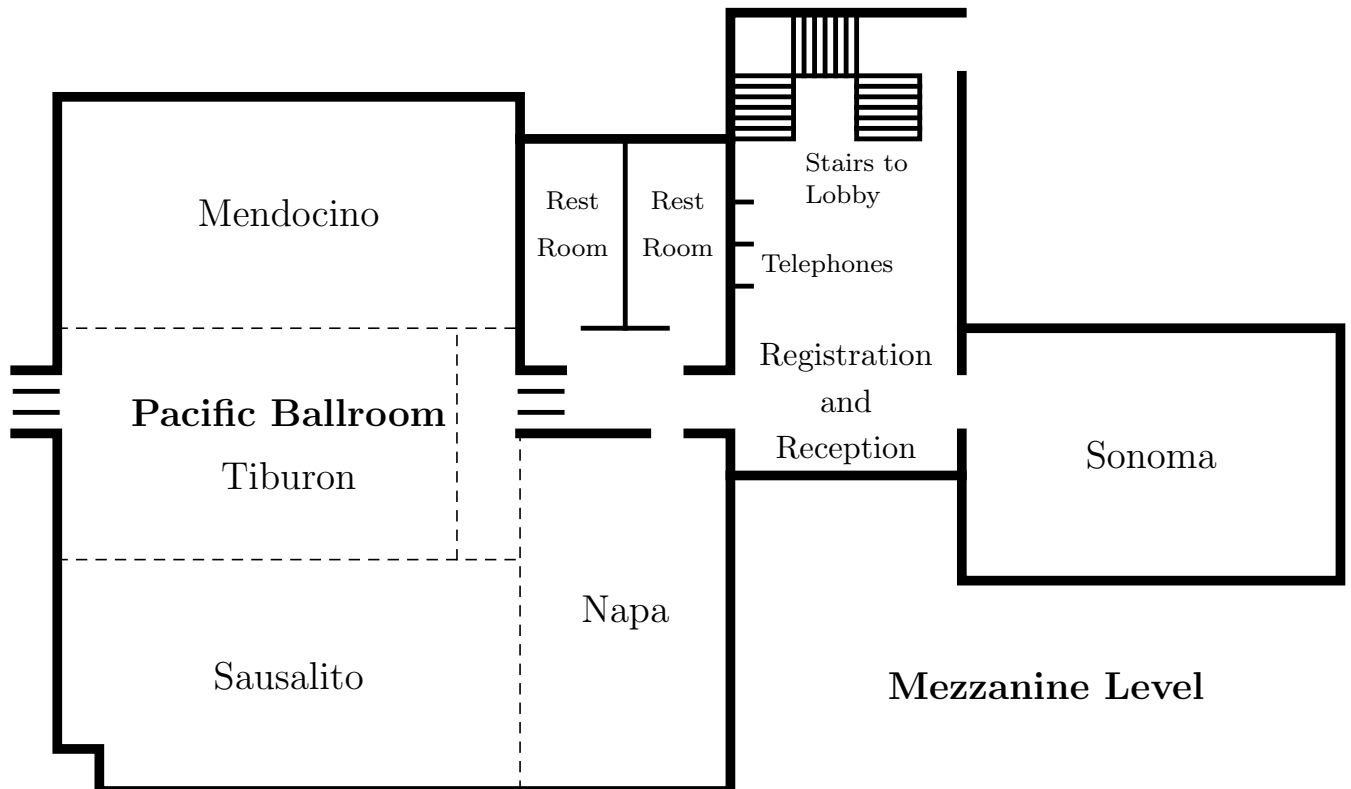
After the conference sessions on Tuesday, Lance Carnes will host a treasure hunt in San Francisco North Beach/Chinatown. Armed with clues, a street map, and keen problem solving ability, treasure hunt teams will look for eight solutions, starting at the conference hotel. The clues will lead hunters to landmarks, pubs, historic and notorious places. The solutions are a place name, a detail from a historic plaque, or some other minutiae. The winning teams will be the ones who arrive at the banquet restaurant at the end of the hunt with the most correct answers in the least amount of time. More information on a separate page near the end of this booklet.

## Banquet

At 7:30 pm Tuesday, the conference will hold a banquet at the Empress of China restaurant. This will be a multi-course Chinese dinner, including Sizzling Rice Soup, Peking Duck, Manchurian Beef, and Walnut Prawns. The banquet room has a commanding view of downtown San Francisco, Alcatraz and the bay, and Telegraph Hill. Several celebrity T<sub>E</sub>X guests will be attending the banquet!

## Conference logistics

- Registration is in the ‘reception and pre-function area’ at the top of the stairs, Sunday 3–5 pm. Please come and pick up your name tag, conference information, and other goodies at this time if possible. Otherwise, see Robin Laakso to register.
- The reception is in the same area, Sunday evening 5 pm–7 pm.
- Lunches will be served in the Bristol Bar Grill, downstairs off the lobby. This hotel restaurant is open to the public for breakfast and dinner but reserved for our group for lunch.
- Breaks will be served in the reception area.
- The main conference sessions are in the combined Mendocino/Tiburon space.
- The Track 2 introductory L<sup>A</sup>T<sub>E</sub>X class is in Sausalito, starting Monday 10:30 am and Tuesday, Wednesday 9 am, and ending at lunch time.
- The Mac OS X session is in Napa, starting Monday lunchtime.
- Internet access is available in Sonoma, ≈ 7:30 am–9 pm. See Robin Laakso for emergency access outside regular hours. General wireless access is available throughout the conference area and lobby.
- Thursday classes will be held in Tiburon, Sausalito, and Mendocino.



# Prac $\TeX$ Treasure Hunt

Tuesday, July 20,  $\approx$  5–7:30 pm

## How does it work?

First, form a team of 3–6 people. You can assemble a team prior to the conference, or join a team the evening of the hunt. Some clues may require knowledge of American idioms, while other clues can be solved with keen observation and logic. All participants, whether from the US or abroad, can play a role in solving clues.

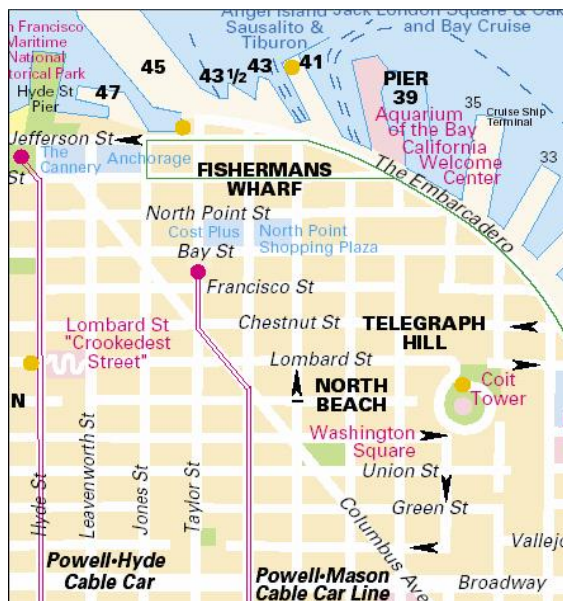
The hunt begins at the Holiday Inn. You will be given a set of clues, a map, and an answer sheet. You may solve the clues in any order; it's best to try to figure out the locations before setting out on foot, to avoid backtracking.

The clues will lead you to a location, and from there you look for the solution. In most cases, you will have to get to a location before the rest of the clue makes sense.

You have a maximum of two hours for the hunt. You may not be able to solve all the clues in that time, so if you cannot get the solution within a few minutes, move on to the next clue. Submit answers at the banquet restaurant (Empress of China) before the designated ending time ( $\approx$  7:30).

## Sample clues

The first part of a clue guides you to the location. The other part of the clue can only be solved once you are at the location. The location part of these sample clues can be solved by referring to the map, along with some shrewd reasoning. Since the answer to the clue can be solved only when you are at the location, for now see if you can find the locations for these clues. Answers are below. (The Holiday Inn is at the corner of Columbus Ave. and North Point St.)



**Sample Clue 1.** Roman seamen will think of this landing place as XXXIX. To the west hundreds of sea creatures have made their home where sailboats once berthed. What are these creatures that have a name like the  $\TeX$  mascot, but whose cry is like a dog's?

**Sample Clue 2.** At the corner where a 15th century sailor crosses a small body of water, this place's name includes circular items which have become smaller in size over the past 20 years, yet their price has certainly not shrunk.

## Rules

The hunt is best when all team members contribute. Teams must collaborate in solving the clues and must stay together at all times. Individuals may not leave the team and solve clues on their own.

You must travel on foot. All team members must be willing to travel at the same pace, so consider this when joining a team.

All team members must be present when the answers are submitted at the end of the hunt.

The solutions to the clues will be distributed after the hunt. The top three teams will receive prizes (not to mention bragging rights).

## Acknowledgments

The Prac $\TeX$  Treasure Hunt is based on the popular Chinese New Year Treasure Hunt created by Jayson Wechter.

## Answers

- Answer to Sample Clue 1: The place is **Pier 39** (we're assuming Roman seamen think in roman numerals). Next to this pier are floating berths that have become a permanent home to **sea lions**.
- Answer to Sample Clue 2: The corner is **Columbus** (15th century sailor) and **Bay** (small body of water), and the place is **Tower Records** (which once sold vinyl records but now sells CD's).

## **L<sup>A</sup>T<sub>E</sub>X to MathML and back: A case study of Elsevier journals**

Kaveh Bazargan

Our main business is typesetting and content management for mathematical journals. One of our clients is Elsevier. As of this year, all of their journals will be archived in XML, with mathematics in MathML. In this presentation I will outline and give a live demonstration of how we tackled the automatic conversion between L<sup>A</sup>T<sub>E</sub>X and MathML. This has led to a workflow which we have standardized for all journals we handle.

## **A bibliographer's toolbox**

Nelson Beebe

This article surveys a portion of a set of software tools that I have developed over the last decade for the production, maintenance, testing, and validation of very large bibliographic archives. It provides resource locations for all them, and shows how they can make bibliography preparation and maintenance more productive, and much more reliable.

## **T<sub>E</sub>X and the interface**

Peter Flynn

T<sub>E</sub>X systems have been a cornerstone of research and academic publishing for a long time. Development of how it interfaces with different classes of user or potential user, however, has been uneven. Recent developments in other areas of text processing are opening up new opportunities for T<sub>E</sub>X-based systems. Should T<sub>E</sub>X development become involved in these areas, or should it be restricted to those areas where it has traditionally been a strong player?

## **What is ConT<sub>E</sub>Xt, that we should be mindful of it?**

Steve Grathwohl

ConT<sub>E</sub>Xt can not only be considered a L<sup>A</sup>T<sub>E</sub>X for the 21<sup>st</sup> century, but, more generally, an evolving platform for document engineering, if we take a very expansive view of what constitutes a 'document'. But in this presentation I want to cover how ConT<sub>E</sub>Xt handles some document features with which we are all familiar. In order to demonstrate ConT<sub>E</sub>Xt's powerful and intuitive key-value setups for document structures, I will then show how I implemented a simple book design, a type of project I always undertake when trying to learn a new system. The book is *A Voyage to Arcturus* by David Lindsay.

## **70 years of the Duke Mathematical Journal**

Steve Grathwohl

The *Duke Mathematical Journal* published its first issue in 1935. Since the 1980s, it has been one of the leading independent general mathematics journals. Over the last year, we at Duke have finished putting the entire corpus on the World Wide Web, in appropriately indexed and searchable form. I give in this presentation a short overview of DMJ Online, which is hosted by Project Euclid at Cornell University, and make a few remarks about the management of metadata for the project, which involves an intersection of the T<sub>E</sub>X and XML worlds mediated by Perl.

## **TeX4ht: HTML production**

Eitan Gurari

TeX4ht is a highly configurable system for producing hypertext from TeX-based sources. The system is distributed with a large set of configuration files. The most commonly used configurations are those supporting L<sup>A</sup>T<sub>E</sub>X inputs and HTML, MathML, OpenOffice, and DocBook targets. The first part of the presentation will describe how the system can be used for different applications.

ConT<sub>E</sub>Xt is a new addition to the style files being supported by TeX4ht. The second part of the presentation will describe the work done to provide TeX4ht configurations for ConT<sub>E</sub>Xt, with the objective of providing insight into the inner working of TeX4ht.

## **How to use micro-typographic extensions of pdfT<sub>E</sub>X in practice**

Hàn Thế Thành

Micro-typographic extensions of pdftex like margin kerning and font expansion have been around for a while. While the demos show interesting results, applying those extensions to daily use is not that simple for an average user. In this article I will share some experiences in using those extensions in practice, and give a few simple and useful recommendations for a quick start for newcomers.

## **T<sub>E</sub>X and XML**

Baden Hughes

With the growing prevalence of XML data, it is logical to consider ways in which XML and processing engines such as T<sub>E</sub>X can be integrated efficiently to produce high quality typographic output. In this session, we will first review the history of approaches to the integration of T<sub>E</sub>X with other structured data types; and motivate the work here by considering a range of typical use cases. Adopting a typological approach, we will consider: XML in T<sub>E</sub>X documents; XML as input to T<sub>E</sub>X processes; XML as output from T<sub>E</sub>X processes; and XML as an intermediary between other processes and T<sub>E</sub>X itself. We conclude with a review of the state of the art of T<sub>E</sub>X and XML integration, and a survey of current directions.

### **The `amsrefs` package**

David Jones

The `amsrefs` L<sup>A</sup>T<sub>E</sub>X package extends the benefits of B<sub>I</sub>B<sub>T</sub>E<sub>X</sub>'s structural markup to all stages of a document's life cycle while simultaneously ameliorating a large class of problems that frequently plague B<sub>I</sub>B<sub>T</sub>E<sub>X</sub> users. Using `amsrefs`—either in conjunction with B<sub>I</sub>B<sub>T</sub>E<sub>X</sub> or independently of it—an author can easily create a self-contained L<sup>A</sup>T<sub>E</sub>X file that retains all the semantic information that is typically lost when converting B<sub>I</sub>B<sub>T</sub>E<sub>X</sub> database records into L<sup>A</sup>T<sub>E</sub>X source code, making it easier, for example, to reuse L<sup>A</sup>T<sub>E</sub>X documents in other contexts, such as on the Web. At the same time, issues such as non-Latin characters in names and capitalization of titles are supported more naturally, eliminating the need for many of the special markup conventions of B<sub>I</sub>B<sub>T</sub>E<sub>X</sub> databases that trip up authors.

The `amsrefs` package is compatible with the `hyperref` and `showkeys` packages and provides support for sorted and compressed citation lists (à la the `citesort` and `cite` packages) and multiple bibliographies. It can also emulate the standard `plain`, `abbrv`, `alpha` and `unsrt` B<sub>I</sub>B<sub>T</sub>E<sub>X</sub> styles, and supports a rich set of author-year citation schemes.

This talk will demonstrate how to use `amsrefs` to solve common problems and compare and contrast `amsrefs` with existing solutions.

### **Label replacement in graphics**

Jenny Levine

In this presentation I show how graphics are manipulated to the *Duke Mathematical Journal* style. I give some examples and a step-by-step approach to assessing a figure file, removing its labels, and placing new ones using `graphicx` and `overpic`.

This is done to maintain a consistent style. Our labels should be in a compatible font and match the look of the journal as well as that of the article (size, placement, emphasis, etc.).

### **Paperless dissertations at the University of Delaware**

Anita Schwartz

The Office of Graduate studies at the University of Delaware decided to do their part in meeting our campus goal of a paperless environment by registering with UMI ProQuest as one of the first test universities to allow electronic submissions of dissertations. This means no more binding hard copies, only PDF (Portable Document Format) files. Eventually we plan to include theses, so we must make this first phase of the project a success. We choose to use the service at UMI because it fit well into our current campus requirements for UDThesis and it would require very little technical support. The plan is to go live in September 2004. This paper will briefly describe our current environment and the necessary steps to support UDThesis in our future environment. The primary goal of this paper will explain our support issues for the software applications (Word, T<sub>E</sub>X and L<sup>A</sup>T<sub>E</sub>X) used to generate theses and dissertations and options for generating a PDF file.

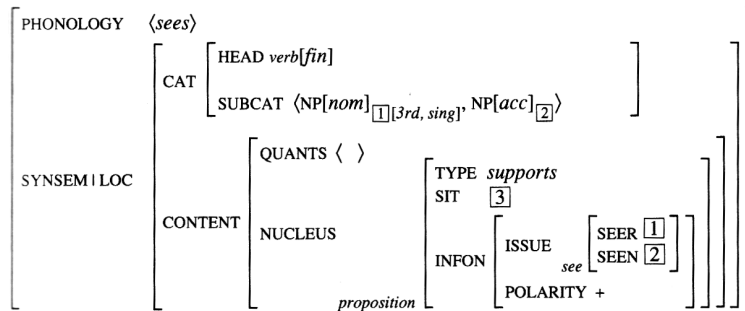
The overall goal of this project is to provide an enjoyable and successful experience for our graduate students and at the same time fulfill our campus goal of being a paperless environment.

# T<sub>E</sub>X and Linguistics

Steve Peter  
Beech Stave Press

Linguistics is a field that stands at the intersection of numerous other disciplines. Therefore it has a number of notational systems, some of which are quite difficult to work with in traditional word processors. However, since many of these notations are based on mathematics, T<sub>E</sub>X is a natural typesetting system for linguistics.

For example, the following lexical entry for the verb *sees* in Head-driven Phrase Structure Grammar is a nightmare to type and maintain in Word, yet it is quite straightforward in T<sub>E</sub>X. (I use a scan here here because in the talk I want to show how T<sub>E</sub>X may be used to typeset real linguistic examples, not to show what in linguistics is generally done via or influenced by T<sub>E</sub>X.)



