Forward

0.1 What is Ptolemy

PTOLEMY is a prototype subsystem for the automatic setup of the numerical solution of Partial Differential Equations (PDEs) via sinc-collocation.

PTOLEMY is a symbolic computing package written in Maple. It does not pretend to advance to the field of computer algebra. **PTOLEMY**'s primary contribution is to demonstrate the feasibility of a new kind of Problem Solving Environments (PSEs), one which combines symbolic computing, numerical computing, geometric computing, and a graphical interface. Although current PSEs-tend to combine numerical computing, geometric computing, and graphical interfaces significantly integrating symbolic computing with such PSE's advances the state of the art.

However, **PTOLEMY** is not a complete PSE, it is just the symbolic portion of that system. It is part of a larger research project and has been used in more complete problem solving activities, but many part of the problem solving process have not been fully integrated with **PTOLEMY**. The notable omissions are:

A parallel numerical linear algebra solver. As part of this research project the author developed such a solver in C++. In fact this represented roughly one third of the research effort. Unfortunately, changes in the symbolic algorithms used by **PTOLEMY** have compromised the ability of the "back-end" to work well with **PTOLEMY**. In addition improvements in packages such as MatLab make it possible to use **PTOLEMY** without a tightly integrated numerical linear algebra solver.

A symbolic visualization mechanism. Maple is not a graphing package and does not provide vary sophisticated visualization tools. Nevertheless, the ability to symbolically describe the desired visualization allows the user construct visualizations which are richer in meaning for the same level of visual quality. The author has developed some modules to demonstrate the potential of hybrid visualization. However, their operation is vary sensitive to the operation of the parallel numerical linear algebra solver and their quality is limited by Maple's graphical capabilities.

A graphical symbolic geometric specification mechanism. The mathematics for graphically specifying symbolically well behaved geometries is well understood. Until recently the investment in code required to create a custom Graphical User Interface (GUI) was well beyond the resources available to this author. However, recent advances in higher level and more portable GUI prototyping environments (particularly TCL TK and Java), suddenly make such an effort feasible.

In spite of these omissions, **PTOLEMY** may be fairly construed as a the core portion of some futuristic symbolic-centric PSE.

The primary numerical advantage of hybrid computing is that it makes the application of more powerful numerical methods practical to a more general class of problems. Numerical methods more powerful then those currently used in state of the art computational engineering systems have been well understood and even frequently applied for most of this century. However, these methods are not part of the mainstream in computation engineering because they are not practical for industrial engineering applications. Most often this is because they require several person years of development effort to apply the method to each new problem. Given the choice between years of effort to apply a powerful numerical method and an hour to apply a weak method, the weaker method is usually the better choice. Of course, the obvious solution is to develop tools which automate and simplify the problem solving process. This approach has gradually made Finite Element Method (FEM) more prevalent then Finite Difference Method (FDM) in computational engineering. Prior to the development of powerful grid generation tools, the grid adaptation (which is necessary to realize the potential of FEM) was often just too difficult to justify. The primary impediment automating even more powerful methods has been the large amount of symbolic math associated with most such methods. By automating the symbolic math parts of the problem solving process it should be possible, in time, to create PSEs which dramatically improved numerical performance.

PTOLEMY demonstrates this approach for sinc-collocation. For this particular method the primary symbolic operation is mapping the problem's subdomains (i.e., performing changes of variable) so that the convergence of sinc-collocation is $O\big(\exp(-c\sqrt{n}\big)$, where c is a constant and n is the size of the linear algebra system to be solved. **PTOLEMY** actually performs this mapping in several steps. The result is that much of the package is devoted to mapping.

0.2 Why Maple

PTOLEMY is based on Maple V. It is assumed that the human users are proficient at manipulating Maple data structures.

Maple makes **PTOLEMY** surprisingly general, by supporting a wide range of mathematical function. Rewriting **PTOLEMY** in C++ would not be especially challenging as long as the set of supported mathematical functions was quite limited, but the result would dramatically limit the ways in which the user could model the problem. Providing even minimum support for the number of mathematical functions supported by Maple is a significant development effort.

Another significant advantage to using Maple is that it provided fast prototyping during the research phase of this project. During much of the development of this project, I only clearly understood a portion of the design. If the cost of trying things had been so high as to preclude the possibility of implementing something that might latter be found to be wrong, I would have never finished this project. To some extent it is only possible to see how to better implement this in C++ now that it has been done once in Maple.

Finally, it is not clear that rewriting this in C++ is a particularly good idea. The interpreted implementation of Maple makes **PTOLEMY** somewhat slower then a fully compiled implementation of **PTOLEMY**Although this loss of efficiency is two to three orders of magnitude for numerical operations, it is only about a factor of two for the symbolic operation. The strategy employed by **PTOLEMY** is to only perform symbolic operations in Maple and to perform any costly numerical operations in C++. This method works fairly well. The primary price of using Maple is the relatively primitive capabilities for interfacing with other applications and the limitations of Maple's graphics.

0.3 Who Was Ptolemy

PTOLEMY is named after the well known Greek scholar. There are several parallels between this work and the primary contributions of this scholar:

- Ptolemy was the father of Geography. He wrote a 13 volume treatises on geographic method that remained the quintessential reference on geographic method well into the Renaissance. In this work he:

  1. Systematize the use of mathematical maps for cartography. It is as a result of this work that in contemporary English "charts" are most often referred to as "maps." Although he was not the first to use such maps, he was the first to reduce the use of such maps to a systematic method.
  2. Provided several sample mathematical maps for use in constructing charts of larger portions of the earth. Assuming that the world was spherical much of the work struggles to minimize the distortions of scale so common in cheaper 20^{th} century maps of the world.
  3. Made one of the more prominent early attempts to chart the world. However, due to an over reliance on hear-say, many of the details of this map were inaccurate. It never-the-less represented a significant contribution to the representation of the world.

- Ptolemy was also the prominent astronomer of his day. He wrote a 21 volume treatises on the motion of the planets. In this work he:

  1. Made one of the first applied uses of a Fourier Series to approximate observational data.

2. Credited Augestous of Purga (known as "The Great Geometer") with first inventing the mathematics of such a series. Because the book, *The Spherics*, in which Augestous is reported to have documented this development is extinct, it is only because of Ptolemy's success at applying this bit of mathematics that we know of Augestous's discovery of this result.

For the benefit of those who are unfamiliar with sinc-research the parallels are a bit surprising:

- The systematic remapping of physical problems to rectangular domains is at the heart of the **PTOLEMY** software package. Just as the man Ptolemy was not the first to use this method, **PTOLEMY** the software package is not the first to use this approach. The software is a direct result of the first attempt to reduce the process to our modern equivalent of a systematic method, a computer program.
- The Ptolemy software package attempts to provide a set of standard mapping procedures for practitioners of sinc-collocation in a way that is analogous to the mathematical maps which the man Ptolemy provided for the cartography community.
- The software package **PTOLEMY** grew out of an effort to chart the new unexplored field of automatic setup for sinc-methods. Although this may not really be analogous to attempting to map the world, in my first year of graduate school the metaphor seemed strong. In a vary real seance the software package **PTOLEMY** was conceived with aspirations of achieving a very coarse chart of the whole world PSEs for sinc-methods.
- Although the Fourier series used by Ptolemy the man to describe the motion of the planets was of very low order, the approach is exactly the same as the modern use of sinc-methods. This equivalence has not been published in the sin literature. I first recognized this equivalence as a direct result of this project.
- Finally, I suspect that the way future generations will remember early sinc-research will not depend primarily on the elegance of the great mathematicians, or even on the prolificness of the lesser mathematicians. Instead it will depend on the success or failure of efforts to apply sinc-methods to contemporary engineering problems. This research effort aspired to make the latter kind of contribution.

I have been surprised at the level of scorn many have for Ptolemy the man. Many seem unable to forgive him for the Papal use of his name in its effort to suppress Renaissance science. Naming this software after Ptolemy is *not* an endorsement of the ill-treatment of Galileo or others who eventually replaced the "Ptolemaic System." I have found no evidence which suggests that Ptolemy would have approved of the way others used his name for unrelated purposes nearly a millenia and a half after his death.

Sure the Ptolemaic System was wrong, but so were the heliocentric models of his day. I do not believe that many of the methods employed by **PTOLEMY** the software package will stand the test of time. I can already see how to do many things better. I would suggest that even the life work of the greatest researchers in history have not been great because they archived truth but rather simply because they proved useful. If **PTOLEMY** can prove useful to sinc-researchers I will have archived some miniscule measure of greatness.

0.4 About This Manual

One of the major purposes of this manual is to provide maintainers of **PTOLEMY** with sufficient information to evolve the "product." Originally, the manual contained much more length descriptions of the high level mathematics and concepts behind this software. However, over time much of this high-level material was extracted into other writing (notably my dissertation). The result has become a more or less traditional programmers manual. However, since at this time using **PTOLEMY** involves a kind of programming the manual is really a *programmer/user manual.*

Readers who consider themselves more users then maintainer or developers of the package can safely read Chapter 3 to Chapter 6. Sinc researchers familiar with Maple should find most of this this manual self contained. The primary exception is a description of SB-notation which can be found in [7]. Readers new to sinc methods will probably benefit by simultaneous reading parts of [5] or [12]. And readers new to Maple will almost currently benefit by first reading large portions of [3].

0.5 Acknowledgments

I started this project during the summer of '91 and excluding 7 quarter long interruptions (mostly devoted to teaching responsibilities) have worked on it ever since.

During this time I've been supported by more than one benefactors. The bulk of the funding came from two grants. The first was from IBM which funded the development of a parallel linear algebra solver for use with **PTOLEMY** and an early, failed, attempt to implement **PTOLEMY** in C++. The bulk of the funding for the current implementation was provided by an NSF grant (number CCR-9307602). In addition I received 1 1/2 quarters of support from the Department of Computer Science at the University of Utah, between \$500 and \$600 from the University of Waterloo, and \$250 from the Department of Energy.

During the time i worked on this project Frank Stenger published a book containing a great deal of new mathematical material relevant to this project, i.e., [10]. He showed me initial drafts of this book as early as Fall of '90, i.e., almost three years before the book appeared in print. Since it took me over a year to understand some of the subtler but essential aspects of the theory, this

provided me with a significant advantageous over those who where not blessed with such access.

Mike O'Reilly tutored me on sinc methods during the first year of this effort. Though much of the project was not directly dependent on this help almost all of the creative insights I've had over the years about the mathematics of sinc methods where the result of seeds planted by Mike.

Kenneth W. Parker
University of Utah
Salt Lake City, UT, USA

Autumn 1996
`kparkerfacility.cs.utah.edu`

Contents

List of Figures

Glossary

automatic data type An organizational unit for source code that implicitly defines a data structure by explicitly defining the operations that may be performed on the data structure. The actual implementation of the data structure is not visible outside the automatic data type. Standard usage in computer science.

bases defined to be an orthonormal spanning set for a specified linear field. In this context the bases is explicitly specified and the field is implicitly defined as the space spanned by the bases. More specifically the bases are the functions used to approximate the state-variables and the field is the set of functions which can be exactly represented by a linear combination of these bases. Standard usage in numerical analysis.

basis components If a specific n-dimensional basis can be represented as the product of n one-dimensional functions, then each of these one-dimensional functions is a bases component. In this document as a matter of convention all n-dimensional bases are constructed to be the product of n one-dimensional bases components.

basis component groups For the class of problems considered in this document each dimension would typically have tens of bases components. This document partitions the bases components in each dimension into groups according to the way they behave with respect to differentiation. In particular all of the sinc bases components in a specific dimension are grouped together in the zeroth group. Spline bases components are all assigned other unique groups. Those spline bases components corresponding to collocation points near the lower end of the interval are assigned negative numbers and those corresponding to collocation points near the higher end of the interval are assigned positive numbers.

bases groups Bases are grouped according to the bases component groups of the bases components used to construct the bases. Each bases group is assigned a coordinate that is the cartesian product of the bases component group numbers of the bases components used to construct the bases within the group.

block order The order of the matrix of blocks representation of a block matrix. A block order of $n \times m$ means that the matrix has n rows of blocks each containing m blocks. For a square matrix the order is sometimes expressed as a single number.

block column A column in the matrix of blocks representation of a block matrix.

block matrix A matrix where the rows and columns of the matrix have been partitioned. Each block of the matrix is the collection of elements in the same row and column partition. It is sometime useful to to think of a block matrix as a matrix of blocks. Standard usage in computer science.

block row A row in the matrix of blocks representation of a block matrix.

boundary orientation The change in parameter order and direction occurring between two equivalent parameterizations of the same boundary. Sometimes used to express the change between a specific parameterization of a boundary and a standard reference parameterization of the same boundary.

Note: a parameterized boundary is not altered by changes in the order in which the parameters are specified or by the direction in which the parameters are traversed.

cartesian ordering An ordering of a set that is the cross product of several sets constructed by specifying both an intraset and an interset ordering for the sets used to form the cross products. A cross-product is in cartesian order if its elements are ordered according to the component from the first set, if for all the elements with the same component from the first set are ordered according to the component from the second set, and if this nested ordering is continued for all the sets used to construct the cross product.

collocation point The geometric point at which a collocation equation is evaluated. Standard usage in numerical analysis.

collocation equation The evaluation of one of the equations in the system of PDEs at a specific collocation point using the approximation for each of the state-variables that appear in this equation.

collocation event The pairing of a collocation point and a collocation equation. For PDEs with only one state-variable there exists a one-to-one correspondence between collocation events and collocation points, but for systems of PDEs the same collocation point will typically be used for each of the collocation equations.

column major ordering An ordering of the cross product of row and column data which is a cartesian order in which the column set is the first set of the cartesian ordering. Usually used to refer to the ordering of the indices of a matrix. Common usage when talking about Fortran, somewhat historical reference in most other contexts. Standard usage in computer science.

component A connected subgraph. That is a portion of a graph that is closed with respect to connectivity. Standard usage in computer science and math.

dynamic scoping A scheme for nesting local variables scopes where the scopes are nested according to the nesting of the procedure calls. This scheme typically causes the scoping hierarchy to change during program execution. Implicitly provided by Maple's "environment variables." Standard computer science usage.

interpret The act of executing source code by directly performing the actions specified by the code. Interpreting is performed by another program called an interpreter. This is usually understood in contrast to the more common technique of compilation. Standard usage in computer science.

lexical scoping A scheme for nesting local variables scopes in accordance with the nesting of the source code. The type of scoping employed by Pascal. Standard computer science usage.

module An organizational unit of source code in which the scope of reference may be more restrictive then the scope of existence. Standard usage in computer science, more specialized then the common usage in other fields.

override equations A collocation equation which replaces the default collocation equation at a specific group of collocation points.

row major ordering An ordering of the cross product of row and column data which is a cartesian order in which the row set is the first set of the cartesian ordering. Usually used to refer to the ordering of the indices of a matrix. The default ordering for matrices in most programming languages. Standard usage in computer science.

scope The region of source code in which variables either exist or may be referenced. Standard computer science usage.

sinc maps The portion of the conformal map used in sinc approximation which maps an oriented parallelepiped to the an infinite space. Typically chosen for the region of the complex hyper-plane mapped to $\mathcal{D}_d^n$.

transformation A "function" from n-space to m-space where $n, m \in \mathbb{Z}^+$. Standard mathematical usage.

type expression An expression which defines a collection of types. Standard usage in computer language analysis.

Chapter 1

General Information

In describing the implementation some pseudo-standard typographical conventions are employed. These conventions are described in the first section of this chapter. The rest of the chapter describes some Maple conventions used throughout the implementation. These conventions, like the typographical conventions, are similar to the conventions used by others but can are not standard either because of special needs or, as is more often the case, because of the lack of an established standard.

1.1 Typographic Information

Computer manuals typically have a very large number of words which within the context of the manual have some special meaning. In more literary styles of prose, where such technical terms are far less frequent, such words might be enclosed in quotes to indicate that a special meaning is intended. However, this practice is rarely followed in computer manuals. One reason is that the number of quoted words would become excessive and cumbersome, but probably the more important reason is that the connotations conveyed by using quotes in literary prose are somewhat different then the connotations associated with using technical terms. As a result computer manuals typically continue to use quotes in a way similar to their use in literary prose, but invent other ways of distinguishing technical terms from the nontechnical words with the same spelling.

To see why this might be, consider a manual that used the word "sum" to mean the aggregating of parts, a variable name or argument name in a procedure interface, the name of a procedure for performing summations, the name of a file in which partial sums or running sums are stored, and a user command to be typed in response to a prompt in order to indicate that now is the time to perform a particular summation. Without some kind of indication of how "sum" is being used, it would often be confusing to the reader exactly what was meant. To resolve this problem computer manuals have a long history of inventing typographical constructs not only to indicate that a word is a technical term but also to give the reader some indication of which of several possible technical meanings are intended.

Within this tradition, quoting is preserved more or less as it occurs in non-technical writing. For example, in the sentence, "If the building burns then the accounts will have to be 'summed' by hand," the quotes around the word "summed" indicate that the process being eluded to is actually more than (or otherwise slightly different) than literal summation.

Due to editorial pressures (real or perceived), I have attempted to avoid some of the excesses of this tradition, such as the use of underlining or special colors. However, I have found it useful to invoke some of the more common conventions of this tradition.

Normal prose is set in the Computer Modern Roman (CMR) font and look like most of the text on this page. Within the context of normal prose italics are used for emphasis. For example the word "not" in the sentence, "It might seem like ..., but this is decidedly *not* the case," is italicized to stress or emphasize the word. This use of italics is common to most modern semitechnical writing. In addition to this typographical construct this manual uses a special font to indicate that a word or phrase is: 1) the name of a program variable, 2) a type name, 3) a procedure name, 4) a library file name, 5) a general filename or a program name, or 6) some other kind of "computer syntax" (e.g., I/O code fragments, and the like).

Computer variable names are set in the Teletype Font and look like `var_name`. Type names are set in a slanted variation of Sans-Serf Font and look like ***type_name***. Procedure names are set in the bold variation of the CMR font, and look like **proc_name**. Library file names are set using the bold-italic variation of the CMR font and look like ***lib_file_name***. All other file names (excluding lengthy path names which are treated as general computer syntax), program names, and system names are set in a slightly enlarged small-caps variation of the CMR font and look like **PTOLEMY** or **.MAPLEINIT**. Notice that the size difference between the letters indicates the case, so the first and second 'S's in **SOLVESYS** are capitalized whereas **SOLVE_SYS** contains no capital letters. Finally, all other computer syntax, such as input commands and code fragments, are set using the Teletype Font, and look like `Sum := subs(N = Name, readlib('N'));`.

Notice that in the last example the typographical information helps make it clear that the semicolon is part of the input sequence being specified and *not* part of the punctuation of the enclosing sentence.