The talk begins with an overview of the field of linguistics, showing the different notational systems each subfield uses. Following that, I show how (many of) these examples may be typeset in T<sub>E</sub>X, citing its advantages over other typesetting systems (and to be fair, its drawbacks as well), drawn from my own practical experience as a linguist and a publisher of linguistics.

# Screen Presentations, Manuscripts, and Posters from the Same L<sup>A</sup>T<sub>E</sub>X Source

David M. Allen  
University of Kentucky  
<http://www.ms.uky.edu/~allen/>

June 9, 2004

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Should we do it?</b>	<b>2</b>
<b>3</b>	<b>The Building Blocks</b>	<b>2</b>
<b>4</b>	<b>Sectioning Commands</b>	<b>3</b>
<b>5</b>	<b>Switches for Selective Input</b>	<b>3</b>
<b>6</b>	<b>Listing of Packages and a Sample Application</b>	<b>3</b>
<b>7</b>	<b>Sample output</b>	<b>6</b>

## 1 Introduction

This presentation describes three small packages, *screen.sty*, *manuscript.sty*, and *poster.sty*. The screen package is used to format the output for a screen presentation; the manuscript package for a manuscript; and the poster package for a poster. With a little care, the same input file can be processed with any of the three packages. These packages are used with the *article class*.

Each of these packages loads the *geometry package* and the *T<sub>E</sub>Xpower package* with options appropriate for the type of output. The package *poster.sty* also loads the *multicol package*. All features of *article class* are available unless there is a conflict with the *T<sub>E</sub>Xpower*, *geometry*, or *multicol* packages. For example, floats cannot be used with the *multicol* package.

Each of the packages defines two new commands, *newscreen* and *titledscreen*, but in different ways. The article sectioning commands are redefined in *screen.sty* and *poster.sty* to be more pleasing for those types of output.

This presentation gives the implementation details of the *screen*, *manuscript*, and *poster* packages. It will also demonstrate the use of the packages. One demonstration illustrates novel incremental building of PSTricks graphics.

## 2 Should we do it?

I anticipate the question: Does it make sense to have the same content in a screen presentation and a manuscript? Books on presentation style say there should be just a few talking points on each screen. However, there are some reasons to the contrary.

Different rules apply for scientific presentations. Often technical presentations are not immediately comprehended. If the audience is given a handout having some detail, then the handout can be studied later.

Composing a paper in nearly self-contained segments forces an author to use more structure. A theme and its discussion have to be kept together in a screen presentation. The screen presentation and the corresponding article for publication do diverge, but this happens toward the end of the composition process.

## 3 The Building Blocks

The packages *T<sub>E</sub>Xpower*, *geometry*, and *multicol* are the building blocks for the packages *screen*, *manuscript*, and *poster*. Here the building blocks and their options are briefly discussed.

The *T<sub>E</sub>Xpower* package is used for incremental displays and for color emphasis. Use the *display* option to activate the *pause* and *stepwise* incremental displays used with screen presentations. The *printout* option is used with posters and manuscripts to show only the final result of the incremental build. The option *lightbackground*, by default, gives black text on a pale yellow background. I use the *lightbackground* option for screen presentations and posters and the *whitebackground* option for manuscripts.

There are other options for color emphasis and color math. The user chooses the the options to suit his own taste. For example, he may use the *darkbackground* option which, by default, uses yellow text on a dark blue background.

The *geometry package* is used to set page dimensions, margins, and magnification. Since I live in the USA, I use the *letterpaper* option to specify the paper size for screen presentations and manuscripts. Outside the USA, one would likely use the *a4paper* option.

At the University of Kentucky, the printer used for posters takes rolls of paper that are 36 inches wide. One uses the *paperwidth* and *paperheight* commands for the desired poster size. It may be necessary to adjust these sizes to offset magnification. The *a0paper* option would likely be used for posters outside the USA.



The option `mag=\magstep4` (magnify by a factor of  $1.2^4$ ) for screen presentations and `mag=\magstep3` (magnify by a factor of  $1.2^3$ ) for posters seem to give pleasing results. Magnification is not used for manuscripts.

The margins I use may be seen in the code listings that follow. The author can set margins to his taste.

The `multicol` package is used to have multiple columns for posters. There are lengths for the amount of space between columns, the width of the rule separating columns, etc. My settings are shown in the listing of `poster.sty`.

## 4 Sectioning Commands

The screen and poster packages redefine the section, subsection, and subsubsection commands to make titles centered and colored. The color is `emcolor` defined in `texpower` for color emphasis.

When there is too much material for a single screen the material just spills over to the next screen. The place at which the material breaks across screen boundaries may not be pleasing, and you might want to break the material at a earlier point. The `newscreen command` is defined as `\newpage` in `screen.sty` and is defined to do nothing in the other two styles.

If the text at the beginning of a screen coincides with the beginning of a paragraph you may want to have a screen title that indicates the theme of the paragraph. The command `\titledscreen` has the screen title as its argument. In screen presentations, the title has the appearance of a sectioning command except that it is not numbered and will not appear in a table of contents. For the manuscript and poster styles, `\titledscreen` is defined as `\par`.

## 5 Switches for Selective Input

The `TEXpower` package loads the `iffthen` package. The packages `screen`, `manuscript`, and `poster` define Boolean strings: `screen`, `manuscript`, and `poster`. In each case, the Boolean with the same name of the file is set to true, and the other two are set to false. The `\iffthenelse` command can be used for selective inclusion of parts of the input file.

## 6 Listing of Packages and a Sample Application

This section contains listings of `screen.sty`, `manuscript.sty`, `poster.sty`, and an example `LATEX` file.

The content of `screen.sty` is

```
\NeedsTeXFormat{LaTeX2e}[1994/06/01]
\ProvidesPackage{screen}[2004/04/10 David Allen,%
```

```

                                version 0.0]
\RequirePackage[display,lightbackground,colorhighlight,
                coloremph,colormath]{texpower}
\newboolean{manuscript}
\newboolean{screen}
\newboolean{poster}
\setboolean{manuscript}{false}
\setboolean{poster}{false}
\setboolean{screen}{true}
\RequirePackage[landscape,letterpaper,nohead,mag=\magstep4,
                hmargin=0.5in,vmargin=0.25in,truedimen,footskip=1em]
                {geometry}
\renewcommand{\section}{\@startsection{section}{1}{0pt}
{-2ex plus 1ex minus 1ex}{2ex plus 1ex minus 1ex}
{\color{emcolor}\centering\large\bfseries}}
\renewcommand{\subsection}{\@startsection{subsection}
{2}{0pt}{-2ex plus 1ex minus 1ex}{2ex plus 1ex minus 1ex}
{\color{emcolor}\centering\large\bfseries}}
\renewcommand{\subsubsection}{\@startsection{subsubsection}
{3}{0pt}{-2ex plus 1ex minus 1ex}{2ex plus 1ex minus 1ex}
{\color{emcolor}\centering\normalsize\bfseries}}
\newcommand{\newscreen}{\newpage}
\newcommand{\titledscreen}[1]{\newpage{\large%
\bfseries\centering\color{emcolor} #1\[\[1ex]] \par}
\setlength{\parindent}{0.0em}
\setlength{\parskip}{1ex plus 0.5ex minus 0.2ex}
\raggedright
\pagestyle{plain}

```

The content of *manuscript.sty* is

```

\NeedsTeXFormat{LaTeX2e}[1994/06/01]
\ProvidesPackage{manuscript}[2004/04/10 David Allen,%
                                version 0.0]
\RequirePackage[printout,whitebackground]{texpower}
\newboolean{manuscript}
\newboolean{screen}
\newboolean{poster}
\setboolean{manuscript}{true}
\setboolean{poster}{false}
\setboolean{screen}{false}
\RequirePackage[letterpaper,nohead,margin=1in,
                footskip=1.5em]{geometry}
\pagestyle{plain}
\newcommand{\newscreen}{}
\newcommand{\titledscreen}[1]{\par}

```

The content of *poster.sty* is

```

\NeedsTeXFormat{LaTeX2e}[1994/06/01]
\ProvidesPackage{poster}[2004/04/10 David Allen,%
                        version 0.0]
\RequirePackage[printout,lightbackground,colorhighlight,
               coloremph,colormath]{texpower}
\newboolean{manuscript}
\newboolean{screen}
\newboolean{poster}
\setboolean{manuscript}{false}
\setboolean{poster}{true}
\setboolean{screen}{false}
\RequirePackage[paperwidth=27.78in,paperheight=20.83in,
margin=0.5in,truedimen,mag=\magstep3,nohead,nofoot]
                        {geometry}

\RequirePackage{multicol}
\setlength{\columnsep}{.75truein}
\setlength{\columnseprule}{2truept}
\raggedcolumns
\pagestyle{empty}
\setlength{\parindent}{0.0em}
\newcommand{\newscreen}{}
\newcommand{\titledscreen}[1]{\par}
\raggedright
\renewcommand{\section}{\@startsection{section}{1}{0pt}
{-2ex plus 1ex minus 1ex}{2ex plus 1ex minus 1ex}
{\color{emcolor}\centering\large\bfseries}}
\renewcommand{\subsection}{\@startsection{subsection}
{2}{0pt}{-2ex plus 1ex minus 1ex}{2ex plus 1ex minus 1ex}
{\color{emcolor}\centering\large\bfseries}}
\renewcommand{\subsubsection}{\@startsection{subsubsection}
{3}{0pt}{-2ex plus 1ex minus 1ex}{2ex plus 1ex minus 1ex}
{\color{emcolor}\centering\normalsize\bfseries}}

```

The source code for an example is shown here. The input file, *body.tex*, is the same for all cases. One of the packages discussed here must be used. The begin and end multicols\* must be active for poster output.

```

\documentclass[12pt]{article}
\usepackage{graphicx}
\usepackage{verbatim}
\usepackage{color}
% Un-comment one of the next three lines
%\usepackage{screen}
\usepackage{manuscript}
%\usepackage{poster}
\pagestyle{plain}
\bibliographystyle{plain}

```

```

\begin{document}
%\title{\color{emcolor}Screen Presentations, Manuscripts,
\title{Screen Presentations, Manuscripts,
and Posters from the Same \LaTeX\ Source}
\author{David M. Allen\University of Kentucky\
http://www.ms.uky.edu/$\sim$allen/}
\maketitle
\thispagestyle{empty}
\ifthenelse{\boolean{poster}}{\begin{multicols*}{5}}{}
\tableofcontents
\input{body}          % input or include your stuff here
%\newscreen
%\bibliography{latex} % list your bib files here
\ifthenelse{\boolean{poster}}{\end{multicols*}}{}
\end{document}

```

## 7 Sample output

This is a reduced image of a screen in my  $\text{\LaTeX}$  tutorial:

**Continued fractions**

The input

```

\[
a_0 + \frac{1}{a_1 + \frac{1}{a_2 +
\frac{1}{a_3 + \cdots}}}
\]

```

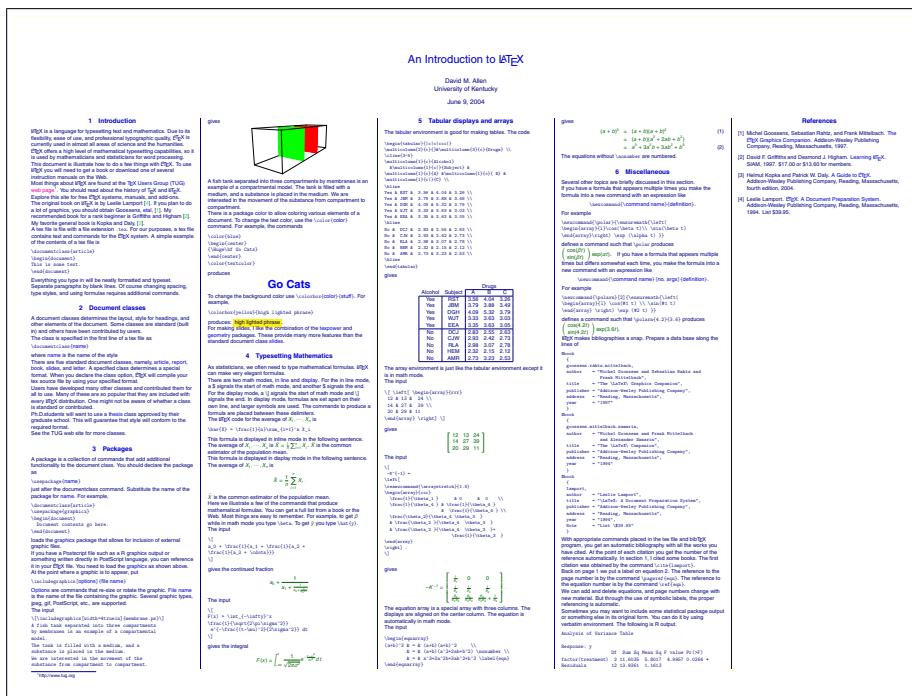
gives the continued fraction

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \dots}}}$$

22

The actual size is 11 by 8.5 inches.

This is a reduced image of my  $\text{\LaTeX}$  tutorial as a poster:



The actual size is 48 by 36 inches.



# TeX and XML

Baden Hughes  
Department of Computer Science and Software Engineering  
University of Melbourne  
badenh@cs.mu.oz.au



# Agenda

- There's Nothing New Under the Sun
- Typology of TeX and XML Relations
- The State of the Art and Challenges
- Recent Developments
- Conclusion



Practical TeX 2004

2

# There's Nothing New Under The Sun



- Even before XML, there were efforts to integrate structured data and typographically oriented software
- Many XML predecessors were seen as complementary to TeX workflows
  - SGML via LaTeX (1998)
  - SGML + DSSSL via JadeTeX (1998)
  - gf (1996)
  - format (1998)
  - Typeset (1995)
  - SDC (1996)
  - derivative markup standards such as TEI via LaTeX (1997)
- As web-friendly information sources grew rapidly, other formats such as HTML also underwent evaluation from a TeX perspective, resulting in new applications which could natively handle this data
- And for XML specifically ...



Practical TeX 2004

3

# There's Nothing New Under The Sun ... Still



- Articles and Conference Papers in the last couple of years (a non-exhaustive list)
  - TUGboat articles
    - 23(1) ConTeXt
    - 22(3) GELLMU
    - 21(3) XMLTeX, PassiveTeX
    - 20(4) TeXML
    - 20(3) TeXML, LaTeX to XML
  - Other User Group Journals and Conference Proceedings
    - Cahier GUTenberg 35-36 (English, French)
    - BachoTeX 2003, 2000 Conference Proceedings (English, Polish)
    - SLT 2004, 2003, 2002, 2001 Conference Proceedings (Czech)
    - EuroTeX 2001 Conference Proceedings (English)
- We can conclude that integrating XML data with TeX processing is just another logical step ...



Practical TeX 2004

4




## Caveat Emptor!

- A fundamental assumption is that the XML under consideration is always valid and able to be validated in demand – we all know that TeX does not gracefully handle with syntax errors!
  - This is especially important when using XML sources which include features such as namespace declarations and URIs ...
- Another implicit assumption is that a standard XML toolkit is available locally on the system



Practical TeX 2004

5



## Typology of TeX and XML Relations

- There's a range of integration approaches for TeX and XML
- Here we'll consider the main ones, which are:
  - XML in TeX Documents
  - XML as Input to TeX
  - XML as Output from TeX
  - TeX as XML Intermediary



Practical TeX 2004

6





## XML in TeX Documents

- Probably the easiest way is to include raw XML:  
`\usepackage{verbatim}`
  - But this approach requires manual line breaking, alignment etc, and isn't scalable
- A better way is to use an XML package:  
`\usepackage{xml}`
  - <http://www.doc.ic.ac.uk/~pjm/automated/resources/xml.sty>
  - Also handles XML-like documents – DTD, XSD, XSL etc
- NB: it's worth using a pretty-printer on your XML source prior to importing it into TeX documents
  - <http://tidy.sourceforge.net> (works for HTML too!)



Practical TeX 2004

7



## XML as Input to TeX

- ConTeXt is a full TeX system which supports native XML input
  - <http://www.pragma-ade.com/>
- `xmltex` is another possibility
  - Essentially a parser package for XML, where parser events can be used to trigger TeX typesetting code
  - Native `xmltex` executables as well as integration with LaTeX
  - Does require a little TeX hackery
  - <http://www.dcarlisle.demon.co.uk/xmltex/manual.html>



Practical TeX 2004

8



## XML as Output from TeX

- Where you have TeX sources and a standardised portable format is required
  - TeX source document conversion to XML is fairly standard
    - Tex4ht is the standout method
    - `<plug type="shameless">Eitan Gurari's talk earlier</plug>`
    - <http://www.cse.ohio-state.edu/~gurari/TeX4ht/>
  - Where TeX was used as the page composition engine, and DVI or PDF was the resulting output
    - Some commercial providers – expect to pay but high levels of automation and accuracy
      - PDF
        - API: P2X <http://www.pdf2text.com/convert-pdf-to-xml-component.htm>
        - Off the Shelf: CambridgeDocs <http://www.cambridgedocs.com/>



Practical TeX 2004

9



## TeX as XML Intermediary

- Fine-grained typographic control provided in TeX makes it the layout engine of choice in certain contexts
- XML/XSL rendering engines aren't currently in a state which competes on this level
- Fortunately there's a way that XML and XSL can leverage TeX to gain better rendering
- XML + XSL transformations have an intermediate form called Formatting Objects (FO's)
  - An output independent expression (similar to DVI in many ways)
  - Many ways to generate FO's including
    - XSLT
    - xalan (from Apache XML Project) + xt (by James Clark)
    - fop (by James Tauber)
    - Embedded solutions such as Apache's Cocoon
- PassiveTeX is specifically designed to take FO's as input :-)
  - <http://www.hcu.ox.ac.uk/TEI/Software/passivetex/>



Practical TeX 2004

10



## Recent Developments

- Some well-supported distributions include XML support off the shelf eg ConTeXt
  - `<plug type="shameless">Hans Hagen's class on Thursday!</plug>`
- TeX environments are now supporting more Web data functionality
  - `<plug type="shameless">Peter Flynn's class on Thursday!</plug>`
- Developers are moving beyond handling just basic XML to other XML-encoded or XML-derived data in TeX environments
  - XSL-FO's
  - RDF
  - Topic Maps
- Complementary improvements in TeX environment support for native XML features such as Unicode
  - Omega
  - XeTeX <http://scripts.sil.org/xetex>



Practical TeX 2004

11



## The State of the Art and Some Challenges

- Natural affinity between the formatting specifications of XML and the style and class models from TeX could be leveraged to provide closer integration between XML and TeX environments
- Fine-grained typographic control as provided by TeX is often an offline feature – contrasting with XML's typical assumption of an online environment
- Opportunities - as the XML data type becomes more pervasive, convergence of typographic technologies and web technologies is inevitable
- Threats - as the XML environment becomes more complex, it may be that TeX is not by default the leading page layout engine



Practical TeX 2004

12



## Conclusion

- TeX and XML: there's more than one way to do it ...
- XML support in default configurations of main TeX distributions is still emerging
- TeX offers compelling advantages in certain XML processing areas
- Widespread interest in TeX and XML integration should ensure that simple, robust XML handling applications from the TeX world will be with us in the near future
- Meanwhile, there's some hackery required, but quite a lot of hackers around :-)



## Questions ?



# MetaPlot, MetaContour, and Other Collaborations with METAPOST

Brooks Moses

Mechanical Engineering,  
Stanford University,  
Building 520,  
Stanford, CA 94305  
U.S.A.  
bmoses@stanford.edu

## Abstract

Most methods of creating plots in METAPOST work by doing all of their calculations in METAPOST, or by doing all of their calculations in a preprocessing program. There are advantages to dividing the work more equitably by doing the mathematical and data-visualization calculations in a preprocessing program and doing the graphical and layout calculations in METAPOST. The MetaPlot package provides a standard, flexible, interface for accomplishing such a collaboration between programs, and includes a general-purpose set of formatting macros that are applicable to a wide range of plot types. Examples are shown of linear plots with idiosyncratic annotation and two-dimensional contour plots with lines and filled contours on a non-cartesian mesh.

## 1 Introduction

One of the challenges of scientific writing in  $\text{\TeX}$  (or in  $\text{\LaTeX}$ ) is producing figures that are of comparable quality to the typesetting. These figures often include plots and graphs that represent mathematically-intense visualization of large data files, implying that some form of specialized program must be used to create them. They also typically contain labels, notes, and other text that should be typeset in a manner consistent the rest of the document, which requires using  $\text{\TeX}$ 's typesetting engine.

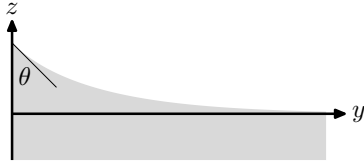
Traditionally, programs that meet these goals have taken one of two approaches. The first approach, used by programs such as ePiX[1] and Gnuplot[2], is to implement the program in a “traditional” programming language such as C++ or FORTRAN, and produce the complete figure as output in  $\text{\TeX}$ /eepic or METAPOST code, which is then post-processed. The other approach, taken by METAPOST's `graph` package and m3D[3], is to implement the program directly in METAPOST's macro language.

There are advantages and tradeoffs related to both of these approaches. Programming in METAPOST allows one to work directly with the language features such as declarative equations and ability to measure the size of typeset text, and thus allow the user to specify the figure layout in an intuitive, simple, and flexible manner. However, pro-

gramming in a traditional language allows one to write mathematically-intensive programs that use floating-point numbers and can be compiled rather than run slowly through an interpreter; in addition, it may allow one to take advantage of existing visualization libraries, or to provide an interactive user interface.

This paper describes an intermediate approach, which combines benefits from METAPOST programs and programs in more traditional languages. The initial data processing is done with a program written in a traditional language, which produces a METAPOST output file containing the processed data in an encapsulated form. This processed data is then fed into a set of METAPOST formatting macros, and the scaling, drawing, and annotation of the plots is all done by user-written commands within METAPOST.

Creating plots in two steps in this manner has several advantages: The initial data visualization can be done in a special-purpose program that uses a programming language and code libraries intended for substantial computations, without the need to implement more than a very simple output routine; the METAPOST macros for formatting plots and arranging them within a figure are largely independent of the details of the plots they are working with, and can be written in a generic manner suitable for widespread distribution; and the layout of any given



**Figure 1:** A capillary surface on a liquid touching a solid wall, after Batchelor [4].

figure can be done using the same processes as for a native-METAPOST drawing.

## 2 A Simple Example

Consider, by way of example, a plot of the shape of a meniscus formed by a liquid surface meeting a solid wall as shown in Figure 1. The surface curve is given by a somewhat complicated expression involving inverse hyperbolic cosines,<sup>1</sup> and is representative of calculations that would be easier to do with a traditional programming language.

The C++ program to produce this curve in a METAPOST format is straightforward. The most complicated part is the function to generate a string containing a METAPOST representation of a point, which we accomplish using the `<sstream>` standard library.

```
string mpoint(double x, double y) {
    ostringstream pointstring;
    pointstring.setf(ios_base::fixed,
                    ios_base::floatfield);
    pointstring.precision(5);
    pointstring << '(' << x << ', ' << y << ')';
    return pointstring.str();
}
```

The `setf` and `precision` commands set the numeric format for the stream (fixed-precision, five decimal places), and then the coordinates are fed into the stringstream with the appropriate punctuation, producing a result like `(0.01556,0.75006)`.

Given this and a `capillary()` function that computes the equation for the surface, creating the METAPOST command for the curve is simply a matter of looping through the points and dumping them to the standard output, with appropriate text before

<sup>1</sup> For those who are curious, the equation (from [4]) is

$$\frac{y}{d} = \cosh^{-1} \frac{2d}{z} - \cosh^{-1} \frac{2d}{h} + \left(4 - \frac{h^2}{d^2}\right)^{\frac{1}{2}} - \left(4 - \frac{z^2}{d^2}\right)^{\frac{1}{2}},$$

where  $h^2 = 2d^2(1 - \sin \theta)$  is the height of the meniscus,  $\theta$  is the contact angle, and  $d$  is a scaling parameter related to the surface tension and liquid density.

and after the loop to define the picture variable and close the curve into a cyclic path.

```
int main() {
    double theta = pi/4.0;
    double d = 1.0;
    double h = sqrt(2.0 * d*d * (1.0 - sin(theta)));
    double y, z;

    cout << "picture capillary;\n";
    cout << "capillary := nullpicture;\n";
    cout << "addto capillary contour "
         << mpoint(0.0, h);
    for(int i = 99; i > 2; i-- ) {
        z = (i/100.0) * h;
        y = capillary(z,h,d);
        cout << " .. " << mpoint(y, z);
    }
    cout << " -- " << mpoint(y, -0.5);
    cout << " -- " << mpoint(0.0, -0.5);
    cout << " -- cycle;\n";
}
```

This produces the following METAPOST code as output:

```
picture capillary; capillary := nullpicture;
addto capillary contour (0.00000,0.76537)
.. (0.00772,0.75771) .. (0.01556,0.75006)
% [...and so forth...]
.. (3.39322,0.02296) -- (3.39322,-0.50000)
-- (0.00000,-0.50000) -- cycle;
```

We can then follow this with additional METAPOST commands, which scale it to an appropriate size for printing on the page and draw axes and labels on it, in order to produce the plot shown in Figure 1.

```
beginfig(1)
draw (capillary scaled 0.5in) withcolor
0.85white;
linecap := butt;
pickup pencircle scaled 1pt;
drawarrow (0,-0.25in) -- (0, 0.5in);
label.top(btex $$$ etex,(0, 0.5in));
x1 := (xpart(lrcorner capillary) * 0.5in, 0)
+ (0.1in, 0);
drawarrow (0,0) -- x1;
label.rt(btex $$$ etex, x1);
pickup pencircle scaled 0.25pt;
x2 := ulcorner capillary scaled 0.5in;
draw ((0,0) -- (0.24in, -0.24in)) shifted x2;
label(btex $\theta$ etex,
x2 + (0.07in, -0.18in));
endfig;
end
```

Although this example produces a perfectly serviceable result, it has some noteworthy drawbacks. The scale factor of 0.5in does not have a clear relationship to the size of the plot, and producing a plot of a particular size would require measurement of the *capillary* picture and explicit computation of the scale factor. The locations of the annotations are likewise determined by explicit measurement, or by being typed in directly. If we were to change one of the parameters in the C++ program and re-run it, many of the values in the METAPOST code would need to be changed as well.

### 3 A more general example: the MetaPlot package

The MetaPlot package is designed to address many of the shortcomings of the example given in Section 2. It provides a consistent way of transferring the plot commands and associated metadata from the generating program into METAPOST, and direct handles for manipulating the plots within METAPOST using its normal idiom of declarative equations rather than procedural assignments.

To accomplish this in a general manner, we define two types of METAPOST data structures: *plot objects* and *plot instances*. A plot object is a plot “in the abstract,” containing paths, filled contours, and metadata that make up the plot (or a set of related plots), represented in a manner that is independent of the details of how the plot is positioned. By contrast, a plot instance is a plot “on the page,” containing parameters for the scaling and positioning of a given plot, and a reference to a parent plot object that gives the actual pictures to be drawn.

A typical preamble for a figure using MetaPlot will consist of an **input metaplot** command to load the MetaPlot macros, an **input** command to load the METAPOST file that contains the plot objects (typically an output file from the preprocessing program), and calls to the MetaPlot macros to generate plot instances from the plot objects.

#### 3.1 The concept of a “plot-object”

Suffix arguments and multi-token variable names in METAPOST allow us to define data structures that approximate structures or objects in more traditional programming. The correspondence is not exact; in particular, there is no data type associated with the overall object. METAPOST is simply passing around a fragment of a variable name and constructing complete variable names from it, so any arbitrary element can be added to the class without changing its type. Thus, the MetaPlot macros can deal with arbitrary types of plots in a generic man-

ner, so long as they meet a few minimal requirements that allow them to be scaled and positioned.

The paths and contours that make up a plot object are not defined in terms of the native data coordinates, but are rescaled to fit within a unit box (that is, extending from 0 to 1 in both coordinate directions), which is treated as the bounding box of the plot for purposes of scaling and positioning. As a result, the possibility of coordinates too small or too large for METAPOST’s fixed-point number representation is avoided; in addition, positioning the plot on the page is a simple matter of scaling by the final width and height and shifting by the final position of the lower-left corner. The original data scales are stored in four numeric components that record the values corresponding to the extents of the bounding box.<sup>2</sup>

The remaining details of the format can be shown by rearranging the example from Section 2 into a plot object, as follows. For purposes of later examples, we will presume that this has been saved as *capillary.mp*.

```
% Definition of capillary plot-object
% Picture components
picture capillary.fplot;
  capillary.fplot := nullpicture;
addto capillary.fplot contour (0.00000,1.00000)
  .. (0.00227,0.99395) .. (0.00459,0.98790)
  % [...and so forth...]
  .. (1.00000,0.41329) -- (1.00000, 0.00000)
  -- (0.00000, 0.00000) -- cycle;
picture capillary.lplot;
  capillary.lplot := nullpicture;
addto capillary.lplot doublepath
  (0.00000,1.00000) .. (0.00227,0.99395)
  % [...and so forth...]
  .. (1.00000,0.41329);

% Required metadata
numeric capillary.xleft; capillary.xleft = 0.0;
numeric capillary.xright;
  capillary.xright = 3.39322;
numeric capillary.ybot; capillary.ybot = -0.5;
numeric capillary.ytop; capillary.ytop = 0.76537;

% Other metadata
pair capillary.contactpoint;
  capillary.contactpoint = (0.0, 1.0);
numeric capillary.contactangle;
  capillary.contactangle = 45.0;
```

<sup>2</sup> Although these variables are represented here as numerics and thus are still vulnerable to under- or overflow, it would be a simple matter to replace them with string-represented numbers from the *sarith* package.

In this case, I have also added an additional component: this version of *capillary* contains a path for the liquid surface line (*capillary.lplot*), as well as the original filled contour (now *capillary.fplot*); the decision about which of them to draw can be made later. A plot object can contain any number of these pictures (even zero), with arbitrary names.

The four required scale variables are *capillary.xleft*, *.xright*, *.ybot*, and *.ytop*; these, for purposes of the MetaPlot macros, must be named thus.

Finally, there are two metadata variables, *capillary.contactpoint* and *capillary.contactangle*, which will be useful in drawing the annotations on this particular plot. These, again can be present in any number, and have arbitrary names. Of note is that *.contactpoint* is given in the same unit-box coordinate system that the paths and contours are in, allowing it to be positioned by the same macros that scale and position the picture components.

### 3.2 Creation of a plot instance

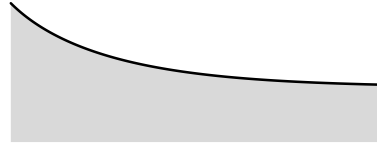
The next step after creating plot objects is manipulating them on the page by means of plot instances. A plot instance thus needs to contain three sets of components: coordinates and dimensions of the plot as shown on the page, a representation of the plot's internal scale for use in alignment and producing axes, and a means of accessing picture components from its parent plot object. These are created by the *plot\_instantiate()* macro, which is part of MetaPlot; the version below is simplified somewhat.

```
% Args: inst is the new plot instance.
% plot_object is the parent plot object.
def plot_instantiate(suffix inst)
  (suffix plot_object) =

% Define (unknown) parameters for plot-instance
% location on page
  numeric inst.pagewidth, inst.pageheight;
  numeric inst.pageleft, inst.pageright,
    inst.pagetop, inst.pagebottom;
  inst.pageleft + inst.pagewidth = inst.pageright;
  inst.pagebottom + inst.pageheight = inst.pagetop;

% Define (known) parameters for plot's scaling
  numeric inst.scaleleft, inst.scaleright,
    inst.scaletop, inst.scalebottom;
  inst.scaleleft := plot_object.xleft;
  inst.scaleright := plot_object.xright;
  inst.scalebottom := plot_object.ybottom;
  inst.scaletop := plot_object.ytop;

% Pointer-function to plot_object's plots, scaled
```



**Figure 2:** The capillary surface, in its unadorned form as plot object elements scaled to 2.0in by 0.75in.

```
% and positioned.
vardef inst.plot(suffix name) =
  plot_object.name xscaled inst.pagewidth
  yscaled inst.pageheight
  shifted (inst.pageleft, inst.pagebottom)
enddef;
enddef;
```

Note that, immediately after a plot instance is created, the page information is unknown and the scale information is known.

We can now start putting plot objects on the page in a limited fashion, by assigning known values to the unknown page information, and then drawing the scaled picture elements.

```
input metaplot % MetaPlot macros
input capillary % capillary plot object

plot_instantiate(plotA, capillary)
plotA.pageleft = 0.0;
plotA.pagebottom = 0.0;
plotA.pagewidth = 2.0in;
plotA.pageheight = 0.75in;
beginfig(2)
  draw plotA.plot(fplot) withcolor 0.85white;
  draw plotA.plot(lplot)
  withpen pencircle scaled 1pt;
endfig;
end
```

The result of this is shown in Figure 2. Note that the color of the filled plot and the line size for the line plot are specified in the draw command, rather than in the plot object.

### 3.3 Manipulation of plot-objects

The bare plot instances are of little use without a set of macros for manipulating them. We start with a macro to set the *x*-axis and *y*-axis scales to equal values:

```
def plot_setequalaxes(suffix inst) =
  inst.pagewidth = inst.pageheight
  * ((inst.scaleright - inst.scaleleft)
    / (inst.scaletop - inst.scalebottom));
```



**enddef;**

This is written so that the page-related variables do not appear in the denominator of fractions, because either one (or both) of them may be unknown when the macro is called, and METAPOST can only solve linear equations.

There are also a set of macros for converting between locations expressed in the plot's coordinates and locations on the page. For example,

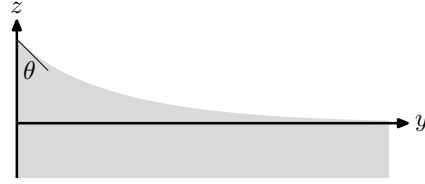
```
def plot_xpageloc(suffix inst)(expr scalex) =
  inst.pageleft + (scalex - inst.scaleleft)
  * (inst.pagewidth
    / (inst.scaleright - inst.scaleleft));
enddef;
```

The additional macros in this series are *ypageloc*, *zpageloc* (which takes an *x* and a *y* coordinate as input, and returns a point), and *xscaleloc* and *yscaleloc* for the reverse direction of converting from a page location to a plot coordinate.

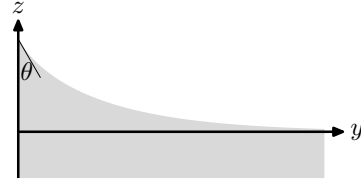
With these, we have most of what we need to manipulate plots in an intuitive way. For instance, consider the figure from Section 2, which can now (with some small changes) be written in a much more general way as

```
input metaplot    % MetaPlot macros
input capillary  % capillary plot object
```

```
plot_instantiate(plotB, capillary)
plot_setequalaxes(plotB);
plotB.pageleft = 0.0;
plotB.pagebottom = 0.0;
plotB.pageheight = 0.75in;
beginfig(3)
  draw plotB.plot(fplot) withcolor 0.85white;
  linecap := butt;
  pickup pencircle scaled 1pt;
  % z-axis (vertical)
  z1 = (plotB.pageleft, plotB.pagebottom);
  z2 = (plotB.pageleft, plotB.pagetop + 0.1in);
  % y-axis (horizontal)
  z3 = (plotB.pageleft, plot_ypageloc(plotB,0.0));
  z4 = (plotB.pageright + 0.1in,
    plot_ypageloc(plotB,0.0));
  drawarrow z1 -- z2;
  label.top(btex  $\theta$  etex, z2);
  drawarrow z3 -- z4;
  label.rt(btex  $y$  etex, z4);
  pickup pencircle scaled 0.25pt;
  % Label for contact angle
  z5 = plotB.plot(contactpoint);
  z6 = z5 + 0.24in
    * dir(-90 + capillary.contactangle);
```



**Figure 3:** The capillary surface, with equal *y* and *z* scales, a page height of 0.75in, and appropriate annotations.