Further notice that the font used to indicate that a word is a program variable is the same as the font used for nonspecific computer syntax. A distinct font could have been used to distinguish program variables form other code fragments. The distinction is usually clear by context and the potential problems associated with failing to correctly distinguish these two classes of technical terms are minor. In the end it was my opinion that the added typographical clutter was not justified.

1.2 Code Structures in Maple

PTOLEMY is a Maple *package* just like ***plots*** or ***linalg***. A Maple package simulates a separate name space for the global variables of the package. In actuality, however, Maple only has one name space, so packages are really a coding convention for simulating the effect of separate naming spaces.

Packages

The key idea of this coding convention is that global variables in the package are assigned names of the form `package_name/var_name`, where `package_name` is the name of the package and `var_name` is the *nominal* variable name. The result of this convention is that global variables in one package cannot conflict with global variables in another (unless, of course, the package names conflict). This frees the package implementor from the impossible task of worrying about whether his variable names conflict with the variable names of some other package implementor.

Because names containing slashes must be enclosed in back-quotes, typing the variable names of a package is particularly awkward. Since at least some of the package's global names must be intended for external reference, in at least some cases this awkwardness will impact the package user. To reduce this inconvenience, each package has associated with it a table with entries that reference some of the global variables of the package. This allows the user to type `package_name[var_name]` when referencing these global variable `package_name/var_name`. This table is assigned to the global variable with the package name, and is referred to as the *package table*.

Because it is slightly faster to avoid going through the package table to reference its global variables from within the package, the global variables of the package are usually references directly by their full global name. So it is useful to include entries in the package table only for package elements that the user is expected to directly use. In fact, this concept is usually taken one step further and the package table is used to *specify* which elements are intended for external use.

Sometimes if a particular package will be used frequently the user may find it more convenient to reference the global variables of the package by their nominal names. Of course, this makes sense only if the nominal name, in question, does not conflict with any currently defined global names. When this is the case the user could enable direct use of the nominal name with the following assignment,

```
var_name := 'package_name[var_name]'
```

In fact this is actually what the Maple's `with` command does. The on-line help page for `with` creates the impression that the command loads portions of the package. This is the intended purpose for creating the command, but loading is actually accomplished by assigning the nominal name to the value of the corresponding package entries, so that the corresponding Loadable Library File (LLF) will be loaded when the nominal name is next fully evaluated.

Another important feature of the `with` command is that it will automatically execute any initialization procedure defined by the package table. If a package table entry is defined with the index `init` then it must specify the package initialization procedure. The package designer may perform any operation in this package initialization procedure; **PTOLEMY** uses the initialization procedure to define new types used by the package.

See Maple's on-line help page titled "with" for more information on the `with` command.

In computer science the word "module" is often used as a technical term to define a specific language feature. See [11] for an informal introduction to this formal meaning of the word "module." Packages may be thought of as closely approximating this formal kind of module. The package table defines those global variables that are to be *exported* by the module, and the user of the package may *import* some or all of the variables exported by the module by using the `with` command.

However, this analogy can be carried only so far. Maple does not, at this time, have lexical scoping (version 5.5 is expected to support lexical scoping). The result is that the `with` command imports the package's global variables into the dynamically defined global scope, whereas modules import the package's variables into a lexical scope. In fact the distinction associated with scoping is greater than just that Maple packages use dynamic scoping and that modules use static scoping. Maple does provide limited support for dynamic scoping, of the type found in Lisp, thru what Maple calls "environment variables." However, in Maple,when a package is loaded, the definitions of nominal names are not defined as environment variables. As a result these definitions persist forever, and do not disappear once the dynamic scope is exited. The result is that loading many packages starts to seriously pollute the name space. A plausible workaround would be to implement an `unwith` command.

Another major distinction between packages and modules is that modules restrict access to the package's components to just those variables that are explicitly exported. This allows the package designer to create private variables with a global scope of existence and a local scope of reference. Again Maple does provide limited concept of protected global variables, but this concept is not integrated with the package concept. As a result part of the coding convention of packages must be that users of the module simply do not access any of the package's global variables that are not found in the package table. Unfortunately, Maple does not enforcing this convention.

Another problem with thinking of Maple's packages as modules is that if the user accesses the package's global variables through the package table the package's initialization procedure may not have been executed. For cultural reasons it seems untenable to propose that part of the coding convention of a Maple package should be that all package elements must be "imported" with the `with` command before they can be accessed. As a result the package designer has to take responsibility for ensuring that the initialization procedure has been executed before the user performs any kind of package access whose result is dependent on the initialization procedure. **PTOLEMY** achieves this by including a wrapper function in all table entries that performs package initialization if it has not already occurred. Most of the system packages avoid using a package initialization procedure, to avoid having to deal with this problem.

Library Files

Because much of Maple's functionality is implemented in the Maple language and all Maple code can be loaded dynamically, the Maple system can be loaded incrementally only as needed. In fact, much of the Maple system is not initially loaded, but rather is loaded on demand.

Maple defines the granularity of code that can be loaded to be the portion of the code within one dot-'m' file. Usually there is a one-to-one correspondence between dot-'m' files and source files; however, on occasion some developers will combine several source files to create one dot-'m' file. The name used to describe the collection of code placed in one dot-'m' file varies slightly throughout the Maple literature. I have chosen to consistently refer to it as the minimum code unit which can be dynamically loaded as a *LLF*. **PTOLEMY** source file names all have a '_m' extension; executing the source (i.e., in Maple) will produce a file with the same name as the source but with a '.m' extension. The contents of the resulting dot-'m' file will be the code defined in the source file. Because there is a one-to-one correspondence between source files and dot-'m' files, in this manual the term *LLF* refers to the logical entity corresponding to this physical pair of files.

LLFs may be loaded with the `read` command or the `readlib` command. The primary advantage of the `readlib` command is that it allows the path name to be specified relative to the library path list defined by the global variable `libname`. However, Maple places an unusual restriction on LLFs that can be loaded with `readlib`; specifically, the library file must define a global variable with the same name as the *relative* path name of the file. Although this restriction is at times cumbersome, the advantages of using relative path names outweigh the drawbacks and **PTOLEMY** defines only `readlib`-able LLFs.

This works reasonably well if one of the global variables defined by the LLF is of paramount significance within the LLF. In this case it is usually not difficult simply to make the (relative) file name the same as the prominent global variable. If the prominent element of the LLF is an exported element of the package, this implies that the relative file name must be of the form `package_name/var_name`. This is easy to do as long as there is a one-to-one correspondence between exported package elements and LLFs, since the LLF will all occur in a subdirectory of the library path with the package name. However, if either several package elements or no package elements are defined within a single LLF, it becomes unclear which file name is most appropriate. **PTOLEMY** avoids the case of loadable library modules that define more than one prominent package element by never defining more than one exported package element within a single LLF.

However, **PTOLEMY** does have some source files that do not define any exported package element. Usually such LLFs define a collection of related support procedures which are used by more than one of the exported package elements. In these cases, rather than arbitrarily picking one of the globally defined variables to be the file name, **PTOLEMY** chooses a descriptive file name and defines a "no-op" procedure with the same name as the file name. This is done purely to avoid having to use confusing file names but still be able to `readlib` all of the LLF. One benefit of this approach is that it is easy to check if a particular LLF has been loaded; simply check if the global variable with the same name as the file name is of type *procedure*.

Table 1.1 lists all of the **PTOLEMY** modules that contain a no-op primary procedure.

1.3 Loading and Initialization

The loading and initialization portion of **PTOLEMY** creates definitions that have little use by themselves but without which the package as designed will not work. These definitions mostly facilitate dynamic on demand loading, and create types required for interfacing with the package. There are only three LLFs in this portion of the package, ***ptolemy***, ***init***, and ***types***.

The ***ptolemy*** LLF defines the package table. The entries of the package table are not initialized to point to the global variables of the package as might be suggested by way the package table is used. Rather the package table entries are initialized to a command for loading the LLF which defines the associated package variable. This means that all of the components of the package remain unloaded until they are actually referenced. The ***ptolemy*** LLF also defines a procedure, **ptolemy/load**, to make the defining of the package table entries easier.

The ***init*** LLF defines the package's initialization procedure. Package initialization primarily involves defining types needed to interface with the **PTOLEMY** package, but also establishes the default values of the exported global variables

Table 1.1: Modules With "No-Op" Primary Procedures.

b_ops	Procedures for manipulating B-notation.
comb_ops	Procedures for numbering the combinations of extra bases that occur in B-notation.
kron_ops	Procedures for converting expressions in SB-notation to expressions in the Kronecker product notation.
map_info_ops	Procedures for manipulating *MapInfoType*s.
order_ops	Procedures for manipulating *SubOrderType*s, *OrderTypes*s, and *OrderSpecType*s.
proc_dimen_ops	Procedures for determining the dimension of a mapping type of procedure.
types	Definition of all of the package-specific types.

`SimpProc` and `OneDRange`. If part of **PTOLEMY** is loaded using the `with` command, then the initialization procedure will automatically be executed at that time. In addition if the **ptolemy/load** procedure is executed, presumably because one of the package table entries was evaluated, the initialization procedure will be executed at the same time.

The ***types*** LLF defines *all* of the **PTOLEMY**-specific types. These types might be needed in order to run some user created code to manipulate **PTOLEMY** defined objects even if no part of the package is used.

Though a significant effort is made to ensure that the initialization procedure will be executed before any of the package's components are evaluated, there are times when a user will want to explicitly initialize the package. This can be done with the command `ptolemy[init]()`. An example of when this might be necessary is if the user performs some lengthy problem defining computation prior to the first usage of a **PTOLEMY** procedure and as part of this defining computation it is necessary, or convenient, to use some of the **PTOLEMY** specific types.

In fact, it is this occasional need to explicitly load the package types without actually using the package, that motivates making the package initialization procedure explicitly visible. Without this need, each component of the package could "secretly" ensure that any structures that it requires are defined as part of its execution.

ptolemy

This LLF defines the package table. As mentioned in the introduction to this section the entries of the **PTOLEMY** package table are actually commands for loading the appropriate LLF and defining the appropriate global variable.

This LLF defines one global variable, `ptolemy`, and two procedures,

load_proc(`NomName`: *name*)
check_init(`NomName`: *name*)

The global variable `ptolemy` is the package table. The **load_proc** procedure and the **check_init** procedure are used to define the package table. Specifically the **load_proc** procedure loads the procedure with the nominal name `NomName`, and the procedure **check_init** checks that the package has been initialized as part of the reference to exported global variable `NomName`.

Method of Dynamic Loading

All the information for loading the **PTOLEMY** LLFs is defined in the package table. From the definition of the *package table* each entry in the table must evaluate to a reference to the variable in the package with the nominal name of the entry's table index. See Section 1.2 for more information on Maple Packages, but the **PTOLEMY** package table entries are not direct references to the appropriate package variables. Instead they are commands that when executed load the LLF in which the package variable is defined and then return a reference to the variable.

This allows the package components to be loaded only when they are referenced. Often this avoids unnecessary work, since referencing every **PTOLEMY** LLF in the same session would be the rare. More importantly, it helps reduce response time by distributing package loading over many user interactions (waiting for I/O is the biggest component of response delay for most Maple interactions).

The Package Table

At this time the package table exports only the global variables presented in Table 1.2.

Each package table entry corresponding to an exported procedure will equal an expression of the following form

```
`ptolemy/load_proc`(nominal_name)
```

This implies that fully evaluating the table entry will execute **load_proc** with the nominal name as its argument. The procedure **load_proc** must be responsible for performing all the necessary operations. These operations are enumerated in the next subsubsection.

Referencing the package variable `SimpProc` requires package initialization, but does not require that any LLFs be loaded. As a result its package entry is equal to

```
`ptolemy/check_init`('SimpProc')
```

Table 1.2: The Global Variables Exported by the **PTOLEMY** Package.

init	See page 14		
ClipPlot	See page 191.	MakeRecMap	See page 76.
CollocateRec	See page 117.	MultiToRec	See page 96.
CollocateMultiRec	See page 129.	MultiSToLinKron	See page 152.
Linear	See page 142.	SolveLinKron	See page 165.
LogRatioMap	See page 105.	SToLinKron	See page 146.
LogSinhMap	See page 113.	ToRec	See page 87.
MakeTradMap	See page 79.	Warp	See page 69.
GridInvMap	See page 181.	PlotDomBound	See page 176.
GridMap	See page 187.	sinc	See page 197.
PlotBound	See page 173.		
SimpProc	A global variable used to specify the simplification procedure is to be used.		
OneDRange	A global variable used to specify the range used for graphing 1D grids.		

The procedure **check_init** is similar to the procedure **load_proc**, except that it does not try to load the corresponding LLF.

In the source code the actual table entries are of the form

```
eval(nominal_name = ''ptolemy/load_proc'(nominal_name)'
```

or

```
nominal_name = ''ptolemy/check_init'(nominal_name)'
```

so that when the source is "compiled" the **load_proc** and **check_init** are merely referenced, not executed.

Importing *Ptolemy* Variables The **with** procedure assigns the variable `proc_name` to the result of

```
eval(package_name[proc_name],1)
```

and executes the result of evaluating `package_name[init]` if that result is a procedure. The `with` command does not actually load any LLFs. This is especially counterintuitive, since it is common to refer to the act of executing a `with` command as "loading package components." Technically, the result of executing a `with` command is to create definitions such that evaluating either the package table entries or the variable name "loaded from the package" will cause the associated LLF to be loaded.

This causes a problem if the `with` command is used to assign the nominal variable name a command sequence for loading the associated LLF and then the nominal variable name is used as a symbol. Since Maple does not fully evaluate variables that reference procedures, a variable that is assigned a procedure may be used either as a name or as a procedure, depending on the context. So had the nominal variable name been assigned a procedure instead of a command that eventually evaluates to a procedure, it would be valid to use the nominal variable name as a symbol. Unfortunately, the commands for loading a procedure from an LLF cannot be used as a name.

To remedy this problem **load_proc** must check if the nominal variable name has been assigned to the result of

```
eval(package_name[proc_name],1)
```

If this has happened then **load_proc** must reassign the nominal variable name to the procedure which it is loading, after the procedure has been loaded.

The load_proc Procedure and the check_init Procedure

The **load_proc** procedure will:

- Check if the package initialization procedure has been executed. If it has not been executed, then execute it.
- Check if the associated nominal variable name has been assigned the value found in the specified table entry. If it has, then the nominal variable name will be modified whenever the table entry is modified.
- Assign the specified package table entry to directly reference the full name of the procedure that will be loaded.
- Use the `readlib` command to load the LLF with the same relative path name as the full variable name.
- Assign the nominal variable name the procedure and return the result, *if the nominal variable name is assigned to the value that use to be in the table entry then.* Just return the procedure if this is not the case.

This prevents **load_proc** from being reexecuted on subsequent references to either the same package table entry or the same nominal variable name.

Because subsequent evaluations of the same package table entry will directly evaluate to the full procedure name, there is rather little overhead caused by referencing package components through the package table and there is no overhead associated with referencing the procedure via its nominal name after the `with` command was used to import a procedure.

The **check_init** procedure performs the same operations as **load_proc**, except that no LLFs are loaded. This is useful when referencing global variables of the package that do not have an associated LLF.

However, because using the `with` command to load portions of the package may cause naming conflicts, it should never be used by code within the **PTOLEMY** package. Furthermore since loading LLFs through the package table is more expensive then directly using the `readlib` command this method is never used by **PTOLEMY** procedures to load other portions of the package that it might need. Finally, if it is known that the necessary LLFs have been explicitly loaded there is little reason not to reference the procedures directly by their full names. In short, the package table is intended to create a view of the package's exported variables for the outside world, it is not intended for use internal to the package.

The init Procedure

Because the `with` command makes nontrivial internal use of references to `ptolemy[init]`, using this mechanism to load the initialization procedure can lead to a self reference within the `with` command which leads to an infinite loop.

As a result **PTOLEMY** always pre-loads the ***init*** LLF. Specifically, this LLF is loaded during the compilation of the ***ptolemy*** LLF and the **init** procedure is assigned directly to `ptolemy[init]`. However, the two source files are kept separate to facilitate experimentation with other design options.