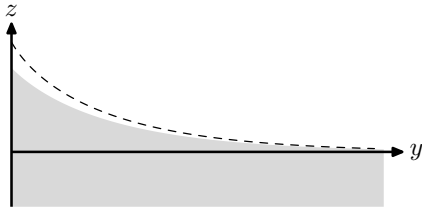
**Figure 4:** The capillary surface with parameters and page height as in Figure 3, but with  $\theta = \pi/6$ .

```
z7 = z5 + 0.18in
  * dir(-90 + 0.5*capillary.contactangle);
draw z5 -- z6;
label(btex  $\theta$  etex, z7);
endfig;
end
```

The result of this is shown in Figure 3. We can demonstrate that this is flexible by adjusting the value of  $\theta$  to  $\pi/6$  rather than  $\pi/4$ , and recreating the figure using exactly the same files; the result is shown in Figure 4. Note that changing the contact angle raises the contact point, making the plot taller in scale coordinates; thus, it is drawn at a smaller scale to maintain the 0.75-inch page height.

Having two figures in this way is not the clearest way to compare the two plots, particularly with the differences in scale. A better approach is to overlay them at the same scale, making use of the existence of the filled plot from one plot object and the line plot from the other to provide a visually clear result. A simple way of placing both plots on the same coordinate axes is to require that their (0,0) and (1,1) points coincide on the page, which we do by means of the *plot\_zpageloc* command; the remainder of the file is as much in the previous plots, although there is a little additional code in making certain that the axis-arrows cover both plots.

```
input metaplot    % MetaPlot macros
input capillary  % capillary plot object
input capillary2 % capillaryb plot object
```



**Figure 5:** Two capillary surfaces, as in Figure 3 and Figure 4, showing the difference in the curves as a result of varying  $\theta$ .

```

plot_instantiate(plotB, capillary)
plot_setequalaxes(plotB);
plotB.pageleft = 0.0;
plotB.pagebottom = 0.0;
plotB.pageheight = 0.75in;

plot_instantiate(plotC, capillaryb)
plot_zpageloc(plotB, 0.0, 0.0)
= plot_zpageloc(plotC, 0.0, 0.0);
plot_zpageloc(plotB, 1.0, 1.0)
= plot_zpageloc(plotC, 1.0, 1.0);

beginfig(5)
  linecap := butt;
  pickup pencircle scaled 1pt;
  draw plotB.plot(fpplot) withcolor 0.85white;
  draw plotC.plot(lpplot) dashed evenly
  withpen pencircle scaled 0.5pt;
  % z-axis (vertical)
  z1 = (plotB.pageleft, plotB.pagebottom);
  z2 = plotB.pageleft;
  y2 = max(plotB.pagetop, plotC.pagetop) + 0.1in;
  % y-axis (horizontal)
  z3 = (plotB.pageleft, plot_ypageloc(plotB,0.0));
  x4 = max(plotB.pageright, plotC.pageright) + 0.1
    in;
  y4 = plot_ypageloc(plotB,0.0);
  drawarrow z1 -- z2;
  label.top(btex $z$ etex, z2);
  drawarrow z3 -- z4;
  label.rt(btex $y$ etex, z4);
endfig;
end

```

The result of this is shown in Figure 5.

### 3.4 Creation of axes

Any quantitative graph is meaningless without grid-labels for the coordinate axes, and so MetaPlot includes macros to create them. Unlike METAPOST's `graph.mp` package, MetaPlot's axis-drawing functionality requires that the user specify most of the

details of the formatting, with the benefit of having a much more flexible implementation.<sup>3</sup>

The core of the axis-drawing functionality is a set of macros for creating generic tickmarks, labeled tickmarks, rows of tickmarks, and so forth, which are included with MetaPlot in a `axes.mp` file (and thus, for consistency, are prefaced with `axes_` rather than `plot_`). These are interfaced to the plot object coordinates by the `plot_xtickscale` and `plot_ytickscale` macros.

```

def plot_xtickscale (suffix inst)
  (expr startpoint, endpoint,
   ticklength, tickspace, tickdir,
   tickzero, tickstep, ticklabelformat) =

  axes_tickscale (
    startpoint, % First endpoint of the tickrow
    endpoint,   % Second endpoint of the tickrow
    ticklength, % Length of tickmarks
    tickspace, % Space between tickmark and label
    tickdir,   % Tickmark direction
    plot_xscaleloc (inst)(xpart(startpoint)),
                % Coordinate value at first endpoint
    plot_xscaleloc (inst)(xpart(endpoint)),
                % Coordinate value at second
                endpoint
    tickzero,   % Coordinate value for a known
                % tick location
    tickstep,  % Coordinate space between ticks
    ticklabelformat
                % Format for tick labels
                % (syntax from format.mp package)
                % (use "" for no tick labels)
  )
enddef;

```

The `plot_ytickscale` definition is nearly identical. Note that these macros do not actually draw the tickmarks; they return a picture object, which can then be explicitly drawn or otherwise manipulated.

A simple way of adding grid labels to the previous example would be the following:

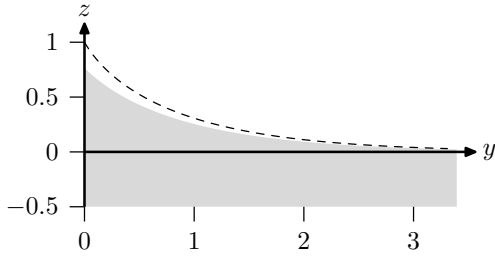
```

beginfig(6)
  % [...repeat of definitions of fig(4)...]

  x5 = plotB.pageleft;
  x6 = x4;
  y5 = y6 = plotB.pagebottom;
  draw plot_xtickscale(plotB)(z5, z6,
    0.08in, 0.06in, down, 0.0, 1.0, "%3f")

```

<sup>3</sup> There is, of course, no need for flexible implementations and simple interfaces to be mutually exclusive, and functions for more automated axes may be included in MetaPlot as it continues to be developed.



**Figure 6:** A repeat of Figure 5, with simple grid labels added.

```

withpen pencircle scaled 0.5pt;
y7 = plotB.pagebottom;
y8 = y2;
x7 = x8 = plotB.pageleft;
draw plot_ytickscale(plotB)(z7, z8,
    0.08in, 0.06in, left, 0.0, 0.5, "%3f")
withpen pencircle scaled 0.5pt;
endfig;
    
```

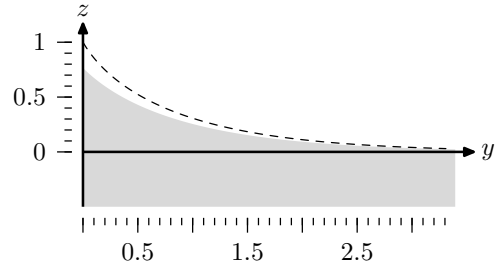
The results of this are shown in Figure 6. As can be seen with the  $x$ -axis, the `tickscale` macros do not include the axis-lines themselves, thus allowing the user to draw them with a different line style than that used for the ticks, or to leave them off entirely.

For a more polished look, we can move the grid ticks a small distance away from the plot, limit the  $y$ -axis range to the region that has meaningful significance, and add intermediate ticks without labels. In addition, this example illustrates the use of the `tickzero` parameter to start the labeled  $x$ -axis ticks at 0.5 rather than zero.

```

beginfig(7)
% [...repeat of definitions of fig(4)...]

x5 = plotB.pageleft;
x6 = x4 - 0.1in;
y5 = y6 = plotB.pagebottom - 0.06in;
draw plot_xtickscale(plotB)(z5, z6,
    0.08in, 0.06in, down, 0.5, 1.0, "%3f")
withpen pencircle scaled 0.5pt;
draw plot_xtickscale(plotB)(z5, z6,
    0.08in, 0.06in, down, 0.0, 1.0, "")
withpen pencircle scaled 0.5pt;
draw plot_xtickscale(plotB)(z5, z6,
    0.04in, 0.06in, down, 0.0, 0.1, "")
withpen pencircle scaled 0.5pt;
y7 = y4;
y8 = y2 - 0.1in;
x7 = x8 = plotB.pageleft - 0.06in;
draw plot_ytickscale(plotB)(z7, z8,
    0.08in, 0.06in, left, 0.0, 0.5, "%3f")
    
```



**Figure 7:** A repeat of Figure 5 again, with more advanced grid labels.

```

withpen pencircle scaled 0.5pt;
draw plot_ytickscale(plotB)(z7, z8,
    0.04in, 0.06in, left, 0.0, 0.1, "")
withpen pencircle scaled 0.5pt;
endfig;
    
```

This is shown in Figure 7.

### 3.5 MetaContour: a C++ program for contour plots

Now that the METAPOST side of the collaboration has been described in some detail, we return to the matter of programs that generate plot objects as output. One of the particular reasons for developing MetaPlot was to have a way of producing contour plots, and so the MetaPlot package comes with a C++ program, MetaContour, for creating them.

The internals of MetaContour are beyond the scope of this paper, but it does make use of one additional capability of plot objects that is worth noting—the ability to include color information. The plot object is defined with commands like the following, with color directives.

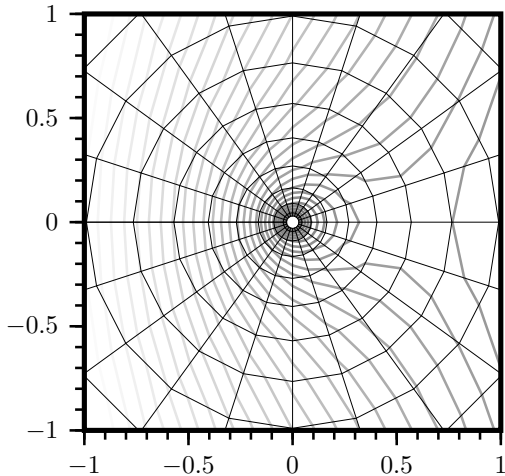
```

picture contplotA.LinePlot;
contplotA.LinePlot := nullpicture;
addto contplotA.LinePlot doublepath
    (0.48075,0.50000)-- (0.48163,0.50597)
withcolor contourcolor27;
addto contplotA.LinePlot doublepath
    (0.48420,0.50000)-- (0.48492,0.50490)
withcolor contourcolor28;
addto contplotA.LinePlot doublepath
    (0.45994,0.50000)-- (0.46169,0.51245)
withcolor contourcolor23;
% [...and so forth...]
    
```

Then, before the plot object file is read into the main METAPOST file, the `contourcolor` array is defined as desired.

```

% Contour colors for grayscale scheme
color contourcolor[ ];
    
```



**Figure 8:** Sample graph created by MetaContour and MetaPlot, showing potential lines for a combination of a linear gradient and a point source, plotted on a polar grid.

```
contourcolor0 = 1white;
contourcolor1 = 0.98white;
% [...and so forth...]
contourcolor30 = 0.4white;
```

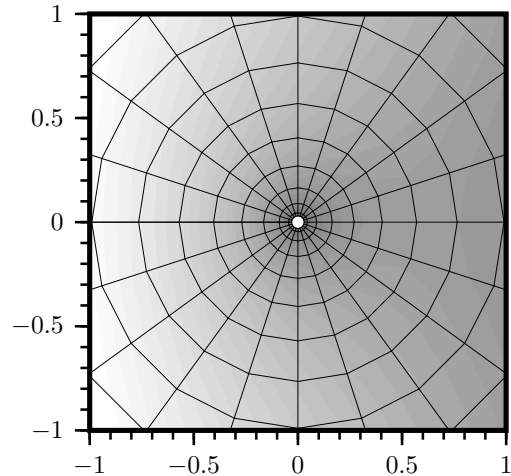
Thus, each line of the contour plot is associated with a color, and it will be drawn in that color unless it is overridden by another color directive; for instance, if we wanted to plot the contour lines all in black, we could do so simply by specifying:

```
draw continstA.plot(LinePlot) withcolor black;
```

Aside from the color contour-line plot just described, the MetaContour output contains a filled contour plot, and an image of the mesh of data points. Some examples of these are shown in Figure 8 and Figure 9; although these are much more complex than the examples from preceding sections, the MetaPlot commands used to generate them are nearly identical.

#### 4 Conclusion

The examples that have been shown illustrate only a small sampling of the capabilities of MetaPlot. In using METAPOST to generate the figures, it provides an easily extensible layout capability that is not limited by the imagination of the package author. The standardized plot-object interface simplifies the process of writing plot-generation programs, as they can leave the details of layout and annotation to the MetaPlot postprocessing.



**Figure 9:** Another sample graph created by MetaContour and MetaPlot, illustrating a filled contour-plot style rather than using contour lines.

At the time of this publication, MetaPlot and MetaContour should be available from CTAN in the `/graphics/metaplot` directory. They are still very much works in progress; I look forward to suggestions and improvements, and hope that others will find them to be useful tools.

#### References

- [1] Hwang, A., ePiX, <http://mathcs.holycross.edu/~ahwang/current/ePiX.html>.
- [2] Gnuplot, <http://www.gnuplot.info>.
- [3] Phan, A., m3D, <http://www-math.univ-poitiers.fr/~phan/m3Dplain.html>.
- [4] Batchelor, G. K., *An Introduction to Fluid Dynamics*, Cambridge University Press, 1967.

# T<sub>E</sub>X and Scripting Languages

William M. Richter

Texas Life Insurance Company

900 Washington Avenue

Waco, TX 76703

wrichter@texaslife.com

## Abstract

T<sub>E</sub>X is an ASCII text-based markup language. In a scheme of automated document preparation TeX provides the foundation. The idea is for programs to do the work of 1) Generating the T<sub>E</sub>X code for documents, 2) Running T<sub>E</sub>X on these documents, and 3) Post-processing the resulting .dvi files to obtain the finished documents. Resulting PostScript documents may be further post-processed to produce files that exploit the output capabilities of various printers. Discussed herein are the techniques and benefits of such a scheme and how scripting languages (those languages outside the traditional edit/compile/link/run cycle) can make the whole process fun and easy.

## Introduction

In his web essay, *Hackers and Painters*[2], Paul Graham equated the much maligned and misunderstood activity of “hacking” [6] with the long-esteemed tradition of painting (e.g. portrait painting, as opposed to painting of porches, peeling house trim, and such). He observed that what, today, we acknowledge as masterworks actually evolved during the artist’s act of creation from a sketch, the details only gradually being filled in, to a finished, glorious work of art. He argued that a writer goes through the same process of refinement, starting from rough outline or foggy idea until she finds nothing which needs refining. One reason T<sub>E</sub>X is appealing to authors is that it makes the process of refinement secondary. The tasks of *creation* (thinking is hard work for most of us) and *presentation* are orthogonal. Moreover the presentation task is assumed almost entirely by T<sub>E</sub>X<sup>1</sup> One can, after all, create a T<sub>E</sub>X document that is 90% complete using nothing more than a tool as simple as NotePad. The implication being that simple tools equate to less loss of creative energy.

Graham believes that authors of computer code (programmers, we often call them) follow the same nonlinear/circuitous paths of painters and authors. Seldom, if ever, is software conceived of and implemented by following in a direct route from beginning to end. Most great software, Graham claims, is the product of hacking, that the implications for soft-

---

<sup>1</sup> Except when we T<sub>E</sub>X-nicians decide we know better and begin to muck around in T<sub>E</sub>X’s own internal affairs.

ware design are significant, and that what a computer language is and how an author interacts with it defines the end result. In his view it means...

...a programming language should, above all, be malleable. A programming language is for thinking of programs, not for expressing programs you’ve already thought of. It should be a pencil, not a pen.

And he continues,

We need a language that lets us scribble and smudge and smear, not a language where you have to sit with a teacup of types<sup>2</sup> balanced on your knee and make polite conversation with a strict old aunt of a compiler.

A class of programming languages, called “scripting languages,” is compatible with Grahams ideas of what a hacker’s language should be. “Malleable” in nature, and easy to *think with*, scripting languages are similar in spirit to T<sub>E</sub>X. Indeed, T<sub>E</sub>X itself may even be considered as a scripting language for typesetting.

So, on the one hand, we have T<sub>E</sub>X, a tool which lets authors “scribble and smudge and smear” about with their ideas. On the other hand we have hackers using scripting languages pursuing similar creative avenues. The question then arises, “What happens

---

<sup>2</sup> For readers unfamiliar with the art of computer programming, the teacup of types to which he is referring will be addressed in a subsequent section on the attributes of scripting languages where static vs. dynamic data types are discussed.

if these two tools are combined and used in a collaborative effort?” We now explore various ways that  $\text{\TeX}$  and scripting languages can be combined.

### Scripting Languages

Before delving into scripting languages proper, let us review a few of the attributes of traditional computer languages (Paul Graham’s compiler that he refers to as a strict old aunt).

### Traditional Computer Languages

For readers unfamiliar with the art of authoring computer software (programming computers) here is what programmers do: they think of a task that computers can accomplish better than humans (say, typesetting, for example). Then they sit and think, potentially at length, about how humans would go about doing that task, and how to express those steps algorithmically[3]. After sketching said algorithm, they formalize and codify it in a so-called “language” that is a sort of half-way meeting ground between the way humans think and the way computers operate. This prose, called a program, consists of two distinct entities: variables, which declare what it **is** that the computer will be working on, and imperative procedures that define what is to be **done** to that data.

Some salient details about these traditional languages:

1. The variables: Computer hardware can work with data in different formats: numbers (integers and real numbers), strings of character data, etc. Each variable in a program must be defined in advance of its use to be of a specific type. In computer science lingo this is called *static typing*.
2. The code: Codifying an algorithm in a particular computer language isn’t really enough for computer hardware. More work must be done. This language must be converted by John Graham’s compiler “aunt” in to “machine code” on which the computer’s logic circuits can act.
3. But even the work of the compiler-aunt isn’t enough. The fruit of her strict dominance must then be *linked* with the work of other compiler-aunts to produce a final collection of unreadable “goo” that only a computer can understand (machine code is unreadable to all but the most deviant of human brains).
4. Nor is this the end of the story. When an edited/compiled/linked program (called an executable) has finally been produced and a blazingly fast 3-Gigahertz CPU is unleashed to ex-

ecute it the first time, the most likely end result is either an almost immediate decision by the CPU that its human programmer is capable only of producing flawed code for it to execute (it communicates this fact by printing some rude message like “**Segmentation Violation**” and producing a very large file on disk containing the entire contents of its memory), or it lapses into a seemingly semi-comatose state consuming large amounts of CPU time until its programmer/master gets its attention with violence of the **kill** command.

One can see a definite “cycle of pain”: *Edit, Compile, Link, Test* that must be repeated many times until a flawless executable is produced. No wonder computer programming is seen by many an outsider as a black art to be pursued by only the most intrepid and determined souls.

### Why Scripting Languages are Better, and Why More People Should be Hackers

Scripting languages[9] shrink the cycle of pain to *Edit, Test*. With the cruffy old compiler-aunt gone, the whole process of software development proceeds in a more efficient and pleasant manner with attention shifting to the “creative,” editing part and the refinement, or testing part. But measure of pain is not the only attribute that makes scripting languages attractive. Other import attributes are:

1. Simple syntax,
2. High-level data types,
3. Loosely typed,
4. Standard control structures: if/else, while, for,
5. Interfaces well with host operating system,
6. Plays well with external entities,
7. Embeddable inside more complex systems,
8. Often used as “glue” languages to link multiple standalone applications and tools together,
9. Requires a runtime interpreter to execute the script,
10. Compiles to bytecode which executes on a virtual machine,
11. Often ‘dynamic’ in nature.

We need to expound on a few of these points:

**Simple Syntax** If a language is to satisfy Graham’s requirement that it be a malleable pallet for the smearing and smudging of ideas, it cannot be verbose (we don’t want to spend time typing). So scripting languages (**SLs** (I’m tired of typing, too)) are succinct in nature; able to convey a significant

amount of procedural instruction in as few words as necessary to maintain clarity of meaning.<sup>3</sup>

**High-level data types** The concept of high-level data types parallels simple syntax. Just as with the need to state procedural algorithms in a succinct fashion, we also need constructs that allow for the representation of bundles of data that may be arbitrarily complex. We demand more than simple integer, floating point, and strings of character data that traditional languages like C and C++ provide.<sup>4</sup> Usually these higher-level data types come in the form of lists and dictionaries; containers that hold other data elements and allow for the expression of relationships between our data.

**Loosely Typed / Dynamic Nature** Discussion of esoteric topics like *Strongly vs. Loosely Typed Data* and *Early vs. Late Binding* is more than can be discussed here.[1] Some understanding is essential, however. Earlier, we pointed out that in traditional languages, each element of data that a program will use (its variables) must be defined to exist as a particular type before it can be used.<sup>5</sup> Moreover, as variables are passed between parts of a program (function calls) the type of each variable passed must match exactly the type expected by the called function. This check is done by the strict old compiler-aunts, and was designed to keep programmers from making errors that would only manifest themselves during the test phase. Strict type checking makes a lot of sense with traditional languages. However, with dynamic SLs, there is a critical difference, rooted in the 'dynamicness' of the language. SLs never declare variables. Variables are created or 'allocated' (on-the-fly, so to speak) when they are first referenced. When a variable is allocated it is associated with a particular type that is implied from the context in which it was initially used. The association to type is permanent and observable. So not only can one ask, "What *value* does a variable contain?", one can also make an inquiry about its *type*. For example, the statement `A = 123` allocates a data element called A whose value is 123 and whose type is integer. The statement `B = 3.14` allocates a variable called B whose value is 3.14 and whose type is floating point. B was made a variable of type

<sup>3</sup> The language APL comes to mind, but perhaps not THAT succinct. It would be nice for non-hackers to be able to read and understand our prose, too.

<sup>4</sup> Admittedly, C, C++, and other traditional languages may be made to represent arbitrarily complex data, but those types are not intrinsic in the language.

<sup>5</sup> This isn't actually true. Data elements may be dynamically allocated in tradition languages, but this introduces additional complexity in both the design and debugging steps.

floating point because, contextually, the statement contained a decimal point in the value implying a floating point value. Had we desired A to be a floating point variable we would have coded `A = 123.0`.

This leads to a new world of ways in which to think about writing code. Functions, now dynamic in nature, can easily accept an arbitrary number of arguments, the type of each being one of a range of possible types. Depending on the number and type of variables passed to a function, the function may act in different ways. This goes to the heart of malleability. In the creative process if we change our mind and decide to "smudge and smear" in a different direction, our existing code may not go to waste. It may be possible just to extend it to conform to our new conditions. A world of new and easier programming languages, the SLs, may also introduce hacking to a wider audience. Whereas the "old world" traditional languages excluded or intimidated many people for the reasons above (there are, after all, only so many work hours in a day), SLs remove the complexity of programming and make hacking the creative process that it should be.

Finally there is another reason more people (at least for those who must live with a computer) should become hackers. While most of us are not master software developers, developing cathedral-size financial accounting packages, for example, we do a surprising amount of "sketch" work (in Graham's paradigm) and having skills to write small programs can be effective.

## Real Scripting Languages

A mid-June *google-search* of the keywords, `script language programming` returned approximately 1,570,000 hits. Top-ranked pages returned from a search of keywords `scripting languages` reside on the sites:

1. [www.php.net](http://www.php.net)
2. [www.python.org](http://www.python.org)
3. [www.ruby-lang.org](http://www.ruby-lang.org)
4. [www.perl.org](http://www.perl.org)

All these websites are homes of import scripting languages. And there are more SLs; many more... a veritable zoo of them with names like: Awk, JavaScript, Lisp, Lua, Perl, PHP, Python, Rebol, Ruby, Small, Groovy, Tcl. If one were to rank SLs in order of popularity, the top of that list would include:<sup>6</sup>

- Perl
- Python
- Tcl/Tk

<sup>6</sup> Not listed in order.

- JavaScript
- Unix shell scripts (sh/bash/csh/etc.)

Indeed several of these **SLs** have outgrown the group’s scripting origins and have gone on to become “general purpose programming languages of considerable power.” [5] The only argument for continuing to use the term “scripting-language” is the lack of a better term.

### A Particular Scripting Language: Python

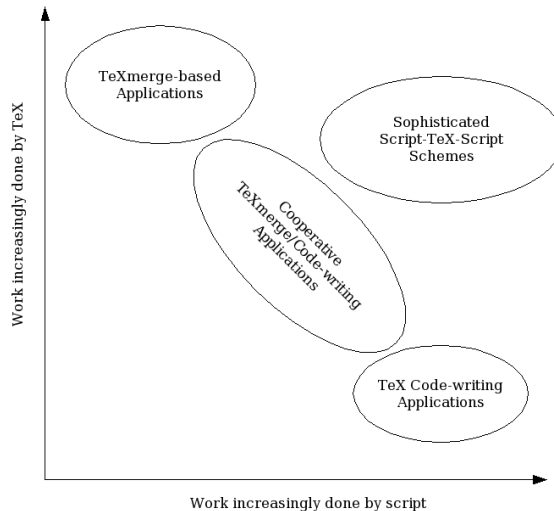
Chapter one of the official Python Tutorial reads:<sup>7</sup>

Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than the shell has. On the other hand, it also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictionaries that would cost you days to implement efficiently in C. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.

The tutorial continues to highlight these import attributes:

1. Has a modular architecture so that code developed for one application can be reused in other programs. Likewise, it comes with a large number of built-in modules for things like file I/O, system calls, sockets, and many common Internet protocols (FTP, HTTP, SMTP, etc.),
2. It is an interpreted language conforming to the edit / test cycle discussed previously,
3. Its interpreter can be used interactively, making it easy to experiment with features of the language, or to test code before actually running a program (see fig. 11),
4. It has a high-level syntax that allows for writing compact, readable programs,
5. It has high-level data types allowing for expressions of complex data relationships,
6. It is object-oriented[10], but does not require the use of those object-oriented features, or O-O programming skills to use the language,
7. Statement grouping is done by indentation instead of begin/end brackets,
8. It is extensible: if you know how to program in C it is easy to add a new built-in function

<sup>7</sup> www.python.org



**Figure 1:** Application Domains of Python/TeX Integration.

or module to the interpreter, either to perform critical operations at maximum speed, or to link Python programs to libraries that may only be available in binary form (such as a vendor-specific graphics library),

9. It is embeddable: You can link the Python interpreter into an application written in C and use it as an extension or command language for that application.

An excellent first book for readers unfamiliar with but interested in learning Python is Mark Lutz’s *Programming Python* [4]

Finally, about the name: The tutorial enlightens us:

...the language is named after the BBC show *Monty Python’s Flying Circus* and has nothing to do with nasty reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

### Combining Python and TeX

There are a number of ways in which to combine TeX and Python to automatically produce documents. If one considers the amount of “work” necessary to produce a document as fixed, then that work can be allocated partly to TeX and partly to Python. One can then imagine a scatter diagram with X and Y axes that represent, for any possible scheme, the amount of work allocated to Python and TeX, respectively. Such a diagram is illustrated in fig. 1. The diagram shows that there are several



“application domains” defined by which component (TeX or Python) receives the most development effort, or places the most demands on computing resources. These domains allow us to classify various approaches to Python/TeX integration.

**The Imperative Approach** Imagine writing a Python script that produces a file of TeX code by executing a series of `write` statements as in fig. 2 and then runs TeX and `dvips` on that file. Here the emphasis is clearly all on the Python script and the details of how the TeX code is to be produced; we know TeX will dutifully do it’s job if it is provided good code. Applications of this nature we call *imperative*, and occupy the lower right region of fig. 1.

**Figure 2:** Imperative TeX code-writing script.

---

```
#!/usr/bin/env python
import sys
import os
f = open('MyDocument.tex', 'w')
f.write('\nopagenumbers\n')
f.write('This is my first \TeX\ document \
produced from a script.\n')
f.write('\vfil\ject\bye\n')
f.close()
os.system('tex MyDocument.tex')
os.system('dvips MyDocument')
print 'Done.'
```

---

This technique is the simplest way to integrate Python and TeX.<sup>8</sup> and is surprisingly effective. While the example in fig. 2 is trivial, the imperative technique can be used in applications where documents are assembled from a large *database* of text “snippets.” Logic in the Python script provides the “smarts” that determine what snippets to select and how to arrange them for presentation to TeX. More logic and scripts of increasing complexity push the application further to the right on the X-axis in fig. 1.

**Using m4** A slight increase in sophistication (but still remaining near the X-axis of fig. 1, is to employ the macro processor program, **m4**.<sup>9</sup> `m4[8]` is an elaborate search-and-replace engine for text. For example, given the text:

Hello, NAME, today is DATE.

---

<sup>8</sup> The other simple extreme would be to prepare an entire document by hand-editing and then have Python run TeX on that file. Quite uninteresting.

<sup>9</sup> Quoting from the `m4` manual page: “The `m4` utility is a macro processor that can be used as a front end to any language (e.g., C, ratfor, fortran, lex, yacc)...” and now, TeX!

If we present that text to `m4` as input with the following command-line:

`m4 -DNAME=Sally -DDATE='22-June-2004'`

the output from `m4` would appear as:

Hello, Sally, today is 22-June-2004.

Now we can play the same game as in the imperative approach, but with a new wrinkle: tags can be embedded in our text snippets. Once the TeX code is assembled, it is preprocessed through `m4` and *then* presented to TeX. Here are the steps:

1. Assemble TeX code from snippets of text,
2. Gather data for tag-replacement from a data source,
3. Build `m4` command line with `-Dname=value` arguments for each unique tag in the TeX file,
4. Execute the command just built and save the output,
5. Present the saved output to TeX.

### TeXmerge

We now move away from the X-axis of fig. 1.

The `m4` approach introduced an important concept: the idea of *template* files. There exist a large class of applications whose function is to produce, for lack of a better term, “form letters.”<sup>10</sup> The `m4` technique of the previous section lends itself precisely to this *merging* application: Build a `.tex` file complete with tag names, then repeat steps 2-5 above until end of data. The end result will be a stack of form letters ready to print and drop in the mail.

While `m4` is an efficient macro-replacement engine, we know of another engine that eclipses it: TeX. Consider the TeX document in fig. 3.

**Figure 3:** `form.tex`: A merge-ready TeX file.

---

```
\nopagenumbers
This is my first \TeX\ document produced
from a script.
\par
Hello, \NAME, today is \DATE.
\vfil\ject
```

---

Alone, this file will result in undefined macro references because the macros `\NAME` and `\DATE` are not defined. However, when used in conjunction with the Python script in fig. 4, it works beautifully.

---

<sup>10</sup> Every technological advance seems to bring with it a raft of nastiness. With email comes spam, with computer-aided printing comes the dreaded form letter. At least with TeXmerge, they can be beautiful form letters.

**Figure 4:** Imperative  $\text{\TeX}$  code-writing script relying on  $\text{\TeX}$ 's macro replacement facility.

---

```
#!/usr/bin/env python
import sys
import os
f = open('temp.tex', 'w')
f.write('\def\NAME{Sally}\n')
f.write('\def\DATE{22-June-2004}\n')
f.write('\input form.tex\n')
f.write('\bye\n')
f.close()
os.system('tex temp.tex')
os.system('dvips temp')
print 'Done.'
```

---

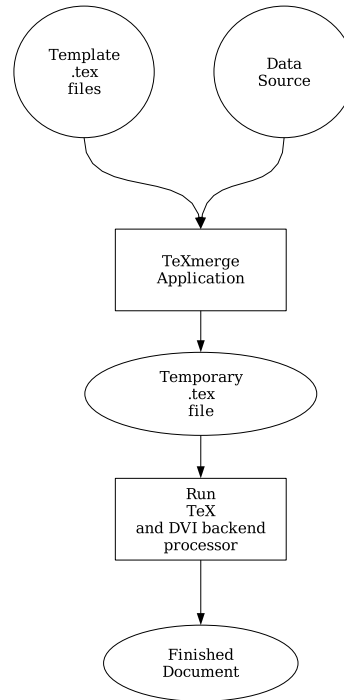
Scripts like 4 can be represented schematically as in fig. 5. It is important to note that in this scheme we are dealing with *two* (or more)  $\text{\.tex}$  files: 1) The template file(s) containing the structure of our form letter(s) (more than a single type of form letter can be produced in a single run simply by inputting different template files), and have tags where merge variables are to be inserted, and 2) the temporary file which defines macros for the merge variables and has input commands to bring in the templates. Inside the temporary  $\text{\.tex}$  file there can be many occurrences of the `def.../input...` lines; one occurrence for each letter to be produced.

**$\text{\TeX}$ merge API** The technique illustrated in fig. 4 works well. Data for the merge variables can be arbitrarily long, for example, and  $\text{\TeX}$  will 'do the right thing' and wrap the merged text into our form, etc. But there are problems:

1. The biggest problem is data containing tokens having special meaning to  $\text{\TeX}$ . If our merge data contains  $\$, \%, \&$ , etc., we have a problem,
2. It's rather tedious to read the script, and we find ourselves repeatedly re-implementing this tedious code for every application.

The whole process of opening the temporary  $\text{\.tex}$  file, protecting sensitive tokens, preparing the `\def` lines for the merge variables, doing the `\input...`, executing  $\text{\TeX}$  and the DVI backend need to be formalized inside an application programming interface (API).

We call that API " $\text{\TeX}$ merge". It was first presented[7] as a C-language API with a Python extension wrapper module. Since that time, the API has been re-written in pure Python and is now presented (see appendix A for full description of the API).

**Figure 5:** Schematic overview of document production via the  $\text{\TeX}$ merge API.

First, an example using the  $\text{\TeX}$ merge API (the  $\text{\TeX}$ merge *module*): Fig. 6 re-implements the script presented in fig. 4 using the module-level interface:

**Figure 6:** A simple Python script using the  $\text{\TeX}$ merge module-level API functions.

---

```
#!/usr/bin/env python
import sys
import os
import TeXmerge
f = TeXmerge.openOutput('temp.tex')
mergeVars = {'NAME': 'Sally',
             'DATE': '22-June-2004'}
TeXmerge.merge('form.tex', mergeVars)
TeXmerge.closeOutput(f)
TeXmerge.process('temp.tex', 'dvips')
print 'Done.'
```

---

Note the following:

1. Access to the  $\text{\TeX}$ merge module is provided via the import statement: `import TeXmerge`,
2. The native Python `open/close` calls have been replaced with calls to `TeXmerge.openOutput()` and `TeXmerge.closeOutput()`,

3. Merge variables are formally presented to the API as a Python dictionary object.
4. The `merge()` call takes care of protecting sensitive tokens in the merge data that would otherwise confuse TeX,
5. The `os.system()` calls have been replaced with `TeXmerge.process()`.