An Example

Start of Maple Worksheet

```
> interface(verboseproc=0);
> eval(ptolemy,1);
```

$$\mathrm{readlib}(\,'\mathit{ptolemy}'\,)$$

```
> ptolemy;
```

$$\mathit{ptolemy}$$

```
> eval(ptolemy,1);
```

table([
$\mathit{MultiSToLinKron} = \mathrm{ptolemy/load\_proc}(\,'\mathit{MultiSToLinKron}'\,)$
$\mathit{CollocateMultiRec} = \mathrm{ptolemy/load\_proc}(\,'\mathit{CollocateMultiRec}'\,)$
$\mathit{OneDRange} = \mathrm{ptolemy/check\_init}(\,'\mathit{OneDRange}'\,)$
$\mathit{SimpProc} = \mathrm{ptolemy/check\_init}(\,'\mathit{SimpProc}'\,)$
$\mathit{GridInvMap} = \mathrm{ptolemy/load\_proc}(\,'\mathit{GridInvMap}'\,)$
$\mathit{ClipPlot} = \mathrm{ptolemy/load\_proc}(\,'\mathit{ClipPlot}'\,)$
$\mathit{LogRatioMap} = \mathrm{ptolemy/load\_proc}(\,'\mathit{LogRatioMap}'\,)$
$\mathit{LogTanMap} = \mathrm{ptolemy/load\_proc}(\,'\mathit{LogTanMap}'\,)$
$\mathit{LogSinhMap} = \mathrm{ptolemy/load\_proc}(\,'\mathit{LogSinhMap}'\,)$
$\mathit{MakeRecMap} = \mathrm{ptolemy/load\_proc}(\,'\mathit{MakeRecMap}'\,)$
$\mathit{MakeTradMap} = \mathrm{ptolemy/load\_proc}(\,'\mathit{MakeTradMap}'\,)$
$\mathit{MultiToRec} = \mathrm{ptolemy/load\_proc}(\,'\mathit{MultiToRec}'\,)$
$\mathit{PlotBound} = \mathrm{ptolemy/load\_proc}(\,'\mathit{PlotBound}'\,)$
$\mathit{SToLinKron} = \mathrm{ptolemy/load\_proc}(\,'\mathit{SToLinKron}'\,)$
$\mathit{SolveLinKron} = \mathrm{ptolemy/load\_proc}(\,'\mathit{SolveLinKron}'\,)$
$\mathit{GridMap} = \mathrm{ptolemy/load\_proc}(\,'\mathit{GridMap}'\,)$
$\mathit{should} = \mathrm{ptolemy/load\_proc}(\,'\mathit{should}'\,)$
$\mathit{sinc} = \mathrm{ptolemy/load\_proc}(\,'\mathit{sinc}'\,)$
$\mathit{init} = (\,\mathrm{proc}\ \ldots\ \mathrm{end}\,)$
$\mathit{Linear} = \mathrm{ptolemy/load\_proc}(\,'\mathit{Linear}'\,)$
$\mathit{ToRec} = \mathrm{ptolemy/load\_proc}(\,'\mathit{ToRec}'\,)$
$\mathit{Warp} = \mathrm{ptolemy/load\_proc}(\,'\mathit{Warp}'\,)$
$\mathit{MakeMappedMap} = \mathrm{ptolemy/load\_proc}(\,'\mathit{MakeMappedMap}'\,)$
$\mathit{CollocateRec} = \mathrm{ptolemy/load\_proc}(\,'\mathit{CollocateRec}'\,)$

$PlotDomBound = \mathrm{ptolemy/load\_proc}(\,'PlotDomBound'\,)$
])

```
> OldLoad := eval(`ptolemy/load`);
```

$$OldLoad := ptolemy/load$$

```
> `ptolemy/load` := proc(Index: name)
>   print(`Executing load`, Index);
>   OldLoad(Index)
> end;
```

$$ptolemy/load := \mathrm{proc}\ \ldots\ \mathrm{end}$$

```
> OldInit := eval(ptolemy[init]);
```

$$OldInit := \mathrm{proc}\ \ldots\ \mathrm{end}$$

```
> ptolemy[init] := proc()
>   print(`  Executing package initialization`);
>   OldInit()
> end;
```

$$ptolemy_{init} := \mathrm{proc}\ \ldots\ \mathrm{end}$$

```
> eval(ptolemy[LogRatioMap],1);
```

$$\mathrm{ptolemy/load\_proc}(\,'LogRatioMap'\,)$$

```
> eval(`ptolemy/types`);
```

$$ptolemy/types$$

```
> ptolemy[LogRatioMap](0,1,x,y);
```

$$Executing\ package\ initialization$$

$$\left[x \to \ln\left(\frac{x}{1-x}\right), y \to \frac{\mathrm{e}^y}{1+\mathrm{e}^y}, x \to (\,x, 1-x\,)\right]$$

```
> eval(`ptolemy/types`);
```

$$\mathrm{proc}\ \ldots\ \mathrm{end}$$

```
> eval(ptolemy[LogRatioMap],1);
```

$$ptolemy/LogRatioMap$$

```
> ptolemy[LogRatioMap](0,1,x,y);
```

$$\left[x \to \ln\left(\frac{x}{1-x}\right), y \to \frac{\mathrm{e}^y}{1+\mathrm{e}^y}, x \to (\,x, 1-x\,)\right]$$

```
> with(ptolemy, LogSinhMap);
```

$$Executing\ package\ initialization$$

$$[\,LogSinhMap\,]$$

```
> eval(LogSinhMap,1);
```

$$\mathrm{ptolemy/load\_proc}(\,'LogSinhMap'\,)$$

```
> LogSinhMap(0,LOW,x,y);
```

$$[x \to \ln(\sinh(x)), y \to \operatorname{arcsinh}(e^y), x \to (\sinh(x), \operatorname{sech}(x))]$$

```
> eval(LogSinhMap,1);
```

proc ... end

```
> LogSinhMap;
```

LogSinhMap

```
> eval(ptolemy[LogSinhMap], 1);
```

ptolemy/LogSinhMap

```
> ptolemy[LogSinhMap](0,LOW,x,y);
```

$$[x \to \ln(\sinh(x)), y \to \operatorname{arcsinh}(e^y), x \to (\sinh(x), \operatorname{sech}(x))]$$

```
> LogSinhMap(0,LOW,x,y);
```

$$[x \to \ln(\sinh(x)), y \to \operatorname{arcsinh}(e^y), x \to (\sinh(x), \operatorname{sech}(x))]$$

End of Maple Worksheet

init

This procedure loads the ***types*** LLF, defines the `&K` operator, and initializes the global variables `ptolemy/SimpProc` and `ptolemy/OneDRange` if they are not already defined.

Many Maple packages do not have an initialization procedure; instead each procedure in the package assumes responsibility for guaranteeing the existence of any definitions which it requires. **PTOLEMY** chooses, as a matter of design, to have an initialization procedure. The primary reasons are:

- Without an initialization procedure, **PTOLEMY** types can not be used to specify argument types as part of the parameter definition. Instead the procedure would have to first check that the requisite type had been defined; if not, it would then have to define the type, and then explicitly check the type of the parameter. **PTOLEMY**'s design allows procedures to assume that all **PTOLEMY** types have been defined prior to the procedure's invocation.

- Occasionally the user will need to assign global variables or package types before the first call to any of package's procedures. This might happen if the global variable were being set in order to pass information to a procedure or if the user wrote involved procedures to construct the problem elements.

- Finally, it is simply better programming practice to place all of the initialization operations together in one LLF than to distribute them throughout the package. This is especially true since making each procedure in charge of ensuring proper initialization would either involve duplicating much of the initialization code or would be equivalent to having a hidden initialization procedure.

In the current design of **PTOLEMY** the only declarations that are performed as part of the initialization procedure are the package types and the package's global variables. The package's procedures can be defined only as part of some loading operation, either using the `with` command, by evaluating a package table entry, or by explicitly executing a `readlib` command.

Explicitly loading an LLF with a `readlib` command would usually not be done by an interactive user, but it is sometimes desirable within a program. If part of the package is explicitly loaded via a `readlib`, then the user or programmer must ensure that package initialization has been performed before executing any of the package's procedures. Within any package procedure this is a nonissue, since all package procedures can assume that package initialization has occurred before they are executed. In a procedure that is not part of the package it may be necessary, in this case, to explicitly initialize the **PTOLEMY** package.

The **PTOLEMY** package initialization procedure can be explicitly executed at any time with the command

```
ptolemy[init]()
```

The initialization procedure is idempotent so reexecuting it does not cause any problems, but a **PTOLEMY** coding practice is to decide if package initialization has been performed by checking if `ptolemy/types` is of type *procedure*. Since the package initialization involves loading the **TYPES** LLF, this is usually a reliable indication of package initialization. Of course, confusion may occur if the user explicitly assigns `ptolemy/types` to a procedure or unassigns `ptolemy/types` after package initialization has occurred. Users should not redefine *any* package elements, unless they first consider the repercussions of such redefinition.

1.6 Overview of User-Level Modules

All of **PTOLEMY**'s exported procedures are listed and summarized in Table 1.3. This table groups these procedure according to the chapters in which they are described.

Using **PTOLEMY** to setup the solution to a PDE consists of successively converting the problem definition from one problem representations to the next in a sequence which eventually leads to a numerical matrix problem representation. **PTOLEMY** defines four primary problem representations:

1. A PDE defined over a collection of any subdomains that can be represented in **PTOLEMY**˙

2. A PDE defined over a collection of parallelepiped subdomains.

3. Representation of the collocation equations, over infinite domains, in terms of the tensor product of 1d basis components.

   The notation used for this representation is unique to **PTOLEMY** and is referred to as SB-notation. SB-notation is defined in [7] .

4. Representation of the problem as a matrix problem. Symbolically the matrix is represented as the product of a diagonal matrix and the Kronecker product row and column vectors and special matrices denoted in "I"-matrix format. This notation is common to much of the sinc-literature.

The procedure **init** is the package's initialization procedure. It loads all of the **PTOLEMY**-specific types and initializes the exported global variables.

The conversion between the general problem representation and a problem representation over parallelepiped domains primarily consists of applying a mapping the problem statement. This is equivalent to performing a change of variable. This step is typically performed with the procedure **ToRec** or **MultiToRec**. The procedure **Warp** allows the user to perform a change of variable on individual equations (or systems of equations). The user need not ever use **Warp** to setup the solution of a PDE but it is imagined that users would find many uses for this function, such as performing the operations of **ToRec** or **MultiToRec** by hand in order to customize the process.

The conversion from the representation over parallelepiped domains to the representation of the collocation equations in SB-notation involves the selection of bases, collocation of the equation using these bases, and remapping the problem onto the n-dimensional cartesian product of strips in the complex plane about the real line. This second mapping of the problem is called "the sinc-map." The sinc-map is designed to transform the region of analyticity in the complex plane. Because the number of collocation equations is potentially large, i.e., one per basis, it is important to use SB-notation to represent many equations with equivalent symbolic structures using a single symbolic result.

For linear problems the conversion from SB-notation to the matrix problem is largely a parsing problem. The procedure **Linear** checks to see if a problem, specified in SB-notation, is linear. The conversion to a matrix problem representation is done by either **LinToSKron** or **MultiLinToSKron**. The

Table 1.3: A summary of the exported procedures.

Warp	Maps a collection of equations from one domain to another.
MakeRecMap	Constructs a map from a rectangular domain to another rectangular domain.
MakeTradMap	Constructs a map from a traditional domain definition to a rectangular domain.
ToRec	Maps a problem defined over an arbitrary domain to a problem defined over a rectangular domain.
MultiToRec	Maps a problem defined over a collection of arbitrary domains to a problem defined over a collection of rectangular domains.
LogRatioMap LogTanMap LogSinhMap & IdentityMap	Constructs a ***MapInfoTypes*** for various sinc-maps.
CollocateRec	Collocates a problem defined over a single rectangular domain.
CollocateMultiRec	Collocates a problem defined over a collection of coupled rectangular domains.
Linear	Checks the collocated form of a problem to see if it is linear.
SToLinKron	Converts the collocated form of a problem to a matrix problem.
MultiSToLinKron	Converts the collocated form of a problem to a matrix problem.
SolveLinKron	Writes a matrix problem to disk and calls the external executable to solve it.
PlotBound	Plots a collection of parameterized boundaries.
PlotDomBound	Plot the boundary of a domain.
GridInvMap	Used to visualize a mapping in the original domain.
GridMap	Used to visualize a mapping in the mapped-to domain.
ClipPlot	Clip any two-dimentional Maple plot to rectangular region.
sinc	Defines the sinc function, its integral, and its derivative.

procedure **SolvLinKron** writes this matrix representation to disk and invokes a user defined "back-end" procedure to solving the problem.

The procedure **PlotBound** produces a Maple plot corresponding to a collection of boundaries. It assumes that all of the boundaries are of the same dimension. The procedure **PlotDomBound** will produce a plot of the boundaries of a collection of arbitrary domains; it relies on **PlotBound** for the actual plotting. The procedures **GridInvMap** and **GridMap** are used to visualize a mapping. The procedure **GridInvMap** creates a uniform grid on the mapped-to domain, uses the inverse map to map this back to the original domain, and plots the result. The procedure **GridMap** creates a uniform grid in the original domain, applies the map to this grid, and produces a plot of the result. The procedure **ClipPlot** will clip any two-dimentional Maple plot to an arbitrary rectangle. Finally, the procedure **sinc** defines the sinc-function which is useful for plotting sums of sinc bases, but is not directly used by any other part of the package.

1.7 Installing Ptolemy

Because **PTOLEMY** is not part of the standard Maple distribution, at the beginning of each session the user must instruct Maple where to find **PTOLEMY** and how to load it. This is easily done by adding one or two commands to the user's Maple initialization file; suggested commands are presented in the next subsection.

Instructing Maple where to find **PTOLEMY** is easily done by setting the variable `libname` to include the path in which the package resides. Directing Maple to dynamically load the package whenever it is first referenced is easily done by assigning the global variable `ptolemy` to a `readlib` command which will load the package whenever the global variable is fully evaluated.

The Libname Variable

The global variable `libname` specifies an ordered sequence of path names which Maple searches when loading library files with the `readlib` command. In general this variable must be modified in order to load and use **PTOLEMY**, but since this variable is also consulted when loading system libraries the user will generally not want to simply assign `libname` to the path name where **PTOLEMY** resides. Instead the path name where **PTOLEMY** resides should added to the front or back of the sequence already assigned to `libname`, using a command like

```
libname := libname, `/new/path/name`.
```

Because most of the `readlib` commands will operate on a name of the form `ptolemy/lib_file_name` where `lib_file_name` is one the LLF names, the path added to the `libname` sequence should actually be the path containing a directory named **PTOLEMY** which contains most of the dot-'m' files.

For example, if libname points to

```
/usr/local/share/contrib/maple
```

then the **PTOLEMY** package table is setup to look for the LLFs in the directory

```
/usr/local/share/contrib/maple/ptolemy
```

Similarly, if the variable `libname` points to `/home/grad/jdoe` then most of the LLFs must be stored in the directory `/home/grad/jdoe/ptolemy`.

Once the `libname` variable is defined so that `readlib`'s can load **PTOLEMY**'s library files, the global variable `ptolemy` should be assigned the expression

```
'readlib('ptolemy')'
```

This will load the LLF which defines the package table, whenever the global variable `ptolemy` is fully evaluated. The result of this `readlib` is the global variable `ptolemy` defined in the LLF, so the result of evaluating the variable `ptolemy` (the first time) is the package table defined in the LLF. A side effect of this `readlib` is to redefine the global variable `ptolemy` reference be the package table. This means that subsequent evaluations of the global variable `ptolemy` will not execute the `readlib`; instead they will merely evaluate the result of the first `readlib`.

The intent is that users of the **PTOLEMY** package should add two lines similar to

```
libname := libname, PtolemyDir:
ptolemy := 'readlib('ptolemy')':
```

in their **.MAPLEINIT** file. If the user already has a **.MAPLEINIT** file the incremental cost upon startup of these two lines is negligible. If the user does not already have a **.MAPLEINIT** file the incremental cost is slight, but may be detectable on some systems because one extra disk access will be added to the Maple startup sequence.

On the University of Utah College of Engineering facilities, **PTOLEMY** is already installed in the path

```
/usr/local/share/contrib/maple/ptolemy.
```

So users on these systems should add the

```
libname := libname, `/usr/local/share/contrib/maple`:
ptolmey := 'readlib('ptolemy')':
```

to their **.MAPLEINIT** files.

Downloading Ptolemy

On systems where **PTOLEMY** is not centrally installed users may easily install it in their own accounts. GNU-zipped `tar` files can be downloaded from the World Wide Web at

```
http://daisy.uwaterloo.ca/~kparker/ptolemy
```

Downloadable files exist both with and without the Postscript version of this manual (the version without the manual is much smaller).

To "unzip" the file use the Unix shell command

```
gunzip ptolemy.tar.gz
```

To "untar" the file use the Unix command

```
tar xf ptolemy.tar
```

This will unpack all of the library files into the subdirectory named **PTOLEMY** of the current directory.

If the library files are "untarred" into the path named `$(HOME)/lib/ptolemy` then file named **DOT_MAPLEINIT**, included in the distribution, will suffice as a **.MAPLEINIT** file. This file will add the pathname `$(HOME)/lib` to the `libname` sequence. This dot file is especially convenient for first time Unix users, but in order to get the value of this Unix environment variable it is necessary to spawn a subprocess, so this file is significantly slower than if the path names are hard-coded into the initialization file. It is suggested that users consider other long term solutions, such as hard-coding the path names or using `m4` to insert the path name into the Maple initialization file. Users who do not already have Maple initialization files may simply copy this file to their home directory (with the appropriate name change). The Unix command for doing this is:

```
cp ptolmey/dot_mapleinit $(HOME)/.mapleinit
```

Chapter 2

Type Declarations

PTOLEMY creates quite a complicated taxonomy of types. A working knowledge of these types is probably the primary component in a less superficial understanding of how to use **PTOLEMY**, but simply cataloging the **PTOLEMY** types, as is done in the bulk of this chapter, may not be sufficiently motivating. The reader who has sufficient conceptual understanding of **PTOLEMY** will benefit from a deeper contemplation of the material in this chapter. The reader who lacks a mental model of the way **PTOLEMY** works may be better advised to regard this section more as reference material, deferring involved consideration until after mastering the material in Chapter 3 and Chapter 4.

2.1 Different Typing Concepts

Maple's concept of typing is sufficiently different from the concept employed by more traditional programming languages that it is useful to discuss typing concepts in the abstract, before enumerating the Maple types defined by **PTOLEMY**.

Structural Types

Maple supports only 37 elementary types. These elementary types are directly implemented by the kernel. All Maple objects are created by nesting this small set of elementary types. It is easy to extend this concept of type to apply to any Maple object, by defining the type of any expression tree to be the "type tree" whose nodes are the type of corresponding nodes in the expression tree.

For example the type of the Maple object

```
[1,'LOW','V' = 0]                    (2.1)
```

is shown graphically in Figure 2.1. Such type-trees can also be represented typographically. For example, this type would usually be denoted as

```
[INTPOS,NAME,EQUATION(NAME,INTPOS)]
```

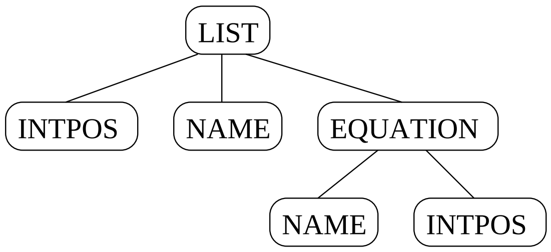

Figure 2.1: The Structural Type of the Maple Expression in Equation 2.1

Notice that Maple has two different elementary types for representing integers, one for "positive" integers and one for "negative" integers. The words positive and negative must be interpreted loosely since zero can be represented using either of these elementary types. However, the parser will create an object of type *INTPOS* when it encounters a zero.

Though in computer science this way of defining the type of a complicated object is the most common, it is not the only reasonable way of doing it. This type concept is sometimes called *structural typing*. The major point of this section is to contrast this concept of type with other typing concepts, specifically *logical typing* and *attributed types*.

Logical Types

In Maple, structural types are often overly narrow. For example in many applications the programmer does not wish to distinguish between positive and negative integers. It is easy to imagine an application in which the program would accept any number, but in Maple the structural type of a number can be an *INTPOS*, an *INTNEG*, a *RATIONAL*, or a *FLOAT*. Checking all of the possible structural types in this case would not be prohibitive, but if this much work is required for such a simple example it should be clear that for complicated objects simply enumerating all of the admissible structural types could become prohibitive.

To remedy this problem Maple uses a completely different typing concept, *logical typing*, defined by a test that can be applied to an object. The set of objects which pass the test are of the prescribed logical type. In the previous example the programmer could use the system-defined logical type ***numeric*** which is defined as the test of whether the object is of the logical type ***integer***, the structural type *RATIONAL*, or the structural type *FLOAT*, where the logical type ***integer*** is the test to see if the object is of the structural type *INTPOS* or *INTNEG*.

The Maple command `type` checks the logical type of a Maple object. Vary rarely can this be directly related to the structural type of the object. For example Maple defines logical types named ***posint*** and ***negint***, but neither one of these can be used to determine if the structural type of the object is *INT*-

POS or *INTNEG*, since zero is excluded from both type ***posint*** and type ***negint***, but can in theory be represented with either the structural type *POSINT* or *NEGINT*. Determining the structural type of a Maple object typically requires the use of the `disassemble` command. In fact, to indicate the specialness of structure types a previous version of the manual, i.e., [2], used uppercase names for structural types; the **PTOLEMY** manual has adopted this convention.

The primary limitation of the logical type concept is that the logical type is *not* an attribute of the object; it is merely a test which can be applied to an object. The important implication of this is that it is easy to ask,"Is this object of a particular logical type?", but it is not possible to ask, "What is the logical type of this object?" In most cases this question is not even meaningful. Given a set of logical types an object might not be any of these logical types or it might be many of them. Even if the object is only one of the logical types and the set of possible logical types is enumerated, the only general way to answer this question is to test the object against each of the logical types. Typically this would be too inefficient to be of much practical interest.

Attributed Types

A key point about structural types is that because they directly reflect the structure of the implementation, they are a unique immutable property of the object. In contrast the logical type is a computational result which is considered to be a type for the purpose of simplifying the code.

Sometimes is would be useful to associate a specific logical type with an object. This manual will refer to this typing concept as the *attributed type*. An attributed type is the logical type of the object that was intended when the object was created.

An example of the need for this typing concept is when fairly diverse logical types may be manipulated by a single procedure, but one of the required methods for manipulating these object varies among the collection of logical types. In more complicated cases it may be difficult to figure out which method should be used without knowing which logical type was the intended when the object was created.

Maple provides no support for the concept of attributed types, so the programmer who wishes to use this concept must explicitly tag the object with an attributed type indicator. **PTOLEMY** does not add attributed type fields to any of its type definitions, but in some cases, like the *DomainType*s, **PTOLEMY**'s design would probably be improved by doing so. Clearly, this is the high level concept being invoked.

2.2 Records in Maple

Maple does not provide any direct support for "record" types. However, there are three common ways of emulating records: 1) using tables, 2) using inert functions, and 3) using lists. Each method has some advantages and some disadvantages. At this time **PTOLEMY** uses lists for all of its record types, although the result is sometimes not intuitive. To help make **PTOLEMY**'s type definitions a little less confusing it is probably helpful to first consider the common methods for implementing records in Maple.

Records Implemented with Tables

Semantically and syntactically the best way to emulate records in Maple is by using tables. Each index can correspond to the name of a field and each field value can be stored in the table entry, indexed by the field name. Runtime checking of field names can be enforced by using an "indexing function" (the keyword `indexfcn` in Maple's on-line help system has more information on indexing functions). The problem with this approach is that Maple's implementation of tables is based on hashing. Hashing is perhaps inherently too inefficient for implementing large numbers of small records no matter how it is done, but Maple's implementation of tables uses fixed size hashing tables which are particularly poorly optimized for this particular operation.

To illustrate this consider the following example:

Start of Maple Worksheet

Create a record with two fields named 'x' and 'y'. This could be an implementation of a two-dimensional point type.

```
> `index/PointType` := proc(Field,Table)
>   if (op(Field) <> 'X') and (op(Field) <> 'Y') then
>     ERROR(`Invalid field name for record PointType`)
>   fi;
>
>   if nargs = 2 then
>     Table[op(Field)]
>   else
>     Table[op(Field)] := args[3..nargs]
>   fi;
> end:
```

```
> A := table(PointType);
```

$$A := \mathrm{table}(\mathit{PointType}, [\\ ])$$

```
> A['X'] := 1;
```

$$A_X := 1$$

```
> A['Y'] := 0;
```

$$A_Y := 0$$

```
> A['Z'] := 3;
Error, (in index/PointType) Invalid field name for record PointType
```

```
> op(op(A));
```

$$PointType, [X = 1, Y = 0]$$

The first element (i.e., the 5) indicates that the object is of type `NAME`. This is because references to tables do not fully evaluate so this command is examining the internals of the variable `A`. The second element is a pointer to the object assigned to the variable. The third element encodes the character string for the variable name.

```
> Var := [disassemble(addressof(A))];
```

$$Var := [5, 1074137236, 1090519040]$$

The first element (i.e., the 28) indicates that the object is of type `TABLE`. The second element is a pointer to the indexing function. The third element is a pointer to the array bounds information. The fact that this is a table and not an array is indicated by pointing to the symbol `false`. The final element points to the table.

```
> Table := [disassemble(Var[2])];
```

$$Table := [28, 1074114476, 1074092944, 1074141660]$$

```
> disassemble(Table[2]);
```

$$5, 0, 1349478766, 1951693168, 1694498816$$

```
> whattype(pointto(Table[2]));
```

$$string$$

```
> pointto(Table[2]);
```

$$PointType$$

```
> disassemble(Table[3]);
```

$$5, 2, 1717660787, 1694498816$$

```
> whattype(pointto(Table[3]));
```

$$string$$

```
> pointto(Table[3]);
```

$$false$$

The first element (i.e., the 37) indicates that the object is of type `HASHTAB`. All but two of the entries do not point to a hashing bucket. The two nonzero entries point to the two hashing buckets associated with this hash table.

```
> HashTab := [disassemble(Table[4])];
```

$$\begin{aligned}HashTab := [&37, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,\\
&0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,\\
&0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,\\
&0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,\\
&0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1074109532, 0,\\
&0, 0, 1074110008, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,\\
&0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,\\
&0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,\end{aligned}$$

```
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

> nops(HashTab);
                                  257

> NonZero := NULL:
> for i to nops(HashTab) do
>   if HashTab[i] <> 0 then NonZero := NonZero,i fi
> od;
> NonZero;
                                1, 130, 134
```

The first element of each hash bucket (i.e., the 36's) indicates that each of these objects is of type HASH. The rest of the elements in each structure are paired so that each pair defines a table entry. The first element in each pair points to the index, and the second element points to the table entry.

```
> Bucket1 := [ disassemble(HashTab[NonZero[2]]) ];
          Bucket1 := [36, 1074115584, 1074063288, 0, 0, 0, 0]

> Bucket2 := [ disassemble(HashTab[NonZero[3]]) ];
          Bucket2 := [36, 1074115620, 1074063272, 0, 0, 0, 0]

> Index1 := disassemble(Bucket1[2]);
                     Index1 := 5, 0, 1476395008

> pointto(Bucket1[2]);
                                  X

> Value1 := disassemble(Bucket1[3]);
                            Value1 := 2, 1

> Index2 := disassemble(Bucket2[2]);
                     Index2 := 5, 0, 1493172224

> pointto(Bucket2[2]);
                                  Y

> Value2 := disassemble(Bucket2[3]);
                            Value2 := 2, 0
```

End of Maple Worksheet

Ignoring the Maple overhead of the indexing function, which is not duplicated for each instance of this record, this representation of the record takes 285 words of memory (i.e., 1140 bytes on a 32-bit machine). An efficient representation in Maple should reasonably take just seven words of memory (28 bytes). By comparison the same record would probably be represented in C++ in 2 words (8 bytes).

The memory overhead of implementing records with this mechanism are acceptable only for unusually large records.

Records Implemented with Inert Functions

Within the Maple, library records are represented using inert functions. For example the record which contains information about plots is implemented using the inert function named **PLOT**. The record's fields are stored as the arguments of the inert function. The field names are indicated by the use of additional functions, and the field values are the arguments of these inert arguments.

This method actually has the widest range of advantages. The primary advantage of this approach is that a special printing procedure can be defined for all objects of the prescribed record type (see the keyword `print` in Maple's on-line help system). This is in fact done for the `PLOT` record type. Another advantage of this approach is that the intrinsic function name can be used to define an attributed type. In addition the memory overhead of this method, although still a problem when implementing very small records, is acceptable for most records.

The following session reveals the internal structure of Maple's `PLOT` "record type." Notice the use of seven different inert functions in this example alone.

Start of Maple Worksheet

```
> Temp := plot([[0,0], [2,1/2], [1,1], [0,0]],
>   scaling=constrained, tickmarks=[5,3]);
```

$$\begin{aligned}&\mathit{Temp} := \mathrm{PLOT}(\mathrm{CURVES}([[0,0],[2.,.5000000000],[1.,1.],[0,0]],\\&\quad \mathrm{COLOUR}(\mathit{RGB},0,0,0)),\mathrm{AXESTICKS}(5,3),\mathrm{AXESLABELS}(\,,\,),\\&\quad \mathrm{TITLE}(\,),\mathrm{SCALING}(\mathit{CONSTRAINED}),\\&\quad \mathrm{VIEW}(\mathit{DEFAULT},\mathit{DEFAULT}))\end{aligned}$$

```
> print(`print/PLOT`);

proc(pin) ... end


> Temp;
```

See Figure 2.2

End of Maple Worksheet

The primary disadvantages of this approach is that manipulating field values within the record is exceedingly painful and mildly inefficient.

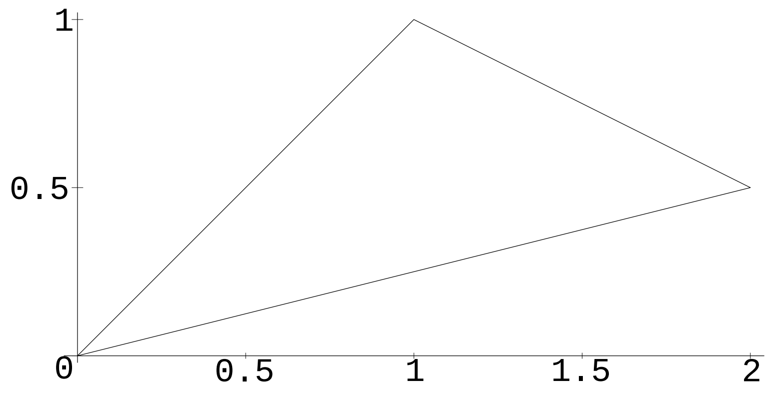

Figure 2.2: A Maple Plot of a Triangle

Records Implemented with Lists

PTOLEMY uses lists to implement records. The order of the fields is fixed and the list of field values is simply stored as a list. This is the way of implementing records in Maple which most nearly mimics the way in which records are *implemented* in more traditional languages. An immediate consequence, is that this implementation choice is also the most efficient.

However, implementing records with lists result in a Maple object which is less recognizable as a record then the results of using either tables or intrinsic functions. A serious manifestation of this problem is that the field names are never explicitly used. The field name is implicitly determined by the position of the field value within a sequence of all field values.

For example if the *PointType* illustrated earlier is this section were implemented as a list, the 'Y' field value of a record named `Point` would be referenced as `Point[2]`. Referencing the field as `Point[Y]` would be much more readable; however, referencing the field if the record where implemented in a manor analogous to the `PLOT` record would require the much more complicated, less efficient, and even less readable expression

```
subs(map(T -> op(0,T) = op(T), Point), Y)
```

A minor disadvantage of using lists to implement records is that none of the fields are optional. With both the table implementation and the intrinsic function implementation it is easy to leave some of the field values unspecified. Of course, there are ways of extending the list implementation of records to include optional fields, but the result is more cumbersome. This is not a serious disadvantage since record types with optional fields are rare and "optimizing for the common case" is a good design principle.

2.3 Types Checking versus Data Checking

Because logical types are simply tests that can be applied to an object, it is possible to define logical types such that all of the data checking is performed as part of the type checking. In some cases this is even appropriate, but this is poor design strategy.

In cases when the cost of checking the validity of the data is significant compared to the cost of the computation, the program should check the validity of the data only once. It is also often desirable to check the type of all arguments passed to all procedures in order to alert the developer of minor errors. In many cases this will result in the type of some objects being checked more than once. This inefficiency can be tolerated only if the cost of the type checking is small compared to the cost of the computation.

Consequently, the desirable design strategy is to separate data checking from type checking except when the two operations are equivalent. For logical types it is not always clear which checks are part of validating the type and which are part of validating the data. Clearly, checks of the object's structural type are part of the type validation and not part of data validation. It would be reasonable to argue that all other checking is data validation. This point of view effectively defines logical types as a computational method of defining ensembles of structural types.

It is my opinion that this point of view is too restrictive. It would disallow Maple's logical type, *POSINT*, since checking for nonzeroness is really data checking. Yet, much of the power of logical typing is that the programmer can separate the logical type definition from the implementation, and defining a type such as ***posint*** (or ***nonzeroinit***) is in some cases a reasonable use of logical types.

PTOLEMY attempts to use the following rules to decide which checks should be included in the type definition:

- Checking the structural conformity of the object should always be part of the type definition.
- Checking the structural consistency of the object should also be included as part of the type definition.
- Checks which take a significant amount of time to perform, compared to checking the structural validity, are part of data validation, not part of the type definition.
- Types should have general meaning beyond the specific purposes of the immediate application. Any checks that do not appear to have such generality are part of data validation.

Maple provides a powerful type expression syntax for defining structural conformity. Almost all **PTOLEMY** types first compare the object against this type expression specification. In some cases this is sufficient, but in a significant portion of cases, verifying the structural consistency of the object requires additional computation. In fact, Maple lacks any particularly good technique for performing checks such as, "The size of each list in a collection of lists must be the same." Only a few **PTOLEMY** types perform additional checks beyond these two classes of structural checks.

Miscellaneous Types

nonzeroint

This type checks that the object is an integer and that it is not zero.

EndType

This is an enumeration type. Variables of this type must be assigned the values `'LOW'` or `'HIGH'`. Typically, variables of type ***EndType*** specify which end of a range is being referenced.

BoundType

This type is used to specify a parameterized boundary.

Assume that the boundary is defined in an n-dimensional space. Then the associated ***BoundType*** must be a list with $2n - 1$ elements. The first n-elements must be of type ***algebraic*** and express functions for the coordinate values. The last $n - 1$ elements must be of type ***name=range*** and define the parameters of the parameterization.

For example:

```
[x*cos(Pi*x), x^2*y, x*sin(Pi*x), x = 0..2, y = -1/2..1/2];
```

describes the boundary illustrated in Figure 2.3.

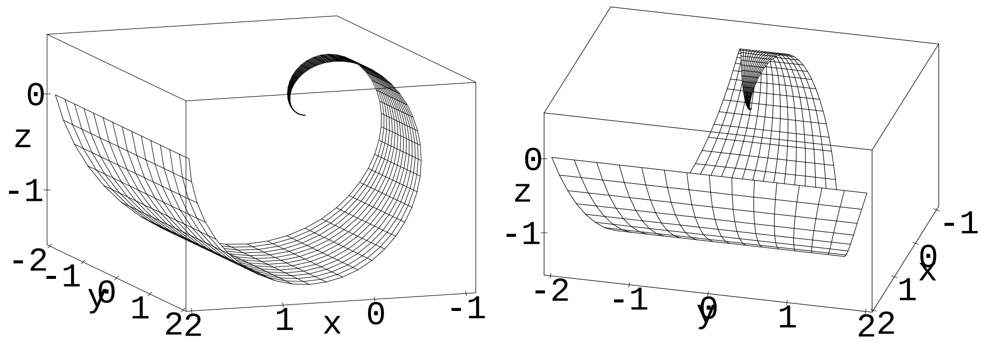

Figure 2.3: A Segment of a Three-Dimensional Cylindrical Spiral

In the xz-plane the boundary curls through the range of angles from 0 to 2π radians. In the y-dimension the boundary thickens from a point when the angle is 0 units to a line segment of width 4 units when the angle is 2π.

MapInfoType

The type *MapInfoType* defines the information about a sinc-map typically required by **Ptolemy**.

The type equation for this type is:

$$\{\texttt{[procedure, procedure], [procedure, procedure, procedure]}\}$$

The first element of the list defines the map, the second element defines the inverse map. If the list has three elements then the third element defines the weight, also called the nullifier function, associated with this map. If the map is denoted by ϕ then the weight is defined to be $1/\phi'$.

If a factorization of the weighting function is known, the third procedure will return a sequence of two results, one for the factor corresponding to the `LOW` end of the interval and another for the factor corresponding to the `HIGH` end. When the factorization of the nullifier is not known the procedure will return just one result specifying the unfactored weight. The LLF **`MAP_INFO_OPS`** contains procedures for using the information in objects of type *map_info*.

Collection Types

The types documented in this subsection are used to describe collections of homogeneous Maple objects. These types are similar to Maple's *list* and *set* types, but less specific about the kind of relationship imposed among the various objects in the collection.

Maple's built in collection types, i.e., *list*/*set*, enforce two interobject relationships: uniqueness and ordering. It would be better if Maple offered greater orthogonality of these properties; for example by supporting "ordered sets" and "unordered lists." In addition to these two obvious new collection types other less obvious collection types could be created based on other interobject relationships. Probably the only other fully general interobject relationships that could be imposed are graph based relationships. However, discovering compelling applications for data dependent relationships, such as partial orders or less trivial mutual exclusion rules, is much easier.

This section does not define new collection types, but merely provides types for abstracting the collection concept for whatever collection constructs are supported elsewhere. Currently these collections only support *set*s and *list*s, but the next version of **PTOLEMY** will probably add support for "unordered lists" and "ordered sets" and make it easier for users to integrate their own collection types with the package.

The need for abstracting the "collection of objects" concept for various other types is motivated by the fact that procedures perform an operation on each object in a collection without caring about the interobject relationship. For example Maple's `map` command will perform some operation on each element of a list, a set, or an expression tree. The result is the same type as the input, but each element or node is the result of applying the prescribed operation to the corresponding input element or node.

However, `map` functional syntax is often difficult to use. The collection types allow procedures to specify that some of its arguments will be treated in a `map`-like fashion.

Each collection type described in this section is actually a parameterized type; that is, it is actually a type function requiring an argument that specifies a subtype and return a fully instantiated type. It is only in the sense that all of the objects in each collection must be of the specified (logical) subtype that these collections are homogeneous. That is, each of the objects in the collection must be instances of the specified logical type, but may otherwise be quite dissimilar.

Collection

Objects of this type may be either sets or lists of the specified subtype. That is, the type expression for the type *Collection(SubType)* is:

```
{set(SubType), list(SubType)}
```

It would be trivial to make the subtype argument optional. In effect, the type *Collection* could be defined to be equivalent to *Collection(anything)*. This is not done for consistency with the *collection* type, where the use of the type *collection(anything)* is strongly discouraged. The user who wishes to omit the subtype should instead use the definition *Collection(anything)*.

collection

This type function is similar to *Collection* except that it also allows an isolated object of the specified subtype. The type expression for the type *collection(SubType)* is:

```
{SubType, set(SubType), list(SubType)}
```

The *collection* structure is often more convenient for interactive users than *Collection*, because a single instence of the item need not be enclosed in a *list* or *set*. However, it is easier to write code to manipulate elements of a *Collection*s because the structural type will always be a *SET* or a *LIST*.