Finally, Python is an object-oriented language, so the TeXmerge module also offers a TeXmerge *class*. Fig. 7 re-implements fig. 6 using the object-oriented interface:

**Figure 7:** A simple Python script using the TeXmerge object-oriented interface.

---

```
#!/usr/bin/env python
import sys
import os
import TeXmerge
mergeObj = TeXmerge.TeXmerge('temp.tex')
mergeVars = {'NAME': 'Sally',
             'DATE': '22-June-2004'}
mergeObj.merge('form.tex', mergeVars)
mergeObj.process('dvips')
print 'Done.'
```

---

### Going Further with Macros

Now it is time to move up the Y-axis of fig. 1, focus attention on the TeX domain and investigate what benefits can be gained by writing specialized macros to enhance integration with TeXmerge.

**Do-Nothing Macros** The first class of macros to be considered are the “do-nothing” macros. These macros, from TeX’s view, evaluate to `\relax`. They exist in a TeXmerge template file to communicate information to a Python script which scans the template file. A more traditional method used to communicate information to an external entity would be to embed that information in comment strings within the file. Writing first-class macros, however, seems to produce a cleaner, readable file, and is more flexible since a do-nothing macro could, in the future, be turned into a “*do-something*” macro.

**Classic Merge Variable Declarations** Do-nothing macros were introduced in the first release of TeXmerge, with the `\texmergevar` macro. Just looking at a merge-ready template `.tex` file, it is not immediately clear what the names of all the merge variables are. `\texmergevar` allows the author of the template file to explicitly state the names of all merge variables that will be referenced in the file by coding:

`\texmergevar name`

for each merge variable. The TeXmerge module has a module-level method, `getNames`, which scans a passed `.tex` file name (and recursively any included files) and returns a list of all declared variable names. Python scripts can inspect TeX template files and determine the names of all declared merge variables.

### Extended Merge Variable Declarations

Several year’s use of the TeXmerge API as shown that document-producing applications could be made more robust if a template `.tex` file could specify precisely what *values* a merge variable should contain. The need for merge variables to take on only one value from a small set of possible values stems from the use of conditional TeX code, via the `\ifx` control sequence, etc. Conditional typesetting is powerful because it allows documents to become intelligent. *A single .tex source file can produce entirely different finished documents by testing the value of merge variable(s) and typesetting text accordingly.*

A life insurance company, for example, falls under the jurisdiction of every state in which it is licensed to conduct business. Often, a document, a “sales practice guide” say, must contain language as mandated by a particular state. Sales practice guides for forty different states may have 90% of their language in common, but each may also have unique state-specific language that none of the others contains. Having a single, intelligent source file, `salesPracticeGuide.tex`, lowers the cost of change management substantially; changes made to shared text need only be made *once*.

The do-nothing macro `\texmergevardef` defines merge variables with extended attributes like this:

`\texmergevardef [attrName=attrValue,...]`

Attributes of the merge variables that can be specified are:

- **name=** the name of the merge field,
- **type=** the type of merge field. The intended use of this attribute is to convey a recommended style of data entry element for graphical (GUI) applications. Valid types are:
  - **entry:** a simple text entry field,
  - **text:** a multi-line text entry field,
  - **toggle:** a toggle button field,
  - **optionmenu:** a drop-down option menu of choices,
  - **radiobutton:** a set of mutually-exclusive toggle buttons,

- **values**= a list of valid values for the variable, separated by |'s
- **labels**= a list of alternate labels that should be associated with the **values** attribute for display purposes. Used with the **toggle**, **optionmenu**, and **radiobutton** field types.
- **descr**= a description of the merge variable.

The `TeXmerge` module-level function, `getExtendedNames`, extracts extended merge variable definitions and returns them in a dictionary (keyed by the **name** attribute's value) of field attribute dictionaries.<sup>11</sup> Fig. 8 shows an example `.tex` file with extended merge variable definitions. Fig. 9 shows the return value from applying `getExtendedNames` on that file.

**Named Text Blocks** Another class of applications have the need to share identical text between two markup languages: `TeX` and `HTML`. Here it is *language elements within the document* that need to be identical (for legal reasons, say) and not the structure of the document that is constant between the two presentation platforms. Indeed, structure of the printed `TeX` document may be substantially more complex than its briefer, light-weight, `HTML` cousin. How can the common text be shared between the markup languages?

One way is to make the `TeX` document “own” the text. It declares, via a set of macros, where the common blocks of text begin and end. We refer to these blocks as *named text blocks*. The demarcation macros look like this:

- `\StartNamedTextBlock[attName=value,..]`  
Text block attributes are as follows:
  - **name**= Name of the text block,
  - **seq**=*Integer* Several sections of text can be assigned the same name, but with unique sequence numbers. The extracted text will be a concatenation of like-named blocks, order by sequence number,
  - **subkey**=*subvalue*: See the text for full discussion.
- `\StopNamedTextBlock`

Once text boundaries have been marked and named with these macros, the text can be extracted and used by the `HTML` producing part of the application. The `TeXmerge` module provides a module-level function, `getNamedTextBlocks`, to extract the named text blocks, and two helper classes `TextBlock` and `TextBlockManager` to make accessing the extracted blocks simpler.

<sup>11</sup> `getExtendedNames()` also detects occurrences of `texmergevar` macros and treats them as extended merge fields having an attribute `type=entry`.

We explain the functional use of named text blocks by way of the example file in fig. 10 and the interactive Python interpreter session shown in fig. 11.<sup>12</sup>

Note the following:

1. The block demarcation macros are essentially invisible to `TeX`, and have no effect on typesetting,
2. `TextBlockManager` class is used to extract the named blocks. One simply passes a pathname to the `.tex` file containing named text blocks in order to instantiate a `TextBlockManager` object,
3. Names of all the text blocks in the file are retrieved by calling the manager object's `getNamedTextBlocks` method,
4. Individually named text blocks are retrieved via the manager object's `getTextBlock` method, or simply by indexing the manager using the name of a text block as the index key (as was done for block C1 in fig. 11. Either operation will return a `TextBlock` object.
5. Access to the text of a `TextBlock` object is via its `getText` method.

## Do-Something Macros

### Hybrid Script-`TeX`-Script Scheme:

**A Case Study** If we have an application where a substantial amount of the document's content may vary, the merge paradigm of `TeXmerge` begins to break down under the complexity of so many variables. This is especially true of variable tabular data.

Example: The annotated page shown in fig. 12 is a rate sheet of life insurance premiums. As the figure shows, there is more variable data than static text on the page. The rate sheet, however, is only one page of a twenty page document. Other pages in its parent document also have variable data, and state-specific language, as well. Overall the document's nature fits well in the `TeXmerge` scheme; the rate sheet page is the “trouble maker.” Another important consideration: the rate sheet needs to be embeddable in many other documents.

One desires a `TeX` macro as in fig. 13 that, when executed, magically produces a finished rate sheet.<sup>13</sup>

<sup>12</sup> About the interactive interpreter session: `>>>` is the interpreter's prompt. Text appearing after that prompt was entered by the user. Python's response appears on the line immediately below the prompt input line.

<sup>13</sup> Writing parameter based macros such as these is effortless with the aid of support macros found in Hans Hagen's `ConTeXt` macro package.

Figure 8: A sampling of extended merge variable declarations.

```
\texmergevardef[name=ISTATE, type=optionmenu, values=TX|OK|AZ|CA|OR|WA, descr=Issuing state]
\texmergevardef[name=ONAME, type=entry, descr=Owner name']
\texmergevardef[name=APPTYPE, type=radiobutton, values=1|2|3, labels=Employee|Spouse|Child,
  descr=Applicant type]
```

Figure 9: Result of getExtendedNames(): a Python dictionary of field-attribute dictionaries

```
{'ISTATE': {'name': 'ISTATE', 'type': 'optionmenu', 'values': ('TX', 'OK', 'AZ', 'CA', 'OR', 'WA'),
'descr': 'Issuing state'}, 'APPTYPE': {'name': 'APPTYPE', 'type': 'radiobutton', 'values':
('1', '2', '3'), 'labels': ('Employee', 'Spouse', 'Child'), 'descr': 'Applicant type'}, 'ONAME': {
'name': 'ONAME', 'type': 'entry', 'descr': 'Owner name'}}
```

Figure 13: Rate sheet macro.

```
\MakeRateSheet[uwclass=express,
  mode=semi-monthly,
  groupsize=150,
  formno=test,
  waiver=yes,
  adb=yes
]
```

\MakeRateSheet[...] is definitely a *do-something* macro. The trick is to do as little work as possible in TeX and most of the *something* in a Python script. The work for TeX in this case is in two parts:

1. Gather macro arguments and marshal them into a Python script command-line, then execute the command with \write18,
2. Input and typeset the TeX code produced by the Python script.

We call schemes such as these *hybrid* or *Script-TeX-Script* schemes. The job of the secondary script (the one executed by TeX via \write18) is to act on arguments received from TeX, or from some other external source, do whatever calculations, etc. and output TeX-code. The whole scheme is represented in fig. 14. Since the secondary script is unbounded by the complexity and amount of TeX code that may be returned, hybrid schemes are the ultimate in flexibility.

**Document Template Macros** Document template macros fall into the class of *do-something* macros. Another case study will serve as a description of their functionality. TeXmerge is in widespread use at Texas Life having applications in almost every major department, from Marketing, to New Business, to Policy Owner Service, to Computing Services. Several years ago, a graphic artist

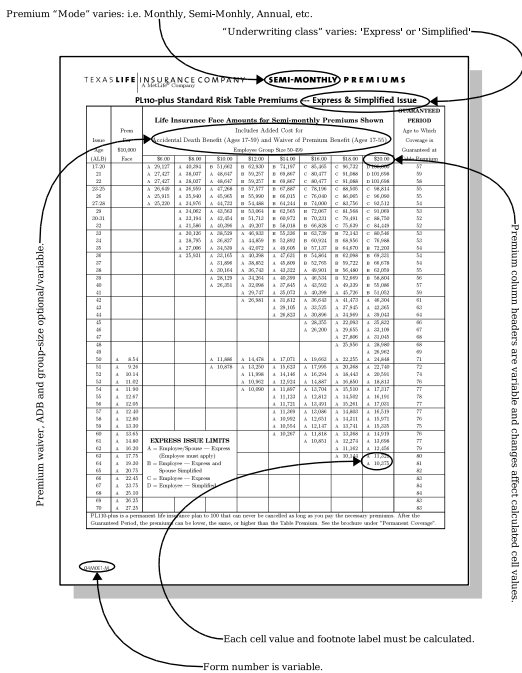


Figure 12: Complex document produced by Hybrid Script-TeX-Script scheme.

**Figure 10:** T<sub>E</sub>X file `test.tex` containing four named text blocks, B1, B2, C1, D1.

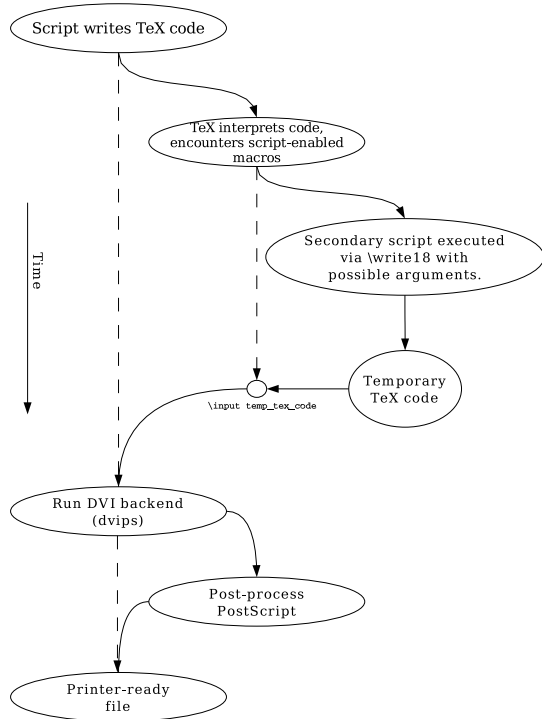
---

```

This is a test document containing \textit{named text blocks.}
\StartNamedTextBlock[name=B1]
This is the first block.
\StopNamedTextBlock
Now for a second block:
\StartNamedTextBlock[name=B2]
Second block
\StopNamedTextBlock
Now for a series of sequenced blocks...
\line{\hbox{\StartNamedTextBlock[name=C1,seq=1]C1.Left\StopNamedTextBlock\hfil}
      \hbox{\hfil\StartNamedTextBlock[name=C1,seq=2]C1.Right\StopNamedTextBlock}}
}
Finally, a named text block having a subkey:
\StartNamedTextBlock[name=D1,istate=TX]
This text is specific to the state of Texas.
\StopNamedTextBlock

```

---

**Figure 14:** Schematic overview of document production via the hybrid technique.

was hired to develop a new ‘look-and-feel’ for all printed material disseminated from the company. A new graphics standards manual was written and all parts of the company were informed that compliance with the new standard was mandatory by a set date. This directly affected users of T<sub>E</sub>Xmerge. The Policy Owner Service department, for example, had 600+ T<sub>E</sub>Xmerge-based form letters used daily for corresponding with clients. Compounding the problem were the non-standard fonts and a peculiar format to which standard letterhead should conform: a wide left margin, except for various items that were to remain left hanging, right-justified. How could over 600 documents be quickly converted to this new format? Language inside the documents could remain unaltered; only the structure was changing.

Serendipitous earlier decisions, made when originally planning and setting up the T<sub>E</sub>Xmerge letters made conversion to the new graphics standard straightforward. The serendipity was in a decision to separate the text for the body of each letter into its own `.tex` file. That being the case, all that was needed was a mechanism to enforce policy of the graphics standard; a way to automatically produce the required layout of the document. This we do with so-called *template* macros. Fig. 15 shows the structure enforced by the `\StartClientLetter` macro. Based on a *plug-and-socket* model, it relies heavily on macro parameters (almost all having default values), as can be seen in the figure. Template macros classify parameters into three categories:

- Simple parameters - parameter names beginning with `mp`,
- Data sockets - parameter names beginning with `sd`,

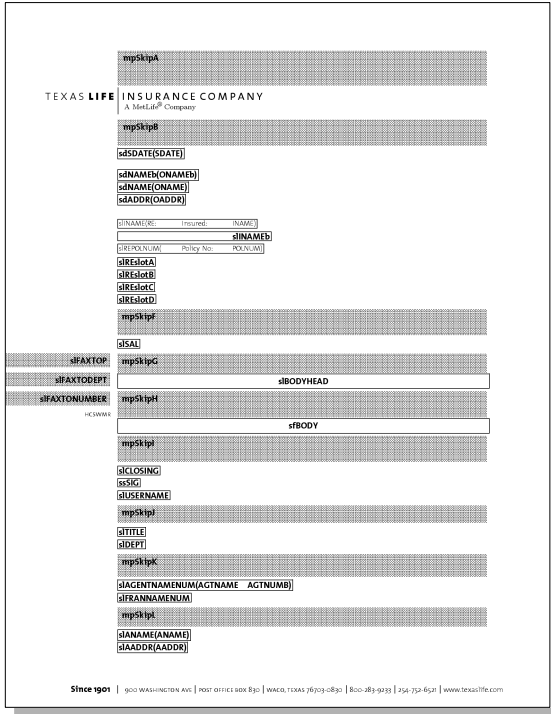


Figure 15: Template view for the client-letter macro.

- Slots - parameter names beginning with `sl`<sup>14</sup>.

The `mpSkip...` parameters (gray strips shown in fig. 15) can be specified to alter whitespace. Merge variable data is connected to a template using a *plug-and-socket* model. Merge variable names are termed *plugs* and the `sd...` macro parameters are termed *sockets*. One plugs a variable to particular position on the letter by equating the name of the plug with the desired socket name. The socket names are shown on the template letter in fig. 15 with default plug values in parenthesis. Finally, *slots* are macro parameters that can accept arbitrary TeX code as arguments.

The body of the letter can be supplied to the template macro in one of two ways:

1. Put the text of the body into a separate `.tex` file and pass the name of the file in the `sfBODY` parameter,
2. Code text of the body immediately after invoking the `\StartClientLetter`. In this case the letter must be finished with the `\FinishClientLetter` macro.

<sup>14</sup> There are two other prefixes: `ss` - related to insertion of digitized versions of handwriting signatures, and `sf` - related to input files.

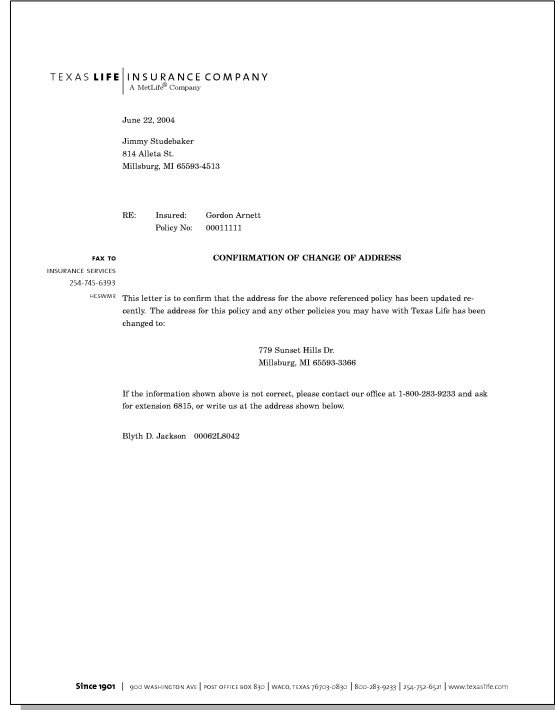


Figure 16: Sample letter produced using the client-letter macro.

Finally, a sample letter produced from the `\StartClientLetter` macro is shown in fig. 16.

### Building GUI Applications with TeXmerge

So far, discussions of TeXmerge have tended more to batch-style applications. The API is also effective in building GUI applications. The module's `getNames` and `getExtendedNames` functions provide useful *metadata* about merge fields, which can be used to construct user interfaces. Python is equally effective in programming GUI interfaces. The “Gimp Toolkit”<sup>15</sup> is especially easy to access from Python and provides a robust set of GUI interface components, including Pixmap buffers which, along with GhostScript<sup>16</sup>, can be used to effectively render PostScript.

**TeXmerge - the Application** The TeXmerge API was originally developed for use in an interactive application, also called TeXmerge, for production of form letters. Originally written in C and based on the Motif toolkit, the current version is written in pure Python and is based on GTK+-2.4.

<sup>15</sup> [www.gtk.org](http://www.gtk.org) and [www.pygtk.org](http://www.pygtk.org).

<sup>16</sup> [www.ghostscript.com](http://www.ghostscript.com)

The application is arranged around *categories* of correspondence (collections of form letters, grouped by activity). Each activity category's letters are stored in a category subdirectory.

A sample  $\TeX$ merge main application window is shown in fig. 17. A category frame consists of the document selection window on the left, and a set of merge variable data entry fields on the right. A single set of input fields (a *record*), generates a single copy of the associated letter. Control buttons exist along the bottom to accomplish tasks such as adding new records, removing records, printing, and saving. A built in PostScript viewer (not visible) is also provided to view the letter before printing or saving.

**$\TeX$ tool** As long as we're writing GUI applications, why not write one that aids in the development of  $\TeX$ merge documents?  $\TeX$ tool is an integrated development utility for editing, " $\TeX$ 'ing," and viewing  $\TeX$ merge documents. Figs. 18, 19, and 20 are three successive views of the application, each view showing one of the major notebook tab pages revealed: Document, Editor, and Preferences. Applications of this style exist that are more effective, in general, however,  $\TeX$ tool is unique because it is oriented especially for  $\TeX$ merge documents. It also shows the feasibility of integrating  $\TeX$  into a non-trivial GUI application written in a scripting language. As can be gleaned from the figures, the Document tab displays the input frame of  $\TeX$ merge variables as they will appear in the normal  $\TeX$ merge application. The edit/test cycle can be quickly done all inside a single application window.

### The Big Picture at Texas Life

As mentioned earlier on in the the case studies,  $\TeX$ merge is in widespread use at Texas Life. Fig. 21 is reproduced from [7]. It is a convincing illustration of how effective  $\TeX$  can be as a document production engine, especially if combined with the right scripting language (Python). Most of the ovals in the figure use  $\TeX$ merge in some fashion. An important lesson learned is that once a facility like  $\TeX$ merge is available, the movement of documents between systems becomes much simpler. Only data required to build documents need be communicated along the arrows in the figure. Documents are only built and rendered when necessary for viewing or printing.

### Conclusion

Because  $\TeX$  is an ASCII text markup language, it is effective to write computer codes to process the  $\TeX$  code for purposes other than typesetting. Scripting languages simplify writing these extraction codes. Embedding metadata into  $\TeX$  files via simple macros allows the  $\TeX$  author to communicate information to other computer applications. And, finally, using  $\TeX$  alongside scripting languages in an automated document production environment provides flexibility and robustness to meet almost any demand imaginable. "Hacking" with scripting languages has never been simpler. Now is the time for more people to become *script literate*; the author encourages those with little or no programming experience to mix up a scripting language with their favorite  $\TeX$  macro package.

### References

- [1] Bruce Eckel. Strong typing vs. strong testing. <http://www.mindview.net/WebLog/log-0025>, 2003.
- [2] Paul Graham. Hackers and painters. <http://www.paulgraham.com/hp.html>, 2004.
- [3] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, third edition, 1997.
- [4] Mark Lutz. *Python Programming*. O'Reilly and Associates, Inc., first edition, 1996.
- [5] Eric S. Raymond. The art of unix programming. <http://www.faqs.org/doc/artu/ch14s01.html>, 2003.
- [6] Eric S. Raymond. The meaning of 'hack'. <http://www.catb.org/esr/jargon/html/meaning-of-hack.html>, 2003.
- [7] William M. Richter. Integrating  $\TeX$  into a document imaging system. *TUGBoat*, 22(3), 2001.
- [8] René Seindal. Gnu m4 - development site. <http://www.seindal.dk/rene/gnu>, 2003.
- [9] Unknown. Technical definition of scripting language. <http://c2.com/cgi/wiki?ScriptingLanguage>, 2003.
- [10] Webopedia. What is object oriented programming? <http://webopedia.com/TERM/O/object-oriented-programming-OOP.html>, 2003.



---

**Figure 11:** Interactive Python interpreter session. Working with named text blocks.

---

```
[hawkeye2:~/sftug] williamr% python
Python 2.3.2 (#1, Nov 6 2003, 13:18:07)
[GCC 2.95.2 19991024 (release)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import TeXmerge
>>> o = TeXmerge.TextBlockManager('test.tex')
>>> o
<TeXmerge.TextBlockManager instance at 0x750648>
>>> o.getBlockNames()
['C1', 'B1', 'B2', 'D1']
>>> b1 = o.getBlock('B1')
>>> b1
<TeXmerge.TextBlock instance at 0x72b5d0>
>>> b1.getText()
'This is the first block.'
>>> c1 = o['C1']
>>> c1.getTextSegments()
{1: 'C1.Left', 2: 'C1.Right'}
>>> c1.getText()
'C1.Left C1.Right'
>>> d1 = o['D1']
>>> d1.getSubkeys()
['istate']
>>> d1.getSubkeyValues('istate')
['TX']
>>> d1.getText('istate','TX')
'This text is specific to the state of Texas.'
```

---

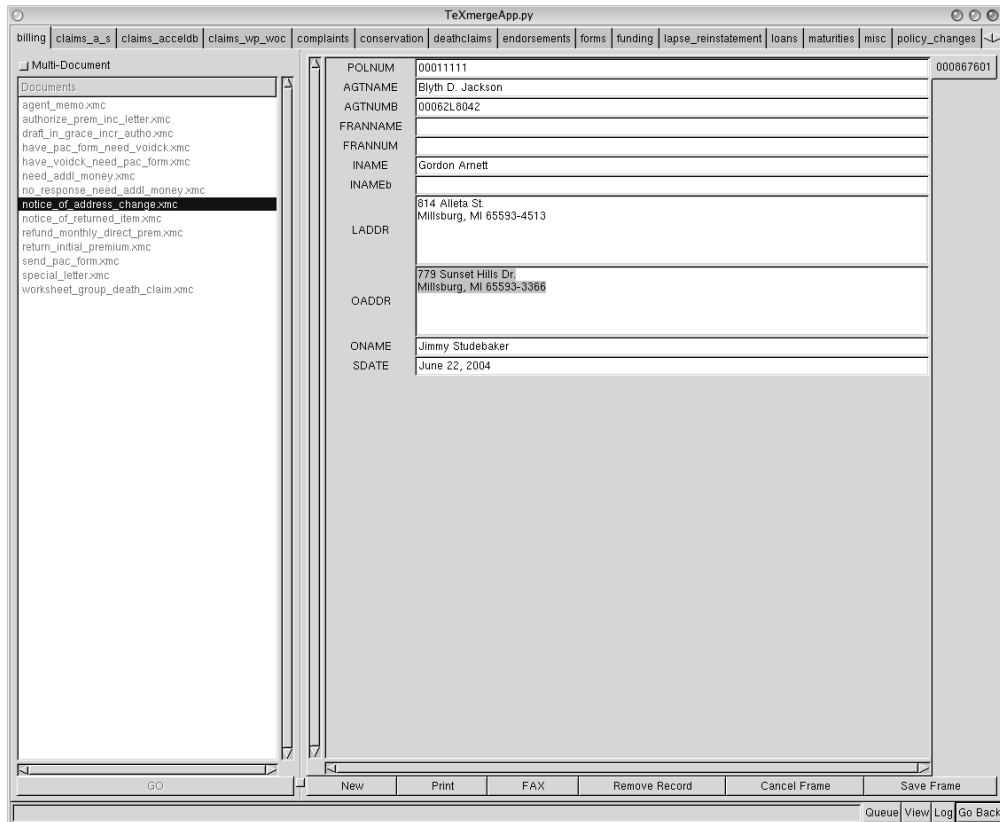


Figure 17: The TeXmerge application main window.

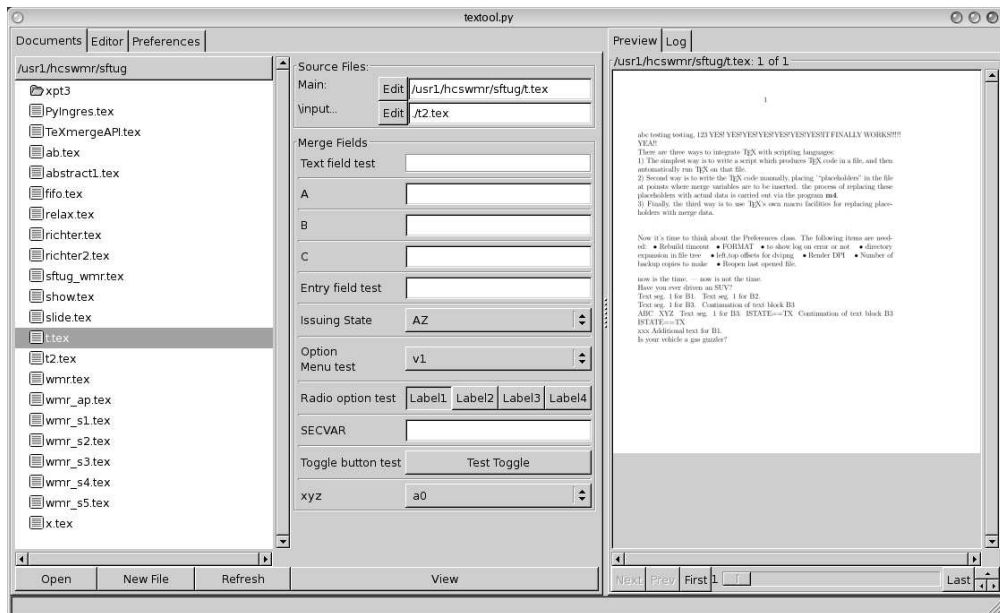


Figure 18: The textool app. with the Documents tab visible.

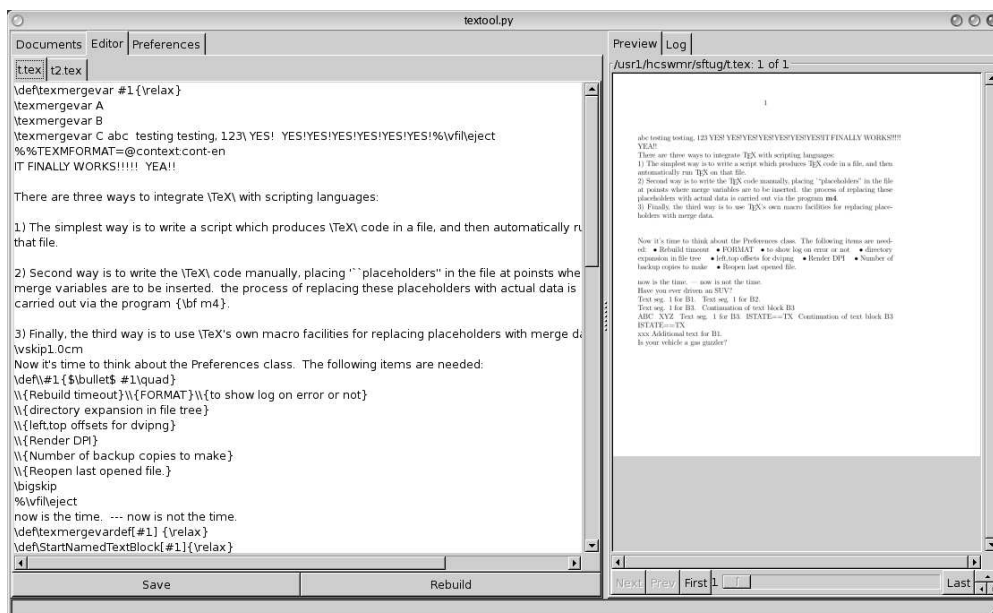


Figure 19: The textool app. with the Editor tab visible.

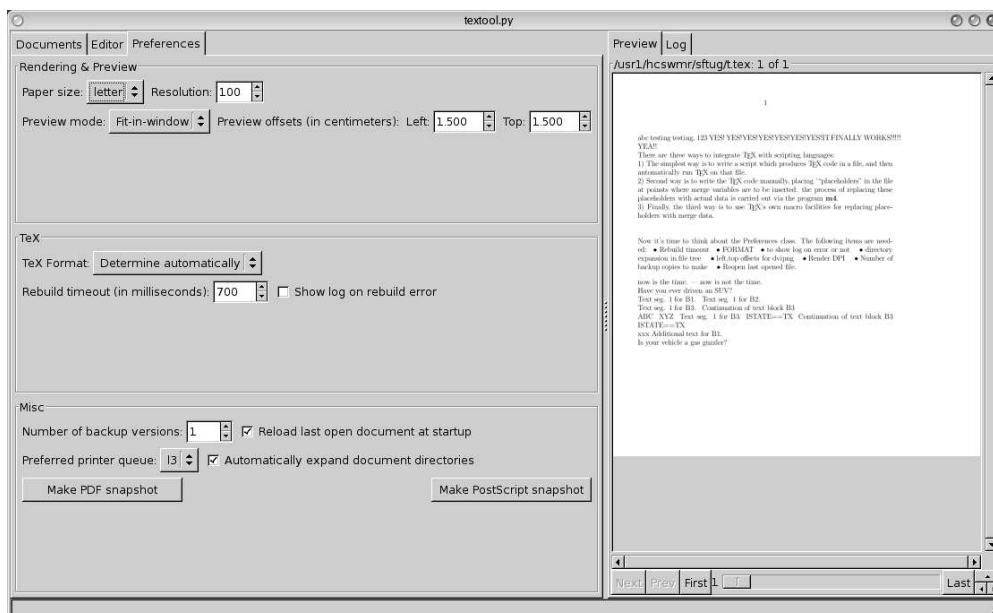


Figure 20: The textool app. with the Preferences tab visible.

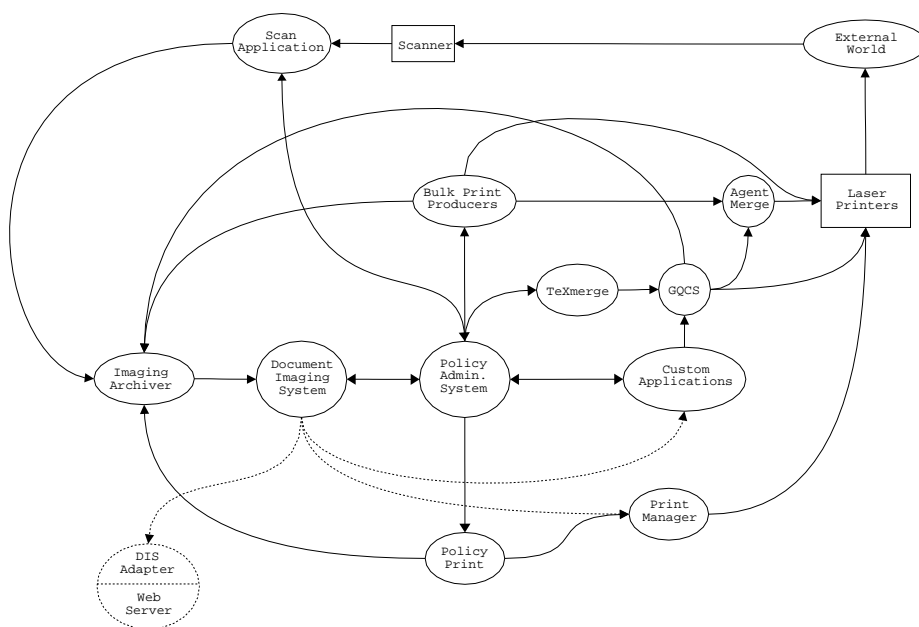


Figure 21: The big picture of TeXmerge at Texas Life.

## Appendix A Python TeXmerge API

### A Note About How TeXmerge Runs TeX:

Because there are a significant number of macro packages available as TeX formats, TeXmerge needs to be adaptable, both to what format to use, and also to the way in which the TeX interpreter is started. To allow for this flexibility, many of the functions below take two arguments, `format` and `strategy`. `format` specifies what TeX format to use and `strategy` specifies the way in which TeX will be started. In many cases, these arguments are optional and appropriate values will be derived, either from the context of use or from the environment variable, "TEXMFFORMAT." The environment variable has two different forms:

1. TEXMFFORMAT=*format*
2. TEXMFFORMAT=@*strategy*:*format*

The second form allows for specification of both the strategy and format. Currently `strategy` can be set to one of: `context`, `latex`<sup>17</sup>, or `plain`. The table below maps strategies to command-lines:

strategy	command-line
context	texexec -format <i>format</i> -once %s
latex	latex %s
plain	tex & <i>format</i> %s

### Module-level Functions

- `getNames(pathname)` → [*name*<sub>1</sub>, *name*<sub>2</sub>,...]
 