Also, using *collection* can result in ambiguity when instances of the subtype might, or might not be, *list*s or a *set*s. In such cases it may not be possible to distinguish an isolated object that is a *set* or *list* from a *set* or *list* of objects each of which is of the specified subtype. Even when this ambiguity can be resolved, the code for doing so is generally less efficient and often more complicated. In addition, almost all code for manipulating *collection*s is more complicated than the analogous code for manipulating *Collection*s, because the meaning of the `map` command is different when applied to a *list* or *set* of objects than when applied to an isolated object. The result is that an extra conditional is often required in order to handle the case of an isolated object differently than *list*s or *set*s of the object.

Since all objects are trivially of type *collection(anything)*, simply by virtue of being of type *anything*, use of the type *collection(anything)* is strongly discouraged. Usually in cases where the the *collection(anything)* seems appropriate what is really needed is the type *collection(NotListOrSet)*, where the type *NotListOrSet* is defined to be of any type except for a *list* or a *set*.

CollectStruct

This type function defines a type that is a specified number of nesting of the type *Collection*. Unlike the other "types" defined in this section, this type function has two arguments, i.e., a subtype specification and a depth of nesting specification.

The type expression for *CollectStruct(SubType,Depth)* is

```
Collection(...Collection(SubType)...)
```

where there are exactly `Depth`, `Collection` functions in the type expression.

collectStruct

This type function defines a type that is an arbitrary nesting of the type *collection*. The type expression for the *collectStruct(SubType)* is,

```
{SubType, Collection(collectStruct(SubType))}
```

Another way of defining this type is that each subtree in the the object's expression tree must be of either type *SubType* or *Collection(SubType*.

Example Usage

Start of Maple Worksheet

```
> ptolemy[init]();
```

```
> A := [1, 2, 3];
```

$$A := [1,2,3]$$

```
> type(A,collection(posint)); type(A,Collection(posint));
```

true

true

```
> B := {3,2,1};
```

$$B := \{2,3,1\}$$

```
> type(B,collection(posint)); type(B,Collection(posint));
```

true

true

```
> C := 3;
```

$$C := 3$$

```
> type(C, collection(posint)); type(C, Collection(posint));
```

true

false

```
> NotA := [1,2,[3,4]];
```

$$NotA := [1,2,[3,4]]$$

```
> type(NotA,collection(posint));
```

false

```
> A := [[1, 0], [[3, 1], [4, 0]], {[2, 2], [5, -2]}];
```

$$A := [[1,0],[[3,1],[4,0]],\{[2,2],[5,-2]\}]$$

```
> type(A, collectStruct([posint,integer]));
```

true

```
> type(A, CollectStruct([posint,integer],2));
```

false

```
> B := [{[1,0], [2,3]}, [[3,-1]], {[4,1], [5,-2], [6,0]}];
```

$$B := [\{[1,0],[2,3]\},[[3,-1]],\{[5,-2],[4,1],[6,0]\}]$$

```
> type(B, CollectStruct([posint,integer],2));
```

true

End of Maple Worksheet

Order Types

The types documented in this subsection are used to describe the order (with respect to differentiation) of an approximation. This is important both in problem specification, for expressing the required order of approximation of the solution, and in solution reporting, for reporting the order of approximation achieved.

The order of an approximation is defined to be the order of the highest derivative "accurately" approximated. So a zeroth order approximation, accurately approximates the function, but not any of the function's derivatives. Similarly, if $\tilde{f}$ is a first order approximation of f, then not only does $\tilde{f}(x) \approx f(x)$ (for all x in the region of approximation) but $\partial\tilde{f}(x)/\partial x \approx \partial f(x)/\partial x$.

SubOrderType

This type specifies the order of approximation for a single-dimensional approximation. The type expression for this type is

```
{nonnegint, [nonnegint,nonnegint]}
```

If an object of this type is a nonnegative integer then it indicates the order of approximation. On the other hand, if an object of this type is a list of two nonnegative integers it indicates the order of the approximation at the low end and high end (respectively) of the implicitly defined interval of approximation. In this case the order of approximation in the interior of the region is at least the minimum order at the ends of the interval, but is otherwise unspecified by this data structure.

For sinc methods the order of approximation becomes very large at the point in the domain which maps to zero. In fact in the limit as N goes to infinity the order of approximation becomes infinite at this point, but the order of the approximation will decrease towards the ends of the interval of approximation to no more than that guaranteed by the suborder type.

OrderType

This type indicates the order of approximation for a parallelepiped region. The type expression for this type is

```
list(SubOrderType)
```

Each element of the list indicates the order of approximation in the corresponding coordinate direction. The number of elements in the list implicitly defines the dimensionality of the region of approximation.

Assume that the following Maple fragment

```
Spec1 := [[1,2], 0];
```

indicates the order of an approximation of f over the domain $[\ell_1, h_1] \times [\ell_2, h_2]$ and that the coordinate names are denoted by x_1 and x_2. Then f and $\partial f/\partial x_1$ are accurately approximated over the entire domain. It also indicates that the $\partial^2 f/\partial x_2^2$ is accurately approximated along the upper boundary of the domain (i.e., where $x_1 = h_1$), but is not accurately approximated near the lower boundary of the domain (i.e., where $x_1 = \ell_1$). Finally, it indicates that the $\partial f/\partial x_2$ is not guaranteed anywhere in the domain.

OrderSpecType

This type serves the same purpose as the type *OrderType*, except that it associates a state-variable with the order information. The type expression for this type is

```
[name, OrderType]
```

The Maple fragment

```
[[V, [1,1]], [P, [0,0]]];
```

defines two *OrderSpecTypes*. They indicate that V, and it is first partial derivate are accurately approximated over the entire domain, but that only the function value of P is accurately approximated over the domain. That is, none of the partial derivatives of P are guaranteed to be accurately approximated anywhere in the domain.

MultiOrderSpecType

This type defines order information over a collection of parallelepiped domains and associates all of the order specifications with a single state-variable. This is a preferable representation to a list of *OrderSpecType*'s, because the equivalence of state-variables across domains is unambiguous.

The type expression for this type is:

```
[name, list(OrderType)]
```

The **multi_spec_to_list** and the **spec_list_to_multi** procedures in the **ORDER_OPS** LLF can be used to convert between a list of *MultiOrderSpecTypes* and a list of lists of *OrderSpecTypes*. This essentially converts order specification information between a representation for each individual domain in the collection and one representation for the whole collection of domains. The procedure **spec_list_to_multi** also provides data checking to ensure that the state-variable names are consistent across all the domains.

2.7 Domain Types

Domain types provide a *geometric* description of a problem domain or of a collection of problem subdomain.

Unlike most other types defined by **PTOLEMY** domain types are actually Automatic Data Types (ADTs). See [1] for an authoritative introduction to ADTs. The domain type must not only define a data structure describing the domain geometry but it must also define two specific operations for that data structure. In the language of Object Oriented Programing (OOP) a domain type definition is not complete unless it also defines two associated methods. The required methods for a domain types construct a smooth map and its inverse from the domain to an arbitrary finite parallelepiped.

It is easy to create data structures that unambiguously define many kinds of geometric regions, but a significant investment is required to construct classes of geometric regions that can be easily mapped to a parallelepiped and for which the inverse of the resulting maps can be symbolically expressed. This is the reason that, at the moment, **PTOLEMY** support such a limited set of domain types.

At the moment, all supported domain types are of arbitrary (but fixed) dimensionality. *When the mathematics allows nearly trivial, Maple makes it easy to implement such generality*, but there is nothing in **PTOLEMY** that requires domain types to have this generality. All that is required is that the domain representation have the right dimensionality for the problem definition in which it is used.

Because Maple does not directly support ADTs or true objects, **PTOLEMY** uses a naming convention to associate the methods with the type. Domain types are given names of the form *ThisDomainType* where `This` is a distinguishing mnemonic for the specific domain type. Currently **PTOLEMY** defines three domain types, named *RecDomainType*, *TradDomainType*, and *MappedDomainType*.

Restrictions on the Map

The maps associated with each domain type should satisfy the following properties:

1. Both the map and its inverse must be representable by a finite symbolic expression, which Maple can differentiate.
2. The map from the domain to the parallelepiped must be a bijection.
3. The map should be "smooth."

A paragraph is required to rigorously explain what is meant by smooth in this context. First notice that the except for one-dimensional problems mapping is a *transformation* from $\mathbb{C}^n$ to $\mathbb{C}^n$, not a function. The desired smoothness property is that each coordinate function of the transformation be conformal with respect to each coordinate at every point on the interior of the domain. That is, if the mapping is

$$M(x_1, \ldots, x_n) = \big(f_1(x_1, \ldots, x_n), \ldots f_n(x_1, \ldots, x_n)\big)$$

then

1. For all $j, k \in \{1, \dots n\}$ the $\partial f_j / \partial x_k$ exists at every point, $(x_1, x_2, \dots, x_n)$ on the interior of the domain.

2. For all $j, k \in \{1, \dots n\}$ the $\partial f_j / \partial x_k \neq 0$ at any point on the interior of the domain.

3. The map from the domain to the parallelepiped is a *bi-junction*.

This smoothness requirement is confusing and sometimes hard to test. Fortunately, it is rather natural for the map to satisfy this requirement if the map satisfies the other two requirements. The most practical implications of this smoothness requirement is that any sharp corners in the boundary of the domain must map to the corners of the parallelepiped and that any sharp edges in the boundary of the domain must map to the edges of the parallelepiped.

The more troubling difficulty is constructing maps with closed form inverses. This difficulty is primarily what has prevented the development of more domain types.

PTOLEMY is designed to work with maps that satisfy these requirements, but only the first two requirements are necessary. Researchers intimately familiar with sinc-methods may, on occasion, wish to use maps that do not satisfy the smoothness property. The resulting sinc approximation will typically still converge, but the error of approximation will not decrease exponentially. In some cases an "almost smooth map" may actually give slightly better performance for "small N" (i.e., a small number of sinc points).

In addition it is possible for the researcher who is willing to either modification **PTOLEMY** or who can employ special insight into the current problem to relax the first requirement. One way of doing this would be to use a finite symbolic approximation of either the map or its inverse. The adequacy of the symbolic approximation would either have to be guaranteed by the user or **PTOLEMY** would have to select the appropriate truncation of an infinite symbolic form.

Maps that do not satisfy all three of these requirements are expected to be well outside the range of normal usage.

Multidomain Types vs Domain Types

When several subdomains are defined over the same space it would be possible to allow for the use of different coordinate names in each subdomain. However, much of **PTOLEMY**'s code is simplified if the same coordinate names are used in all of the subdomains within a particular problem. Because of this many of **PTOLEMY**'s procedures require that the coordinate names be the same across all of the subdomains within a particular problem.

If a list of *DomainType*s were used to specify a list of subdomains then user errors would arise because of failure to satisfy this requirement. In addition to check for these errors a significant amount of extra code would need to be distributed throughout the package. To circumvent these problems **PTOLEMY** defines "multidomain" types that allow a list of domains types to share a single specification of the coordinate names. If the domain type name is *ThisDomain-Name*, then the corresponding multidomain type is be named *MultiThisDomain-Type*.

Because each of the domain types currently supported by **PTOLEMY** involves lengthy procedures for checking the validity of the representation, **PTOLEMY** attempts to use one procedure for validating both the components of a multidomain type and the corresponding domain types. In order to facilitate this **PTOLEMY** separates the purely structural part of a domain type definition from all of the other computations needed to check the type.

For each domain type **PTOLEMY** defines a structural type that embodied the structural requirements of the type, except for the field that defines the coordinate names, and defines a procedure for checking the validity of this type. The name of the both the type and validation procedure can be derived from the name of the domain type. If the domain type name is *ThisDomainType* then the name of the data checking procedure will be **ptolemy/check_this_domain**, and the name of the type that captures the structural essence of the domain type will be *ThisDomainStruct*. This name is intended to emphasize the distinction between a purely structural type and a logical type (which in Maple is the standard kind of type).

The logical domain type must: 1) include the specification of the coordinate names, 2) include fields needed to construct the associated structural type, and 3) contain information that is accepted by the associated validation procedure.

All of this may sound like a contradiction of the idea espoused in Section 2.3 that "data checking" should not be part of the logical type definition. The data validation included as part of the current set of domain types, checks for misuse of coordinate names. This kind of validation can be performed quickly (even though it involves a fair amount of code) and it seems more like the symbolic equivalent of range checking than like the analog of checking the rank of a matrix.

Domain Structs

`PTOLEMY` currently supports only three domain types: 1) *RecDomainType*, i.e., oriented parallelepiped domains 2) *TradDomainType*, i.e., domain of the type traditionally used in the sinc-literature, and 3) *MappedDomainType*, i.e., domains that are implicitly defined by explicitly specifying the map, its inverse, and the parallelepiped onto which it is mapped. Each of these domain types has an associated "struct" type and a validation procedure.

RecDomainStruct

The type *RecDomainStruct* is equivalent to

```
list(range)
```

Each element of the list corresponds to a coordinate and defines the range of that particular coordinate. The order of the range specification is important, since the coordinate range being specified is determined by the order of the coordinates.

The limits of each the range can be finite, infinite, or a symbolic constant, but the ranges may not depend on any of the coordinates.

check_rec_domain(`Coord:` *list(name)*, `Struct:` *RecDomainStruct*)

This procedure checks that all of the limits are independent of all of the coordinate names.

TradDomainStruct

The type *TradDomainStruct* is defined to be

```
list(name=range)
```

Each element is a range specification for one of the coordinates. Unlike the *RecDomainType*, the coordinate ranges need not be specified in the order of the coordinates. As a result in order to tell which coordinate is being specified, each range specification must have the coordinate name on the left hand side (LHS) of the equation. However, the order of the range specifications is not unimportant; the limits of each range may be defined in terms of any of the previously defined coordinates, but not in terms of the coordinate currently being defined or in terms of any of the yet undefined coordinates.

check_trad_domain(`Coord:` *list(name)*, `Struct:` *TradDomainStruct*)

This procedure first checks that the number of range definitions matches the number of coordinates. Then it checks each of the range specifications in order, keeping track of which coordinates still need to be defined. At each step the procedure ensures that the coordinates currently being defined are in the set of still undefined coordinates. Then it uses the Maple `indets` command to quickly build a set of intermediate expressions occurring in the limits of current range specification. The subset of intermediate expressions that are of type ***name*** are extracted to form the set of free variables. Each free variable is then checked to see if it is in the set of coordinates names still to be defined. Finally, the coordinate name currently being defined is removed from the set of coordinate names still to be defined.

MappedDomainStruct

Mapped domains implicitly specify a domain by specifying the mapping from the domain onto a specified parallelepiped. This provides a user with the ability to employ problem specific domains of a type for which automatic map generation is not currently possible. Of course, this capability requires the user to explicitly specify both the map and its inverse.

The type *MappedDomainStruct* is defined to be

```
[RecDomainStruct, procedure, procedure]
```

The first element specifies the region onto which the domain is mapped, and the second and third arguments define the map and its inverse.

check_mapped_domain(`Coord:` *list(name)*, `Struct:` *MappedDomainStruct*)

This procedure first calls **check_rec_domain** to validate the first element, then it checks that both procedures are from $\mathbb{C}^n$ to $\mathbb{C}^n$ where n is the number of coordinates. The dimension of the range and domain of the procedure are checked by calling **range_dim** and **domain_dim** from the LLF ***proc_dim***. The possibility of **range_dim** returning `UNKNOWN` is also handled. The range dimensionality is assumed to be correct unless it can be conclusively determined to be wrong. That is, results equal to `UNKNOWN` results are assumed to be correct, because they can not be proven to be incorrect (even though they can not be proven to be correct).

Domain Types

RecDomainType and MultiRecDomainType

The conceptual framework for these types is presented in the subsubsection labeled "RecDomainStruct" on page 42.

The type expression for *RecDomainType* is

```
[list(name), range,..., range]
```

The first element is a list of the coordinate names. The rest of the elements in the list are the range specifications corresponding to each dimension.

For example, the following code fragment

```
[[x1,x2,x3], 0..1, 0..infinity, -n..n];
```

is a three-dimensional *RecDomainType* specifying the region shown in Figure 2.4.

The type expression for *MultiRecDomainType* is

```
[list(name), list(range),..., , list(range)]
```

The first element defines the coordinate names and each list of ranges defines one of the subdomains.

For example the code fragment

```
[[x,y] [0..1,0..1], [1..2,0..1], [1..2,1..2]]
```

defines an object of type *MultiRecDomainType* that defines the three subdomains shown in Figure 2.5.

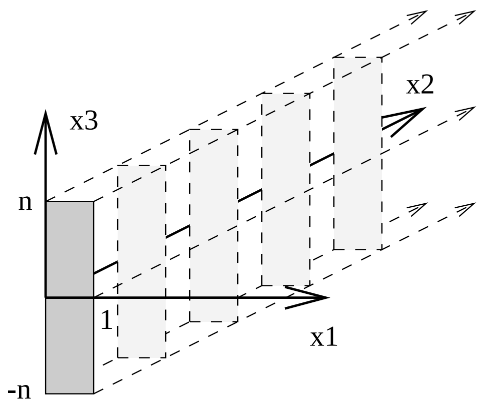

Figure 2.4: A Three-Dimensional Semiinfinite Parallelepiped

This domain extends from the $x1$-$x3$ plan in the direction of increasing $x2$, forming a rectangular cylinder.

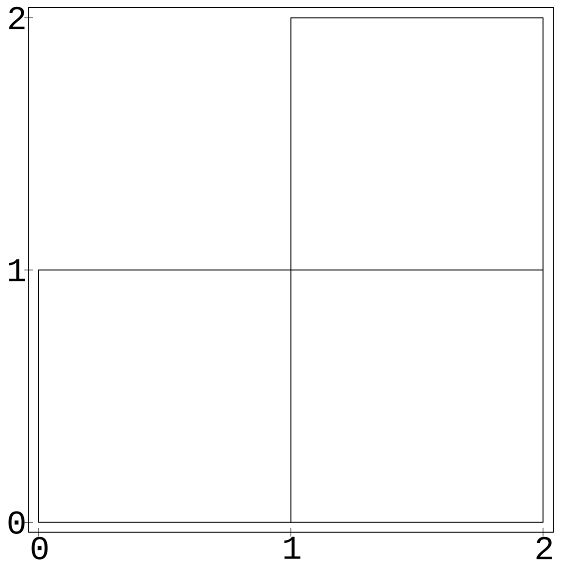

Figure 2.5: The ‘L’-Domain Divided into Rectangles

TradDomainType and MultiTradDomainType

The conceptual framework for these types is presented in the subsubsection titled “TradDomainStruct” on page 42.

The type expression for *TradDomainType* is

```
[list(name), name=range,..., name=range]
```

The first element is an ordered list of the coordinate names and the other elements are the range specifications. An ordering list of the the coordinates is vital in order to construct a usable map from the domain to an arbitrary parallelepiped. If an unordered list of coordinates would suffice it could be extracted from the range specifications.

The code fragment

```
[[x,y], y=0..1, x=sin(2*Pi*y)/2 ..  sin(2*Pi*y)+1];
```

is of type *TradDomainType* and specifies the two-dimensional domain illustrated in Figure 2.6.

The type expression for *MultiTradDomainType* is

```
[list(name), list(name=range),...,, list(name=range)]
```

The first element defines the coordinate names and each subsequent element defines one of the subdomains.

For example the code fragment

```
[[x,y], [x=0..1, y=0..2-x], [y=1..2, x=2-y ..  y+1], [x=2..3, y=0..x-1]]
```

defines an object of the type *MultiTradDomainType*, which defines the three subdomains shown in Figure 2.7.

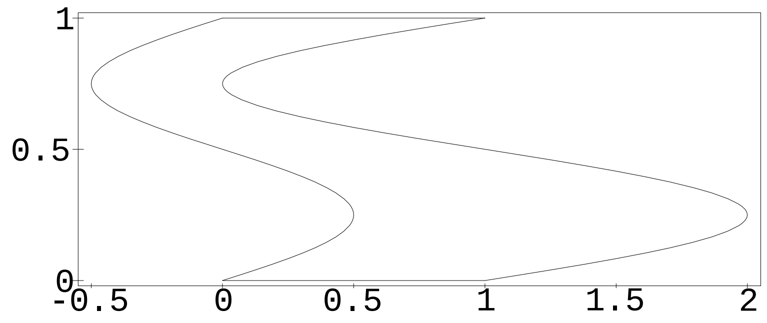

Figure 2.6: The Fat 'S' Domain

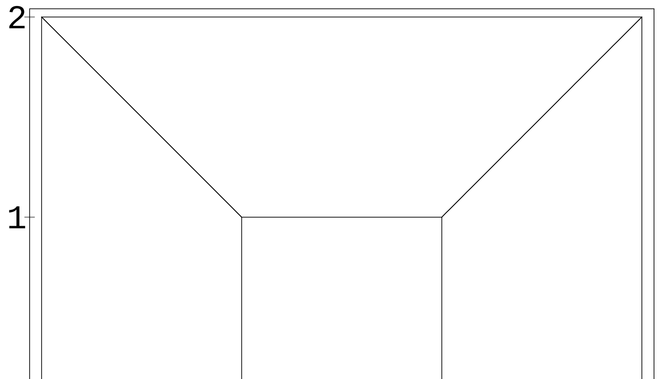

Figure 2.7: Three Trapezoidal Subdomains

MappedDomainType and MultiMappedDomainType

The conceptual framework for this type is presented on page 43.

The type expression for *MappedDomainType* is

```
[list(name), RecDomainStruct, procedure, procedure]
```

The first element specifies the coordinate names and the rest of the elements are the elements of the corresponding type *MappedDomainStruct*.

The following simple example illustrates using a *MappedDomainType* and a map that converts from cartesian to polar coordinates to define a domain that could not be represented as a *TradDomainType*.

Start of Maple Worksheet

```
> with(ptolemy,PlotDomBound);
```

$$[\mathit{PlotDomBound}]$$

```
> Rec := [[r,theta], 1/2..1, Pi/6..11*Pi/6];
```

$$\mathit{Rec} := \left[[r, \theta], \frac{1}{2}..1, \frac{1}{6}\pi..\frac{11}{6}\pi\right]$$

```
> Map := (x,y) -> (sqrt(x^2 + y^2), arctan(y,x));
```

$$\mathit{Map} := (x, y) \to (\mathrm{sqrt}(x^2 + y^2), \arctan(y, x))$$

```
> InvMap := (r,theta) -> (r*cos(theta), r*sin(theta));
```

$$\mathit{InvMap} := (r, \theta) \to (r\cos(\theta), r\sin(\theta))$$

```
> Domain := [Rec, eval(Map), eval(InvMap)];
```

$$\mathit{Domain} := \left[\left[[r, \theta], \frac{1}{2}..1, \frac{1}{6}\pi..\frac{11}{6}\pi\right], \right.$$
$$\left. (x, y) \to (\mathrm{sqrt}(x^2 + y^2), \arctan(y, x)), (r, \theta) \to (r\cos(\theta), r\sin(\theta))\right]$$

```
> PlotDomBound(Domain, -1..1, -1..1, tickmarks=[5,5]);
```

See Figure 2.8

End of Maple Worksheet

The type expression for *MappedTradType* is

```
[list(name), [list(range), procedure, procedure],...,
 [list(range), procedure, procedure]]
```

The first element defines the coordinate names and each subsequent element defines one of the subdomains.

The following example shows the use of a *MultiMappedDomainType* to define a rather complicated set of subdomains that could not be defined using *TradDomainTypes*.

Start of Maple Worksheet

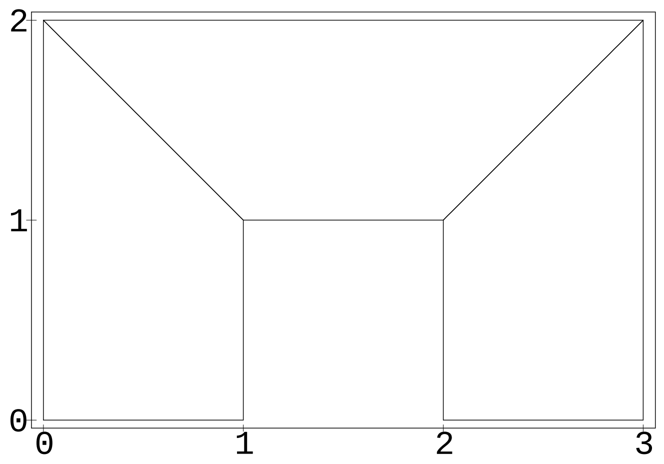

Figure 2.8: A 'C'-Shaped Domain

```
> with(ptolemy, PlotDomBound);
```

$$[\, PlotDomBound \,]$$

```
> MakeMapPair := (CentX,CentY) ->
>   (subs(Result =
>       (sqrt((x-CentX)^2 + (y-CentY)^2), arctan((y-CentY),(x-CentX))),
>      (x,y) -> Result),
>    subs(Result = (r*cos(theta)+CentX, r*sin(theta)+CentY),
>      (r,theta) -> Result));
```

$$MakeMapPair := (\, CentX, \, CentY \,) \to \Big(\text{subs}\,\Big(Result = \Big(\text{sqrt}\,\big((\, x - CentX \,)^2 + (\, y - CentY \,)^2\big)\,, \arctan(\, y - CentY, x - CentX \,)\Big)\,, (\, x, y \,) \to Result\Big)\,, \text{subs}(\, Result = (\, r\cos(\,\theta\,) + CentX, r\sin(\,\theta\,) + CentY \,), (\, r, \theta \,) \to Result \,)\Big)$$

```
> Domain := [[x,y],
>   [[1/2..3/2, 0..Pi], MakeMapPair(0,0)],
>   [[1/2..3/2, -Pi..0], MakeMapPair(-2,0)],
>   [[1/2..3/2, -Pi..0], MakeMapPair(2,0)],
>   [[5/2..7/2, 0..Pi], MakeMapPair(0,0)]];
```

$$Domain := \Bigg[[\, x, y \,], \Bigg[\bigg[\frac{1}{2}..\frac{3}{2}, 0..\pi\bigg], (\, x, y \,) \to \Big(\sqrt{x^2 + y^2}, \arctan(\, y, x \,)\Big)\,, (\, r, \theta \,) \to (\, r\cos(\,\theta\,), r\sin(\,\theta\,)\,)\Bigg], \Bigg[\bigg[\frac{1}{2}..\frac{3}{2}, -\pi..0\bigg], (\, x, y \,) \to \Big(\sqrt{x^2 + 4\,x + 4 + y^2}, \arctan(\, y, x + 2 \,)\Big)\,, (\, r, \theta \,) \to (\, r\cos(\,\theta\,) - 2, r\sin(\,\theta\,)\,)\Bigg], \Bigg[\bigg[\frac{1}{2}..\frac{3}{2}, -\pi..0\bigg],$$

$$(x,y) \to \left(\sqrt{x^2-4\,x+4+y^2}, \arctan(\,y, x-2\,)\right),$$
$$\left.(r,\theta) \to (\,r\cos(\,\theta\,)+2, r\sin(\,\theta\,)\,)\right],\ \left[\left[\frac{5}{2}..\frac{7}{2}, 0..\pi\right],\right.$$
$$\left.\left.(x,y) \to \left(\sqrt{x^2+y^2}, \arctan(\,y, x\,)\right), (\,r,\theta\,) \to (\,r\cos(\,\theta\,), r\sin(\,\theta\,)\,)\right]\right]$$

```
> type(Domain, MultiMappedDomainType);
```
true

```
> PlotDomBound(Domain, axes=boxed);
```
See Figure 2.9.

End of Maple Worksheet

DomainType and MultiDomainType

The type *DomainType* is simply the intersection of all of the domain types known to **PTOLEMY**. As has repeatedly been stated, at the moment there are only three domain types, *RecDomainType*, *TradDomainType*, and *MappedDomainType*. Nevertheless, for software engineering purposes, programmers should still use the type named *DomainType* rather than enumerate the current list of supported domains.

The *MultiDomainType* represents a collection of subdomains each of which may be in any of the domain classes. The type expression is

```
[list(name), DomainStruct,..., DomainStruct]
```

The following example illustrates the use of the *MultiDomainType* to mix subdomains from different domain classes.

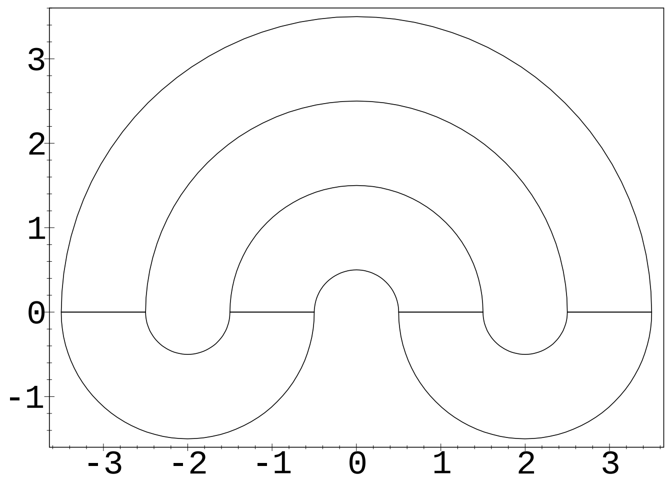

Figure 2.9: Four Arc Sector Subdomains with Different Centers

Start of Maple Worksheet

```
> with(ptolemy, PlotDomBound);
```

$$[PlotDomBound]$$

```
> Domain := [[x,y],
>   [y=-1..0, x=-3/2+y/2 .. -1],
>   [[1..2, Pi/2..Pi],
>      (x,y)->(sqrt((2*x + 1)^2 + y^2), arctan(y,2*x+1)),
>      (r,theta)->(r*cos(theta)/2 - 1/2,r*sin(theta))],
>   [-1/2..1/2, 1..2],
>   [[1..2, 0..Pi/2],
>      (x,y)->(sqrt((2*x - 1)^2 + y^2), arctan(y,2*x-1)),
>      (r,theta)->(r*cos(theta)/2 + 1/2,r*sin(theta))],
>    [y=-1..0, x=1 .. 3/2-y/2]];
```

$$Domain := \left[[x,y], \left[y=-1..0, x=-\frac{3}{2}+\frac{1}{2}y..-1\right], \left[\left[1..2, \frac{1}{2}\pi..\pi\right],\right.\right.$$
$$(x,y)\to\left(\mathrm{sqrt}\left((2x+1)^2+y^2\right), \arctan(y, 2x+1)\right),$$
$$\left.(r,\theta)\to\left(\frac{1}{2}r\cos(\theta)-\frac{1}{2}, r\sin(\theta)\right)\right], \left[\frac{-1}{2}..\frac{1}{2}, 1..2\right], \Bigg[$$
$$\left[1..2, 0..\frac{1}{2}\pi\right], (x,y)\to\left(\mathrm{sqrt}\left((2x-1)^2+y^2\right), \arctan(y, 2x-1)\right),$$
$$\left.\left.(r,\theta)\to\left(\frac{1}{2}r\cos(\theta)+\frac{1}{2}, r\sin(\theta)\right)\right], \left[y=-1..0, x=1..\frac{3}{2}-\frac{1}{2}y\right]\right]$$

```
> type(Domain, MultiDomainType);
```

$$true$$

```
> PlotDomBound(Domain, axes=boxed);
```

See Figure 2.10.

End of Maple Worksheet

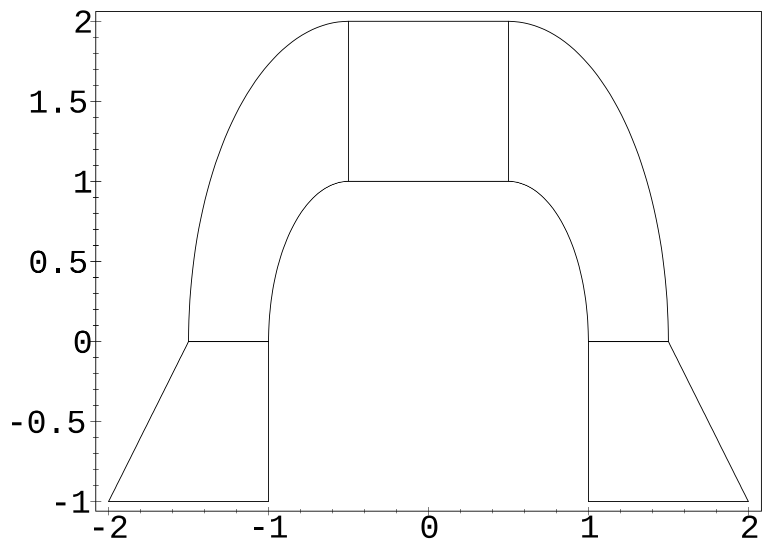

Figure 2.10: Five Subdomains of Three Diferent Types

Problem Types

Problems defined over multiple subdomains require inter-subdomain coupling equations and geometric information indicating which subdomain boundaries are connected to each other. The resulting extra notational complexity is significant enough to be awkward when defining a problem over a single domain. As a result, **PTOLEMY** usually defines different types for corresponding problem elements of a single domain problem and a multidomain problem. In such cases, the corresponding types have the same name, except that the prefix *Multi* is prepended to the type name of the type corresponding to a multidomain problem.

This section describes the types required to define the various kinds of problem descriptions over a single domain. The next section titled, "MultiProblem Types" on page 57 describes the types required to define problems over collections of subdomains.

BoundCondType, BoundTagType, and BoundFormType

The type *BoundCondType* describes a boundary constraint, whereas *BoundTagType* describes a tagged expression to be evaluated on the boundary, and *BoundFormType* is the union of the two types.

The type expression for *BoundCondType* is

```
[posint,'EndType', equation]
```

The first two elements of the list indicate the boundary along which the constraint is to be applied and the third element specifies the constraint.

The first two elements only indirectly specify a boundary in the original domain; instead they directly specify a boundary of the parallelepiped onto which the domain will be mapped. The first element indicates the dimension perpendicular to the edge and the second element specifies which limit of the range of this particular dimension comprises the boundary.

For the case of a *RecDomainType* the correspondence between boundaries of the original domain and the mapped-to domain is trivial, so the specification can easily be thought of as directly specifying a boundary in the original domain. In the case of other domain types the correspondence in still unambiguous, but it may not be intuitive. To minimize the resulting confusion the maps created by the *TradDomainType* ADT avoid reflections and rotations. See the section titled "MakeTradMap" on page 79 for a more complete description of these maps.

The type expression for *BoundTagType* is

```
[posint,'EndType', algebraic, anything]
```

The first two elements describe the boundary along which the expression is to be evaluated (in the same manor as for *BoundCondType*). The third element is an arbitrary expression to be evaluated along this boundary. The typical use of this type is to equate two different expression evaluated along two different boundaries; in such applications it is useful to associate a label with each expression, referred to as "the expression tag" in this manual. The fourth element specifies this tag.

The type expression for *BoundFormType* is

```
{'BoundCondType', 'BoundTagType'}
```

ProbType

This type defines a PDE in its most natural sense over a single domain defined by a *DomainType* field.

The type expression for this type is

```
[collection(equation),
 collection('BoundCondType'),
 'DomainType',
 collection('OrderSpecType')]
```

The first element defines the PDE(or system of PDEs), the second element defines the boundary constraints, the third element defines the domain over which the PDE is applied, and the final element defines the requested order of approximation of the final solution.

For example the potential in the interior of the resistive conductor shown in Figure 2.11 can be determined by solving a rather simple PDE. The following code shows how the PDE might be defined using a *ProbType* structure.