Recursively scans the passed *pathname* and returns a list of merge variable names declared by instances of the `\texmergevar` macro.
- `getExtendedNames(pathname)` → {attrDict<sub>1</sub>, attrDict<sub>2</sub>,...}
 

Recursively scans the passed *pathname* and returns dictionary of merge variable field attribute dictionaries. The merge field attribute dictionaries are created from instances of the `\texmergevardef` macro which defines merge variables with extended attributes like this:

`\texmergevardef [attrName=attrValue,...]`

Attributes of the merge variables that can be specified are:

- `name`= the name of the merge field,
- `type`= the type of merge field. The intended use of this attribute is to convey a recommended style of data entry element for graphical (GUI) applications. Valid types are:
  - \* `entry`: a simple text entry field,
  - \* `text`: a multi-line text entry field,
  - \* `toggle`: a toggle button field,
  - \* `optionmenu`: a drop-down option menu of choices,
  - \* `radiobutton`: a set of mutually-exclusive toggle buttons
- `values`= a list of valid values for the variable, separated by |'s
- `labels`= a list of alternate labels that should be associated with the `values` attribute for display purposes. Used with the `toggle`, `optionmenu`, and `radiobutton` field types.
- `descr`= a description of the merge variable.
- `hashNames(fieldAttributesDict)` → *StringObject* containing hex representation of MD5 hash
 

Computes a 64-bit MD5 hash of passed field attributes dictionary and returns it as a string object of hexadecimal characters.
- `getInputFiles(pathname)` → [*pathname*<sub>1</sub>, *pathname*<sub>2</sub>,...]
 

Recursively scans the passed *pathname* for occurrences of `\input` control sequences and returns a list of pathnames.
- `openOutput(pathnameOrFileObject, preambleCode=None, formatIn=None, strategyIn=None)`

→ *FileObject*

---

<sup>17</sup> For the latex strategy, format=latex is always assumed.

Prepares a temporary work file for merge operations. The first argument can be either a string object or a file object. In the case of a string object, it is interpreted as the pathname to a file where the temporary merge file should be created. If it exists, it will be removed and re-created. In the case of a file object, the argument is assumed to be a previously opened file. Any write operations issued by `TeXmerge` will be executed against the passed file object. `preambleCode`, if specified will be written at the beginning of the file in place of `TeXmerge`'s normal preamble code. `formatIn` is currently unused. `strategyIn` determines the default form of preamble code to write. Valid values are `context`, `latex`, or `plain`.

- `closeOutput(fileObject, postambleCode=None, formatIn=None, strategyIn=None, keepOpen=False)`

Completes preparation of a temporary work merge file for processing. `postambleCode` is written to the file if passed, otherwise an appropriate postamble will be supplied depending on the values of `formatIn` and `strategyIn`, if passed, or a default postamble will be written. The passed `fileObject` will be closed unless `keepOpen` is passed as `True`.

- `merge(targetPathname, mergeVariableDict, fileObject, options=0) → None`

Encapsulates the merge variables passed in `mergeVariableDict` for use in `targetPathname`. The merge variables are written to the merge work file as `\def` control sequences, and `targetPathname` is referenced via an `\input targetPathname`.

Several merge options can be passed in the `options` argument:

1. `TXM_FRAMEVARS` - draw boxes around all merged variables,
2. `TXM_DUPLEX` - assume the output will be printed on a duplexing device and insert `\eject` macros between merge invocations, when appropriate, to ensure that each merge invocation starts on the front side of the printed sheet.

- `process(pathname, driverCommand, format, strategy) → Integer Object`

Runs the `TeX` interpreter and a DVI backend against the merge work file `pathname`. The command used to run the `TeX` interpreter is derived from the `format` and `strategy` parameters. `Strategy` may be one of `context`, `latex`, or `plain`. If `strategy` is set to `context` then the environment variable `TEXENGINE` is used as the `TeX` processor, if set, or `texexec` otherwise. The DVI command string passed in `driverCommand` is used to run the DVI backend. It can contain a single `"%s"` which will be replaced with `pathname`. If no `"%s"` is present, `pathname` will be appended to `driverCommand`.

Returns the exit status of `TeX` interpreter or of the DVI backend command.

- `processWithExtendedOutput(pathname, driverCommand, format, strategy) → (texstderr, texstdout, texlog, dvistderr, dvistdout)`

Works identically as with `process` above, except for error handling. Failure of the `TeX` interpreter raises the exception, `TeXException`. Failure of the DVI backend command raises the exception `DviException`. Successful completion of both the `TeX` interpreter and the DVI backend returns a tuple as above, providing complete diagnostics of the run.

- `getNamedTextBlocks(pathname) → {block1:{block1AttrDict}, ...}`

Recursively scans `pathname` for occurrences of *named text blocks* as demarked by the pair of macros `\StartNamedTextBlock[attName=value, ..]` and `\EndNamedTextBlock`.

Text block attributes are as follows:

- `name`= Name of the text block,
- `seq`=*Integer* Several sections of text can be assigned the same name, but with unique sequence numbers. The extracted text will be a concatenation of like-named blocks, ordered by sequence number,
- `subkey`=*subvalue* Subkey name/value pairs provide a way to declare multiple blocks with the same name. Assigning differing name/value pairs makes each like-named block unique.

The class `TextBlockManager` can be used as an alternative to this function to provide a simple frontend to this function's return value.

**TeXmerge Class** The T<sub>E</sub>Xmerge class provides an object-oriented interface to the module level functions shown above.

Constructor: `TeXmerge(mergeTargetPathname=None, workPathname=None, mergeOptions=0, preambleCode=None, postambleCode=None, texmformat=None, strategy=None, keepIntermediateFiles=False)`

**Methods:**

- `setMergeTargetPathname(pathname) → None`  
Sets the default merge target pathname that will be used for subsequent merge operations.
- `setMergeOptions(self, mergeOptions) → None`  
Sets the default merge options that will be used for future merge operations.
- `setFormatAndStrategy(self, texmformat, strategy=None) → None`  
Sets the default format and strategy to be used for future merge operations.
- `probeMergeTargetAndSetFormat() → None`  
Scans the current merge target pathname and determines the appropriate format and strategy that should be used during the `process()` method call.
- `setFormatFromMergeTargetParentDirectory() → None`  
Checks the merge target's parent directory for existence of the file `.texmformat`. If found, the contents of the file is assumed to be the format and strategy (similar in format to the environment variable `TEXMFFORMAT`) to be used when processing the merge file.
- `getVariables() → {mergeVariableAttrDict}`  
Calls the module-level function `getExtendedNames`, passing the currently set merge target pathname as an argument. Returns the result of the call.
- `openOutput(workPathnameOrFileObject=None) → FileObject`  
Prepares the work file for subsequent merge operations. If no argument is passed a default filename will be constructed.
- `closeOutput() → None`
- `merge(mergeVars=None, altMergeOptions=None, altMergeTargetPathname=None) → None`  
Performs a merge operation using `mergeVars`, if passed, and alternate merge options and merge target pathname, also, if passed.
- `process(dviCommandString) → None`  
Run T<sub>E</sub>X interpreter according to currently set strategy and format. The DVI command string passed in `dviCommandString` is used to run the DVI backend. It can contain a single “%s” which will be replaced with the current value of the work file's pathname. If no “%s” is present, the current work file's pathname will be appended to `dviCommandString`.  
Failure of the T<sub>E</sub>X interpreter raises the exception, `TeXException`. Failure of the DVI backend command raises the exception `DviException`.

**TextBlock Manager Class**

Constructor: `TextBlockManager(pathname)`

**Methods:**

- `setPathname(pathname) → None`  
Requests the `TextBlockManager` instance to scan `pathname` for named text blocks. Any information about previously scanned blocks is lost.
- `getBlockNames() → [block1, block2, ...]` Returns a list of the names of all named text blocks in the pathname last scanned.
- `getBlock(blockName) → TextBlock Instance` Returns a `TextBlock` instance representation of the text block named `blockName`. Returns `None` if no such named block exists.  
This same operation can be performed by using array indexing notation against the instance. i.e. index it like a dictionary object.

### TextBlock Class

Constructor: `TextBlock(text-block-descriptor-dictionary)`

#### Methods:

- `getName()` → *blockName string object*  
Returns the instance's block name.
- `getSubKeys()` → [*blockName*<sub>1</sub>, ...] | `None`  
Returns a list of unique subkey names associated with the text block or `None` if there are no associated subkeys.
- `getSubkeyValues(subkeyName)` → [*subkeyName*<sub>1</sub>, ...]  
Returns a list of all the subkey values corresponding to the passed subkey name.
- `getTextSegments(subkeyname=None, subkeyValue=None)` → {1: *textSeg*<sub>1</sub>, 2: *textSeg*<sub>2</sub>, ...}  
Returns a dictionary of text segments, keyed by segment sequence number. *SubkeyName* and *SubkeyValue* are optional, and if specified, are used to select the specific text block to access.
- `getText(subkeyname=None, subkeyValue=None)` → *StringObject*  
Returns a concatenation of all text segments in order by sequence number. *SubkeyName* and *SubkeyValue* are optional, and if specified are used to select the specific text block to access.

**Exceptions** Several exceptions can be raised by some of the class methods above. The exception objects have attributes which provide diagnostics about the associated error condition.

**TeXException** This exception is raised when T<sub>E</sub>X cannot successfully interpret a file.

#### Attributes:

- `stdout`: `StringObject` containing the stdout stream from the interpreter invocation,
- `stderr`: `StringObject` containing the stderr stream from the interpreter invocation,
- `logText`: `StringObject` containing T<sub>E</sub>X's logfile output.

**DviException** This exception is raised when a DVI backend driver fails.

#### Attributes:

- `stdout`: `StringObject` containing the stdout stream from backend invocation,
- `stderr`: `StringObject` containing the stderr stream from backend invocation,



## TUG 2005 Announcement and Call for Papers

TUG 2005 will be held in Wuhan, China, from August 23–25, 2005. CTUG (Chinese T<sub>E</sub>X User Group) has committed to undertake the conference affairs, and now announces the call for papers.

*Why go to China for TUG 2005?*

### For fun!

This is the first TUG conference to be held in China. Wuhan is close to the birthplace of Taoism and the Three Gorges Reservoir. China is also the birthplace of typography in ancient times, and is simply a very interesting place to go.

### For keeping up with the community!

The T<sub>E</sub>X community in China has been growing over the years. China is one of the few countries in the world which has heavily applied free software (including T<sub>E</sub>X, GNU/Linux, and more) in industry. The rich human resources and the creative T<sub>E</sub>X hackers have become a part of the engine driving the global T<sub>E</sub>X community. TUG'05 is a good opportunity to meet them.

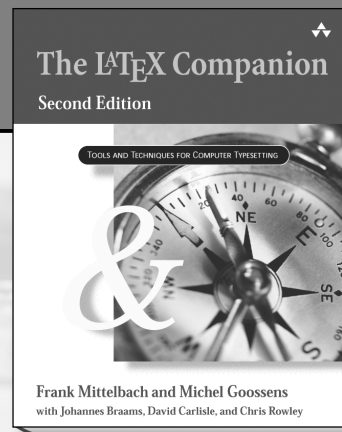
### For your future!

The growing market is ready to use your expertise. Many libraries, publishing houses, and scientific organizations in China are eager to use your T<sub>E</sub>X expertise.

Please submit abstracts for papers to [tug2005@tug.org](mailto:tug2005@tug.org).  
For more information about TUG 2005, please visit:  
<http://tug.org/tug2005>

# The L<sup>A</sup>T<sub>E</sub>X Companion

## Second Edition



ISBN: 0-201-36299-6

**Frank Mittelbach and Michel Goossens**  
*with Johannes Braams,  
David Carlisle, and Chris Rowley*

*The L<sup>A</sup>T<sub>E</sub>X Companion* has long been the essential resource for anyone using L<sup>A</sup>T<sub>E</sub>X to create high-quality printed documents. This completely updated edition brings you all the latest information about L<sup>A</sup>T<sub>E</sub>X and the vast range of add-on packages now available—over 200 are covered. Like its predecessor, *The L<sup>A</sup>T<sub>E</sub>X Companion, Second Edition* is an indispensable reference for anyone wishing to use L<sup>A</sup>T<sub>E</sub>X productively.

*Available at fine bookstores everywhere.*

  
Addison  
Wesley

For more information, visit:  
[www.awprofessional.com/  
titles/0201362996](http://www.awprofessional.com/titles/0201362996)

# A better way

Here at River Valley Technologies we work with clients such as Elsevier and the IOP, dramatically improving the way they produce their mathematical publications. Our culture of innovation has created a completely automated workflow from LaTeX to MathML and back, removing the need for human intervention in the conversion process.

For heavy mathematical typesetting, ours is the most effective, proven system available anywhere in the world. Learn more about it from Dr. Kaveh Bazargan by emailing [kaveh@river-valley.com](mailto:kaveh@river-valley.com).



**RIVER VALLEY**  
TECHNOLOGIES

## Scientific WorkPlace® Scientific Word®

Mathematical Word Processing • L<sup>A</sup>T<sub>E</sub>X Typesetting • Computer Algebra

### Version 5 Sharing Your Work Is Easier

- ◆ Typeset PDF in the only software that allows you to transform L<sup>A</sup>T<sub>E</sub>X files to PDF, fully hyperlinked and with embedded graphics in over 50 formats
- ◆ Export documents as RTF with editable mathematics (Microsoft Word and MathType compatible)
- ◆ Share documents on the web as HTML with mathematics as MathML or graphics

#### The Gold Standard for Mathematical Publishing

*Scientific WorkPlace* makes writing, sharing, and doing mathematics easier. A click of a button allows you to typeset your documents in L<sup>A</sup>T<sub>E</sub>X. And, you can compute and plot solutions with the integrated computer algebra engine, *MuPAD*® 2.5.



Email: [info@mackichan.com](mailto:info@mackichan.com) • Toll-free: 877-724-9673 • Phone: 360-394-6033  
Visit our website for free trial versions of all our software.

[www.mackichan.com/tug](http://www.mackichan.com/tug)

Version 5  
with  
PDF L<sup>A</sup>T<sub>E</sub>X

In the Monte Carlo simulations that follow, three bandwidth choices are parameter combination: The LSCV bandwidth, the "Stanton" bandwidth, an independent and identically distributed (IID) bandwidth. The first choice is the least squares cross validation problem (ref: LSCVfunc). The IID bandwidth for IID data, and it is defined as  $h^{iid} = \hat{\sigma} T^{-1/5}$ , where  $\hat{\sigma}$  is the sample standard deviation and  $T$  is the sample size. The Stanton bandwidth is the one actually used in Stanton (1997) are based directly on equations (ref: Stanton1997). In particular, "inverting" these equations yields:

$$\mu(x_i) = \frac{1}{\Delta} E[x_{i+\Delta} - x_i \mid x_i] + \frac{o(\Delta)}{\Delta}$$
$$\sigma(x_i) = \sqrt{E[(x_{i+\Delta} - x_i)^2 \mid x_i] \frac{1}{\Delta} + \frac{o(\Delta)}{\Delta}}$$

The essence of Stanton's approach is to apply the Nadaraya-Watson (Nadaraya-Watson) regression estimator to construct nonparametric estimates of the conditional moments (ref: diff2) and (ref: diff2):

$$\hat{\mu}(x_i) = \frac{1}{T-1} \sum_{j=1}^{T-1} (x_{i+\Delta}^j - x_i^j) K\left(\frac{x_i - x_i^j}{h}\right)$$

Screen text is reprinted from an article in the Journal of Finance.



# PCT<sub>E</sub>X v5 Makes Using L<sup>A</sup>T<sub>E</sub>X a Snap!

PCT<sub>E</sub>X v5 for Windows has new, enhanced features in an integrated user interface to compose L<sup>A</sup>T<sub>E</sub>X and T<sub>E</sub>X documents:

- ▼ Smooth edit–typeset–preview workflow
- ▼ Project organizer
- ▼ PostScript preview
- ▼ Built-in PDF distiller
- ▼ Export to Adobe Illustrator
- ... and more!



Pricing starts at \$99, complete system including MathTimeProfessional fonts is \$399. Upgrades from earlier PCT<sub>E</sub>X versions start at \$29.

PCT<sub>E</sub>X v5 also offers

## The *MathTimeProfessional* Fonts

Mathematics symbols to match the Times fonts,

$$\alpha\beta\gamma\delta \pm \zeta\kappa\xi\psi\omega \mp \partial f(x, y, z)/\partial x$$

with *individually designed fonts* for different sizes

$$\alpha\beta\gamma\delta\Gamma\Delta\Theta \dots \alpha\beta\gamma\delta\Gamma\Delta\Theta \dots \alpha\beta\gamma\delta\Gamma\Delta\Theta \dots$$

to provide more readable symbols in superscripts

$$\sqrt{\frac{x^y + \alpha^{\beta\gamma} + \Theta^{\Delta\Gamma} + f_{\alpha\beta}(x_{ij}^{\lambda\rho}, y_{ij}^{\mu\sigma}, z_{ij}^{\nu\tau})}{\epsilon\eta\theta + \iota\kappa\lambda + \pi\rho\sigma + (\vartheta\varpi\varphi + \Xi\Phi\Omega)^{\phi\chi\epsilon + \Psi}}}$$

**And:** parentheses and radicals up to **4 inches high** and math accents extending up to **4 inches wide!**

With easy to use guide for L<sup>A</sup>T<sub>E</sub>X and plain T<sub>E</sub>X



For more information and for a Free 30-day evaluation, visit [www.pctex.com](http://www.pctex.com)  
 Personal T<sub>E</sub>X, Inc. 800-808-7906 415-296-7550 [sales@pctex.com](mailto:sales@pctex.com)

The T<sub>E</sub>X Users Group gratefully acknowledges Apple Computer's generous contributions, especially to the *Practical T<sub>E</sub>X 2004* and *TUG 2003* Conferences.

*Thank you.*

The Apple Store in San Francisco is located at One Stockton Street, San Francisco, CA 94108<sup>1</sup>

(This was typeset with the T<sub>E</sub>X variant X<sub>Y</sub>T<sub>E</sub>X<sup>2</sup> created by Jonathan Kew using the Apple System fonts HOEFLER TEXT by Jonathan Hoefler, ZAFFINO by Hermann Zapf and SKIA by Matthew Carter.)

<sup>1</sup><http://www.apple.com/retail/sanfrancisco>    <sup>2</sup><http://scripts.sil.org/xetex>