Start of Maple Worksheet

```
> Top := (x+2)*(x-2)*cos(Pi*x)/3 + 1/2;
```

$$Top := \frac{1}{3}(x+2)(x-2)\cos(\pi x) + \frac{1}{2}$$

Maple Worksheet Continued on Next Page

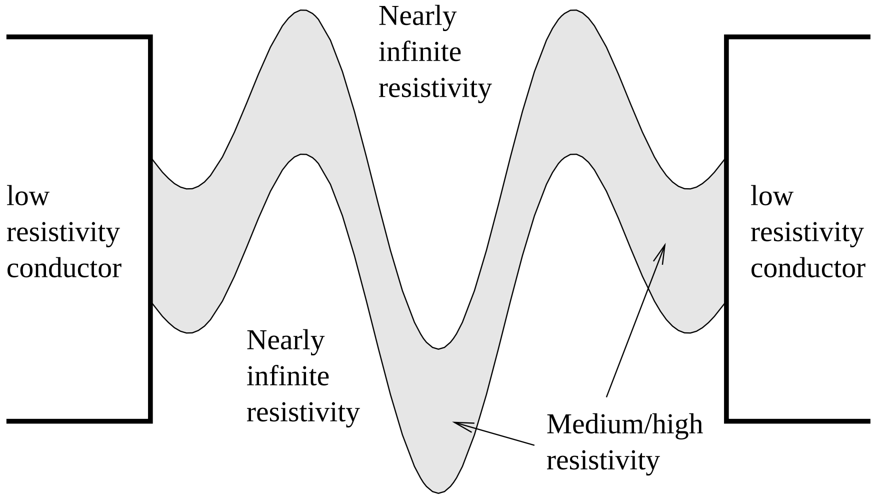
Nearly infinite resistivity
low resistivity conductor
low resistivity conductor
Nearly infinite resistivity
Medium/high resistivity

Figure 2.11: A Fuse Like Conductor

```
> Bottem := (x+2)*(x-2)*cos(Pi*x)/3 - 1/2;
```

$$Bottem := \frac{1}{3}(x+2)(x-2)\cos(\pi x) - \frac{1}{2}$$

```
> Domain := [[x,y], x=-2..2, y=Bottem..Top];
```

$$Domain := \Big[[x,y], x = -2..2,$$
$$y = \frac{1}{3}(x+2)(x-2)\cos(\pi x) - \frac{1}{2}..\frac{1}{3}(x+2)(x-2)\cos(\pi x) + \frac{1}{2}\Big]$$

```
> Laplacian := D[1,1](V) + D[2,2](V) = 0;
```

$$Laplacian := D_{1,1}(V) + D_{2,2}(V) = 0$$

```
> NormDir := [diff(Bottem,x), -1];
```

$$NormDir := \Big[$$
$$\frac{1}{3}(x-2)\cos(\pi x) + \frac{1}{3}(x+2)\cos(\pi x) - \frac{1}{3}(x+2)(x-2)\sin(\pi x)\pi,$$
$$-1\Big]$$

```
> BoundCond :=
>   [[1,LOW, V=Vl], [1,HIGH, V=Vh],
>    [2,LOW, NormDir[1]*D[1](V) + NormDir[2]*D[2](V) = 0],
>    [2,HIGH, NormDir[1]*D[1](V) - NormDir[2]*D[2](V) = 0]];
```

$$BoundCond := \Big[[1, LOW, V = Vl], [1, HIGH, V = Vh], \Big[2, LOW, \Big($$
$$\frac{1}{3}(x-2)\cos(\pi x) + \frac{1}{3}(x+2)\cos(\pi x) - \frac{1}{3}(x+2)(x-2)\sin(\pi x)\pi\Big)$$
$$D_1(V) - D_2(V) = 0\Big], \Big[2, HIGH, \Big($$
$$\frac{1}{3}(x-2)\cos(\pi x) + \frac{1}{3}(x+2)\cos(\pi x) - \frac{1}{3}(x+2)(x-2)\sin(\pi x)\pi\Big)$$
$$D_1(V) + D_2(V) = 0\Big]\Big]$$

```
> Prob := [Laplacian, BoundCond, Domain, [V, [0,0]]];
```

$$Prob := \Big[D_{1,1}(V) + D_{2,2}(V) = 0, \Big[[1, LOW, V = Vl], [1, HIGH, V = Vh], \Big[$$
$$2, LOW, \Big($$
$$\frac{1}{3}(x-2)\cos(\pi x) + \frac{1}{3}(x+2)\cos(\pi x) - \frac{1}{3}(x+2)(x-2)\sin(\pi x)\pi\Big)$$
$$D_1(V) - D_2(V) = 0\Big], \Big[2, HIGH, \Big($$
$$\frac{1}{3}(x-2)\cos(\pi x) + \frac{1}{3}(x+2)\cos(\pi x) - \frac{1}{3}(x+2)(x-2)\sin(\pi x)\pi\Big)$$

$$D_1(V) + D_2(V) = 0\Big]\Big], \Big[[x, y], x = -2..2,$$
$$y = \frac{1}{3}(x+2)(x-2)\cos(\pi x) - \frac{1}{2}..\frac{1}{3}(x+2)(x-2)\cos(\pi x) + \frac{1}{2}\Big],$$
$$[V, [0, 0]]\Big]$$

```
> ptolemy[init]();
> type(Prob, ProbType);
```

true

End of Maple Worksheet

RecProbType

This type specifies a PDE problem over a single rectangular domain. Its definition is exactly the same as *ProbType* except that the third element must be of type *RecDomainType.*

OverRideType

This type specifies an override equations. Override equations specify some alternative to the default equations for a specific group of collocation events.

The type expression for this type is

```
[equation, name,list(integer)]
```

The first element is the override equation, the second and third element's specify the group of collocation events at which this override equations should supersede the default equation.

PTOLEMY identifies collocation events by specifying the associated state-variable and the collocation point "number." The collocation point "number" is always defined relative to the set of collocation points needed to approximation the associated state-variable.

In each dimension the collocation points are numbered from $-(N + n_l)$ to $N + n_h$, where N is the sampling parameter, n_l is the number of extra collocation points on the low end, and n_h is the number of extra collocation points on the high end. In more than one dimension, collocation points are "numbered" by the tuple whose elements are the index number of the projection of the collocation point onto each of the dimensions.

Since **PTOLEMY** also associated a unique collocation point with each bases, collocation events can also be identified by specifying a bases. In addition, since bases are grouped into groups whose linear combination can be symbolically manipulated as a single entity (see [7]), it makes sense to group collocation events in the same manner. It is this grouping of collocation events that is specified by the *OverRideType.*

The bases group number is similar to the collocation point number except that all of the sinc bases are in group 0, the high-end extra collocation points are number from 1 up, and the low-end extra collocation points are numbered from -1 down. This is illustrated in Figure 2.12.

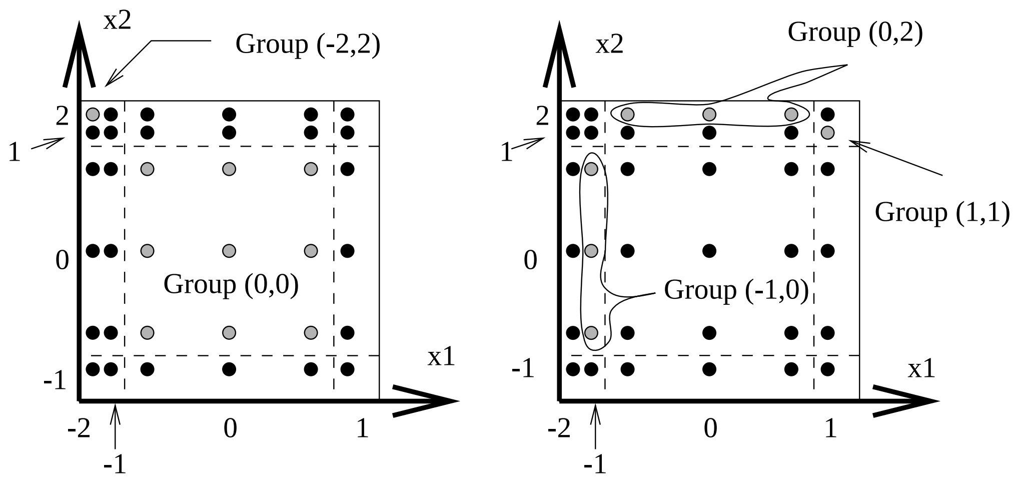

Figure 2.12: Examples of Collocation Group Numbers

SProbType

This type specifies a collocation system over a single (rectangular) domain using SB-notation.

The type expression of this type is:

```
[collection(equation),
 collection(OverRideType),
 list(name),
 list(MapInfoType),
 collection(OrderSpecType)]
```

The first element contains the collocation equations; there is a one-to-one correspondence between these equations and the governing equations of the original PDE. The second element specifies override equations, which are typically derived from the original boundary constraints. The third element is a list of the coordinate names. (Typically these coordinate names are distinct from the coordinate name of the original problem definition.) The fourth element is a list of the sinc-maps used in the collocation process; these maps will be needed in the next stage of the solution process to determine weighting functions. The fifth, and final, element contains the order of approximation actually achieved by the collocation process.

MultiProblem Types

Multidomain problems are similar to the single domain problems discussed in the last section, but they require several additional constructs for specifying the way in which subdomains are coupled.

In the original problem definition it is necessary to define which boundaries are shared between subdomains. The *CoupleType* is used for this purpose.

After each subdomain is mapped to a parallelepipeds, it is no longer required that (or even always possible), for shared parallelepiped boundaries to actually exist at the same position in space. The result is that the relative orientation of coupled subdomains can not be inferred once each subdomain has been mapped to a parallelepiped. This information, combined with the coupling information present in the original problem statement, is preserved in structures of type *CoupleOrientType*.

Associated with each coupled pair of boundaries is a list of coupling equations, used to preserve the needed order of smoothness across the boundary. This list of coupling equations and the coupling orientation information is combined into a record of type *CoupleEqType*.

Finally, when collocation is performed these coupling equations are converted into SB-notation. The list of collocated coupling equations and the coupling orientation information is stored in a record of type *CoupleOverType*.

These types are used, in conjunction with types used to define single domain problem types, to define multidomain types analogous to *ProbType*, *RecProbType*, and *SProbType*.

CoupleType and CoupleOrientType

Both of these types indicate that two boundaries of mapped-to subdomains were a single shared boundaries prior to mapping. This is necessary because each subdomain may be mapped to an arbitrary parallelepiped so that it is no longer obvious how the subdomains connect. In fact, it is not always possible to choose the parallelepipeds so that the shared boundaries in the original domain are shared once each subdomain is mapped to a parallelepiped. This can be seen by considering the four subdomains in Figure 2.13.

The difference between the *CoupleType* and the *CoupleOrientType* is that the *CoupleOrientType* includes information about the way in which the two mapped domains must be oriented in order for the boundaries to match up. This orientation information can be determined by knowing the maps for each of the subdomains, but once these maps are no longer part of the problem specification it is no longer possible to reconstruct the orientation information from other problem elements. So the orientation information must be extracted from the original domain specifications and combined with the coupling information in order to fully define each coupling.

This is the justification for the *CoupleOrientType*. The argument that this type should *not* be used in the original problem definition (i.e., the justification for also having a *CoupleType*) is that the orientation information is implicitly defined in the domain specifications. Letting the user define redundant information is not only unnecessary, but also results in more user errors.

It is important to realize that both of these types are defined with respect to oriented parallelepipeds, that is with respect to the mapped-to subdomains. So for an

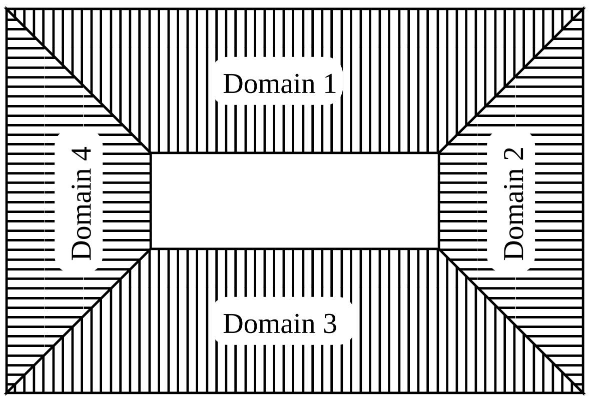

Figure 2.13: Four Domains that Cannot be Mapped to Parallelepipeds While Preserving Connectivity

n-dimensional domain the boundaries are an oriented parallelepiped that varies in exactly $n-1$ of the dimensions.

CoupleType The type expression for this type is:

```
[posint, posint,'EndType'] = [posint, posint,'EndType']
```

Each side of the equation specifies one subdomain boundary. The first element of either boundary specification indicates the subdomain number. The second element indicates the dimension that is perpendicular to the boundary, and the third element indicates which of the two boundaries that are perpendicular to this dimension is being specified.

For example the two *TradDomainType*'s

```
[[x,y], y=0..1, x=0..2-y]
```

and

```
[[x,y], x=1..2, y=2-x..2]
```

share a common boundary, as shown in Figure 2.14. The mappings of these two domains to parallelepipeds is determined by the procedure **MakeTradMap** except for dilation and translations. An illustrative mapping of these two subdomains to parallelepipeds is shown in Figure 2.15. Clearly, the shared boundary in first subdomain is specified by `[1,HIGH]` and in the second subdomain by `[2,LOW]`. So the corresponding *CoupleType* is

```
[1, 1,HIGH] = [2, 2,LOW]
```

CoupleOrientType The type expression for this type is:

```
[posint, posint,'EndType', list(nonzeroint)] =
 [posint, posint,'EndType', list(nonzeroint)]
```

Each side of the equation specifies a *boundary orientation.* The first three elements of each boundary orientation specification are exactly the same as the boundary specification in *CoupleType*. The last element, however, specifies how the boundary must be

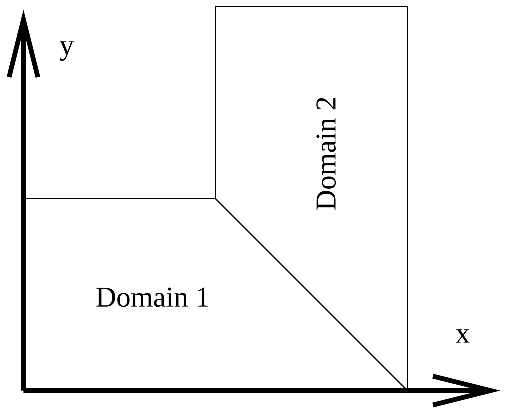

Figure 2.14: The Two Canonical Subdomains of the ‘L’-Problem

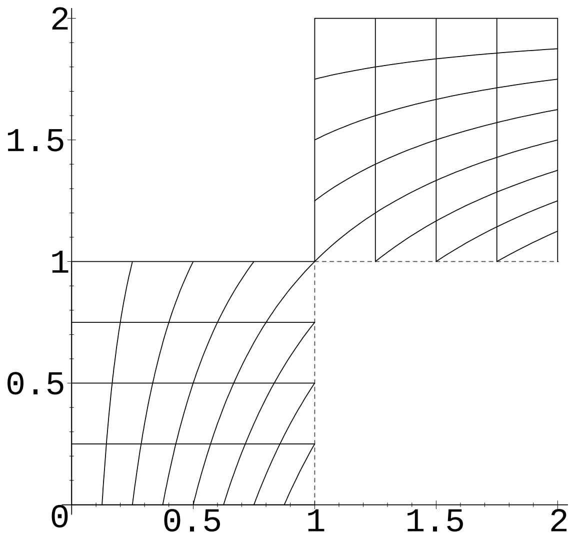

Figure 2.15: Representative Maps for the Subdomains in Figure 2.14

This figure shows the way in which a grid on the original domain appears after being mapped to two parallelepipeds domains. The dashed line is the map of the original boundary between the subdomains. Notice that it appears in each of the subdomains. The grid lines where 1/4 unit a part in the original domain, so the distortion of scale introduced by the maps can be visualized by the distortion of the grids.

oriented in order to match an implicitly defined mapping of the original shared boundary. Specifying how to orient each subdomain so that the boundary matches some implicit mapping of the boundary indirectly specifies how each subdomain must be oriented in order to match each other along the shared boundary.

It would perhaps be less confusing if the structure defined the orientation of the subdomain on the right hand side (RHS) in terms of the mapping of the boundary in the LHS subdomain. In practice this convention is employed by **MultiToRec**, but the extra generality (and associated confusion) is supported because during the building of the matrix systems the boundary coordinates are assigned distinct names. The implicitly defined mapping of the shared boundary defines the correspondence of the boundary coordinates to the domain coordinates. This allows users to explicitly control the assignment of coordinates to the boundary, by manually creating (or editing) the coupling orientation information.

The intent of the orientation information is to describe the required manipulation in a translation and scale invariant manner. That is, the orientation information describes a particular mapping (in this case the mapping required to align two boundaries) *except for some translation and and scaling.* In this case all of the boundaries are aligned with the coordinate system and lie on an $(n-1)$-dimensional subspace. This means that any mapping between boundaries will be a transformation of the form:

$$(y_1, \ldots, y_n) := M(x_1, \ldots, x_n) = \big(a_1\, x_{i_1} + b_1, \ldots a_n\, x_{i_n} + b_n)\big)$$

where $a_k = 0$ for the value of k for which y_k is constant in the mapped to boundary. As a result the transformation may be simplified to

$$\begin{aligned} y_1 &= a_1\, x_{i_1} + b_1 \\ &\vdots \\ y_{k-1} &= a_{k-1}\, x_{i_{k-1}} + b_{k-1} \\ y_k &= b_k \\ y_{k+1} &= a_{k+1}\, x_{i_{k+1}} + b_{k+1} \\ &\vdots \\ y_n &= a_n\, x_{i_n} + b_n \end{aligned}$$

Since the translation defined by the b's and the scaling is defined by the absolute values of the a's all of the orientation information is completely defined by the signs of the a terms and the i_k values. This information is encoded in the last element of the *CoupleOrientType* in the form

$$\big[\, \mathrm{sign}(a_1)\, i_1, \ldots \mathrm{sign}(a_{k-1})\, i_{k-1}, \mathrm{sign}(a_{k+1})\, i_{k+1}, \mathrm{sign}(a_n)\, i_n \big]$$

In more direct terms the orientation indicates, for the subset of coordinates that vary over the mapped to boundary, which coordinate in the form the original domain matches it rather the direction of the coordinate is reversed.

So in the example illustrate in Figure 2.14 and Figure 2.15 the *CoupleOrientType* could be either:

```
[1, 1,HIGH, [2]] = [2, 2,LOW, [-1]]
```

or

```
[1, 1,HIGH, [2]] = [2, 2,LOW, [-1]]
```

CoupleEqType

This type defines all of the required coupling information once the subdomains have been mapped to parallelepipeds. That is it specifies both coupling orientation information and a list of coupling equations.

The type expression for this type is:

```
['CoupleOrientType', list(algebraic=algebraic)]
```

The first element contains all of the geometric information about the coupling, whereas the second element contains a list of *all* of the coupling equations.

CoupleOverType

This type defines all of the required coupling information once collocation has been performed. That is it specifies the coupling orientation information and a list of collocated coupling equations. These collocated coupling equations are similar to override equations except that each side of the equation is applied at a different collocation point, typically in different subdomains.

The type expression for this type is:

```
['CoupleOrientType', list(
 [algebraic,name,list(integer)] = [algebraic,name,list(integer)]) ]
```

The first element contains all of the geometric information about the coupling, and the second element contains a list of the collocated coupling equations. Each side of the representation of the collocated coupling equations represents the expression to be evaluated in the subdomain corresponding to this side of the coupling. Each expression is evaluated in its respective subdomains and then equated to form a single collocation event.

Each side of the collocated coupling equations is analogous to the *OverRideType*s used in the single domain problem types, except that the first element of the list is an expression instead of an equation. Currently, **PTOLEMY** reserves collocation events in both subdomains for this coupling equation. The final conversion to a system of algebraic equations must, of course, use only one of the two collocation events, but the type must specify both collocation points in order to specify where each expression is to be evaluated. It is expected that part of the **PTOLEMY** system assign collocation events will be redesigned in the next version of the system primarily to remedy this weakness.

MultiProbType

This type specifies a problem over a coupled collection of subdomains. The problem must use a single (collection of) governing equation(s) over all of the subdomains. The collection of subdomains may be heterogeneous.

The type expression for this type is:

```
[collection(equation),
 list(collection('BoundCondType')),
 collection('CoupleType'),
 'MultiDomainType',
 collection('MultiOrderSpecType')]
```

The first element is the governing equation (which may be a system of equations). The second element is a list of collections of boundary conditions; each collection of boundary conditions applies to the corresponding subdomain. The third element specifies all of the couplings within the problem. That is it specifies all of the subdomains boundaries that are an artifact of domain decomposition. The fourth element specifies the collocation of subdomains. The fifth and final element specifies the requested order of approximation for each state-variable. The actual order of approximation may need to be greater than this in order to correctly apply boundary conditions, but it will never be less than this.

It is possible for two boundaries of two different subdomains to physically lie on the same spatial surface and still not be coupled. Though such problems are nonphysical this kind of problem definition may be an important part of modeling a physical process. For example in an electrostatics problem two subdomains might be separated by an insulator with negligible conductivity and negligible width. Rather then define a gap between the domains many orders of magnitude smaller then the other dimensions of the problem, the model might simply define the two subdomains to touch but to be uncoupled.

However, two boundaries that do not lie on the same surface should not be coupled. Such coupling might result form some models of physical phenomenon, for example in steady state heat flow problems two parallel plates might be constrained to be at the same temperature, within the precision of the model, because of radiation. Coupling two surface requires the construction of a bijection between the two surfaces. Since such bijections is not unique it must be explicitly specified by the user or infered by through some convention. When the two boundaries are really the same surface **PTOLEMY** can safely employ the identity bijection. All other cases require manual intervention. The user who wishes to do this kind of modeling should add hand constructed coupling equations after the mapping to a collection of parallelepiped has been performed.

MultiRecProbType

This type defines a system of coupled PDE's, each defined on a *RecDomainType*. The governing equations may be completely unrelated from one subdomain to the next. However the state-variables must be the same across all of the subdomain. In addition all of the problems must have a common set of coordinate names (which in turn implies a common dimensionality).

The type expression for this type is:

```
[list(collection(equation)),
 list(collection('BoundCondType')),
 collection('CoupleEqType'),
 'RecMultiDomainType',
 collection('MultiOrderSpecType')]
```

The first element is a list of governing equations one per subdomain. Each "governing equation" may in fact be a system of equations. The second element is a list of collections of boundary conditions, one collection per subdomain. The third element is collection of information about each of the couplings in the problem. The fourth element defines the collection of rectangular subdomains. The fifth and final element specifies the desired order of approximation for each of the state-variables over each of the subdomains. Just as in the ***MultiProbType*** the actual order of approximation may need to be greater than this in order to correctly apply boundary conditions, but it will never be less than this.

MultiSProbType

This type defines a system of collocation events, each defined over the cross product of domains of the type $\mathcal{D}_d$ (See [5] for the definition of $\mathcal{D}_d$).

Just as for ***MultiRecProbType***, the governing equations may be unrelated in each subdomain. However, the state-variables must use the same coordinate names across all of the subdomains..

The type expression for this type is:

```
[list(collection(equation)),
 list(collection(OverRideType)),
 collection('CoupleOverType'),
 list(name),
 list(list(MapInfoType)),
 collection('MultiOrderSpecType')]
```

The first element is a list of governing equations, one per subdomain. The second element is a list of collections of override equations, one collection per subdomain. The third element is a collection of information about each coupling within the problem. The fourth element is a list of the coordinate names used by all of the subdomains. The fifth element is a list of lists of the sinc-maps used in the collocation process. Each list of maps corresponds to a subdomains and contains one sinc-map per coordinate. The sixth and final element specifies the order of approximation actually used by the collocation process. This information implicitly defines all of the bases used as part of the collocation process.

Linear Kron Types

In the final stage of the problem solution the system of collocation events corresponding to a linear problem is converted into a matrix problem. The representation of this matrix problem uses Kronecker product notation, which is discussed in greater detail in [7] and [10].

ParamCompType

This type merely represents a collection of simulation parameters (i.e., N's and H's in the notation of the sinc-literature). The type does not specify the parameters values. Rather, it is used in **PTOLEMY** to specify that a collection of N's (or H's) all have the same value.

The type expression for this type is:

$$\mathit{collection}([\mathit{posint}, \mathit{posint}])$$

The first element in each pair specifies the subdomain, and the second element specifies the dimension; the combination uniquely determines either an N parameter, an H parameter, or the combination depending on the context.

The collection of parameters that must all have the same value is determined by first constructing a graph where the parameter values are the nodes of the graph and equivalence constraints from the edges and then extracting the components of the graph. Thus the type name stands for "parameter component type."

LinKronType

This type represents a matrix problem in Kronecker Product notation.

The type expression for this type is

```
[list(list(algebraic)), list(name), list(algebraic),
 list(name), list(name),
 list(ParamCompType)]
```

The first element represents a matrix of algebraic terms. The matrix is represented in row major form, with each of the inner lists representing a row of the matrix. Each element in the matrix is the Kronecker Product expression for the corresponding block in the actual matrix problem. The block matrix must be a square matrix with a block order equal to the number of block variables.

The second element defines a column-vector of block variable names. The block variable names are specially constructed to contain information about the physical quantities they represent. If the state-variable in the original problem statement was called `V`, then

- The block variables `V_1`, `V_2`, and so on, represent the scalar multipliers for core sinc bases corresponding to the state-variable `V` in subdomain 1, subdomain 2, and so on.

- The block variables `V_1_B1_1`, `V_1_B1_2`, and so on, represent the scalar multipliers for the bases that are splines only in dimension 1, but are sinc-basis in all the other dimensions. The numbering of these bases corresponds to the numbering of the extra bases in dimension 1.
- Finally, block variables of the form `V_1_B13_1`, `V_1_B13_2`, and so on, represent the scalar multipliers for the bases that are splines in dimensions 1 and 3, but are sinc-basis in all the other dimensions. **PTOLEMY** provides the procedures **CombineToNum** and **NumToCombine** to assist the user in computing the correspondence between basses and sequence numbers.

The third element defines a column vector of expressions defining the RHS of the system. Each expression represents a column vector whose size matches the number of columns in the corresponding block-row of the block-matrix .

The fourth element defines the coordinate names. The coordinate names appear in the algebraic expressions in the first and third *LinKronType* elements, so they are integral to the problem definition. The fifth element defines the names of the H parameters appearing in the algebraic expressions in the first and third element. The order of the H parameter names must match the order of the coordinates, and is therefore significant.

The sixth and final element is a list of the parameter components. When the final matrix is built parameter values must be specified for each of these parameter values as a group.

Chapter 3

Mapping to Parallelepipeds

The sinc bases are defined as one-dimensional functions, but most problems are multidimensional. The numerical analysis community realizes that methods for approximating one dimension function can be generalized with suitable work to methods for approximation in any number of dimensions. As a result the vast majority of the sinc literature is devoted to solving one-dimensional problems.

In order to have relevancy as a tool for automatic setup, **PTOLEMY** must directly address higher-dimensional problems. To leave the generalization to two and three dimensions as an exercise for the reader is reasonable if the point of the work is to convey mathematical insight. If the point of the work is to enumerate the details essential for automating the setup of a problem class then this point of view is tantamount to ignoring the common case.

The most common way of generalizing spectral methods to higher dimensions is to use bases that are the tensor product of the one dimensional bases. For the unmapped sinc base, i.e., defined for the entire real line and used to approximate function on the strip $\mathcal{D}_d$, this method works well. However, constructing a map from the domain of analyticity which we wish to exploit to the higher-dimensional extension of the strip $\mathcal{D}_d$ is quite a bit more involved than for the one-dimensional case.

A few sinc researchers (see Section 7.4 of [10]) have proposed to tackle this problem by applying two maps in sequence, the first to map the problem domain to a parallelepiped and the second to map a desired region of analyticity in the complex hyper-plane surrounding this parallelepiped to the higher dimension extension of $\mathcal{D}_d$. The final map might be expressed as $\phi(z) := \phi_1(\phi_2(x))$ and its inverse would be $\phi^{-1} := \phi_2^{-1}(\phi_1^{-1}(z))$.

The primary advantages of this two step approach are that:

1. The tensor products of one-dimensional *mapped* sinc-bases can be used to directly approximate the function over the parallelepiped region. That is, the aspects of ϕ which maps the domain to an orthogonal coordinate system are completely captured in ϕ_1.

2. The aspects of the map that controls which portion of the complex hyper-

plan surrounding the domain will be exploited in the approximation are completely captured by the second map, i.e., ϕ_2.

PTOLEMY adopts this approach. This chapter describe how to map a problem defined over a fairly general domain to a parallelepiped.

The result of this mapping to a parallelepiped is an implicitly defined domain (or subdomain) specific coordinate system that would map onto the naturally defined coordinate system of the parallelepiped onto which the domain is mapped. This domain-specific coordinate system is not explicitly utilized by **PTOLEMY** but is often useful for simplifying proofs and developing insights.

Warp

This procedure performs a change of variable on a *collection* of equations or equation parts.

Warp(`Eq`: *collection({algebraic, equation})*, `Map`: *procedure*, `InvMap`: *procedure*)
Warp(`Eq`: *collection({algebraic, equation})*, `Map`: *procedure*, `InvMap`: *procedure*,
`StateVar`: *collection(name)*})
Warp(`Eq`: *collection({algebraic, equation})*, `Map`: *procedure*, `InvMap`: *procedure*,
`Coord`: *list(name)*, `NewCoord`: *list(name)*)
Warp(`Eq`: *collection({algebraic, equation})*, `Map`: *procedure*, `InvMap`: *procedure*,
`StateVar`: *collection(name)*}, *Coord:list(name), NewCoord:list(name)*)

The argument `Eq` is the "equation" to be mapped. The argument `Map` and `InvMap` should implement a mathematical function from $\mathbb{C}^n$ to $\mathbb{C}^n$ for some positive integer n.

If the optional argument `StateVar` is specified, then it specifies which symbols in the equation should be assumed to be functions of the coordinates. If the optional argument `StateVar` is not specified then the procedure **free_var** is called to construct the list of state-variables. The procedure **free_var** considers any unassigned name in the "equation" that has not been specified as a constant (using Maple's `assume` command), as a state-variable.

Not specifying the optional argument `StateVar` will often cause symbolic constants to be treated as state-variables. Fortunately, since the map of a name is simply the same name, treating a symbolic constant as a state-variable rarely causes an error. Although, doing the extra work of mapping a name only to get an output equal to the input is less efficient than not doing superfluous work.

In addition, a symbolic constant may be used in some ways that would be illegal for a state-variable. In such cases allowing **Warp** to treat the symbolic constant as a state-variable will cause errors. In other cases a particular notation may have two different meanings depending on whether a particular symbol is a state-variable or not. In these instances providing the optional argument `StateVar` is not only more efficient, but logically necessary.

If the optional arguments `Coord` and `NewCoord` are used, they specify coordinate names in the original and the new domains, respectively. If these optional arguments are not specified then the argument names for `Map` and `InvMap` will be used instead. These names are easily extracted and are typically acceptable choices for the coordinate names.

However, problems occur if the arguments for `Map` and `InvMap` overlap, if any of the coordinate names are assigned global values, or if the "equation" explicitly references some of the coordinate names and these names are not the arguments of `Map`. The procedure recognizes the first two of these problems and generates an appropriate error.

The simplification procedure specified by the global variable `ptolemy/SimpProc` will be automatically applied to the coefficient of the resulting "equation."

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, Warp,GridInvMap);
```

$$[\, GridInvMap,\ Warp\,]$$

```
> Map0 := x -> ln(x/(1-x));
```

$$Map0 := x \to \ln\left(\frac{x}{1-x}\right)$$

```
> InvMap0 := x -> exp(x)/(1 + exp(x));
```

$$InvMap0 := x \to \frac{e^x}{1+e^x}$$

```
> ODE := (D@@2)(V) + (1-x)*D(V) + x*V = K;
```

$$ODE := D^{(2)}(V) + (1-x)\,D(V) + x\,V = K$$

```
> Warp(ODE, Map0,InvMap0);
Error, (in Warp) The Coordinate name(s), x, overlap domains.
```

```
> z := 'z';
```

$$z := z$$

```
> InvMap0 := z -> exp(z)/(1 + exp(z));
```

$$InvMap0 := z \to \frac{e^z}{1+e^z}$$

```
> Warp(ODE, Map0,InvMap0);
```

$$\frac{D^{(2)}(V)\,(1+e^z)^4}{(e^z)^2} + \frac{e^z\,V}{1+e^z} + \frac{(1+e^z)\,\big((e^z)^3+(e^z)^2-1\big)\,D(V)}{(e^z)^2} = K$$

```
> z := x;
```

$$z := x$$

```
> Warp(ODE, Map0,InvMap0);
Error, (in Warp) The coordinate name(s), z, have global values.
```

```
> Map1 := (x1,x2) -> (x1, (x2-x1) / (2-x1));
```

$$Map1 := (x1, x2) \to \left(x1, \frac{x2-x1}{2-x1}\right)$$

```
> InvMap1 := (y1,y2) -> (y1,y2*(2-y1) + y1);
```

$$InvMap1 := (y1, y2) \to (y1, y2\,(2-y1)+y1)$$

```
> GridInvMap(InvMap1, [7,7], [0..1, 0..1], xtickmarks=3);
```

See Figure 3.1

```
> Laplacian_2d := D[1,1](V) + D[2,2](V) = 0;
```

$$Laplacian\_2d := D_{1,1}(V) + D_{2,2}(V) = 0$$

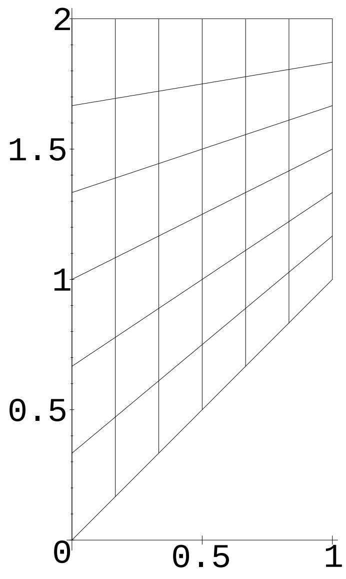

Figure 3.1: Gridding of the First Inverse Map

Maple Worksheet Continued from Previous Page

```
> start := time(): Warp(Laplacian_2d, Map1,InvMap1); time() - start;
```

$$2\,\frac{(-1+y2)\,D_2(V)}{(-2+y1)^2}-2\,\frac{(-1+y2)\,D_{1,2}(V)}{-2+y1}+D_{1,1}(V)$$
$$+\frac{(2-2\,y2+y2^2)\,D_{2,2}(V)}{(-2+y1)^2}=0$$

2.283

```
> Map2 := (x1,x2) -> (x1/(2-x2), x2);
```

$$Map2 := (x1, x2) \to \left(\frac{x1}{2-x2}, x2\right)$$

```
> InvMap2 := (y1,y2) -> (y1*(2-y2), y2);
```

$$InvMap2 := (y1, y2) \to (y1\,(2-y2), y2)$$

```
> GridInvMap(InvMap2, [7,7], [0..1, 0..1], ytickmarks=3);
```

See Figure 3.2

```
> start := time(): Warp(Laplacian_2d, Map2,InvMap2); time() - start;
```

$$2\,\frac{y1\,D_1(V)}{(-2+y2)^2}-2\,\frac{y1\,D_{1,2}(V)}{-2+y2}+\frac{(1+y1^2)\,D_{1,1}(V)}{(-2+y2)^2}+D_{2,2}(V)=0$$

2.650

```
> Special := D[1,1](V) + D[2,2](V) = D[2](Q);
```

$$Special := D_{1,1}(V)+D_{2,2}(V)=D_2(Q)$$

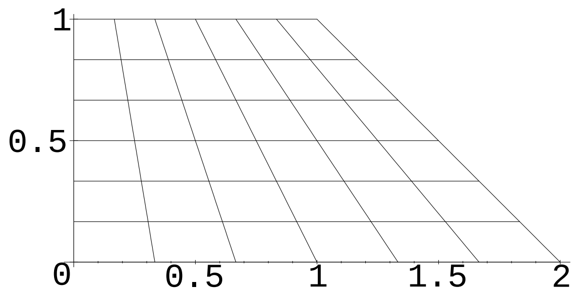

Figure 3.2: Gridding of the Second Inverse Map

Maple Worksheet Continued from Previous Page

```
> Warp(Special, Map2,InvMap2);
```

$$2\,\frac{y1\,D_1(V)}{(-2+y2)^2} - 2\,\frac{y1\,D_{1,2}(V)}{-2+y2} + \frac{(1+y1^2)\,D_{1,1}(V)}{(-2+y2)^2} + D_{2,2}(V) = -\frac{D_1(Q)\,y1}{-2+y2} + D_2(Q)$$

```
> Warp(Special, Map2,InvMap2, {V});
```

$$2\,\frac{y1\,D_1(V)}{(-2+y2)^2} - 2\,\frac{y1\,D_{1,2}(V)}{-2+y2} + \frac{(1+y1^2)\,D_{1,1}(V)}{(-2+y2)^2} + D_{2,2}(V) = D_2(Q)$$

```
> Map3 := (x1,x2) -> (sqrt(x1^2 + x2^2), arctan(x1,x2));
```

$$Map3 := (x1, x2) \to (\mathrm{sqrt}(x1^2 + x2^2), \arctan(x1, x2))$$

```
> InvMap3 := (r,theta) -> (r*cos(theta),r*sin(theta));
```

$$InvMap3 := (r, \theta) \to (r\cos(\theta), r\sin(\theta))$$

```
> Warp(Laplacian_2d, Map3,InvMap3);
```

$$\frac{r^2\left(\sin(\theta)^2+\cos(\theta)^2\right)D_1(V)}{\left(r^2\left(\sin(\theta)^2+\cos(\theta)^2\right)\right)^{3/2}} + D_{1,1}(V) + \frac{D_{2,2}(V)}{\left(\sin(\theta)^2+\cos(\theta)^2\right)r^2} = 0$$

```
> `ptolemy/SimpProc` := simplify;
```

$$ptolemy/SimpProc := simplify$$

```
> Start := time(): Warp(Laplacian_2d, Map3,InvMap3); time() - Start;
```

$$\frac{\mathrm{csgn}(r)\,D_1(V)}{r} + D_{1,1}(V) + \frac{D_{2,2}(V)}{r^2} = 0$$

$$5.484$$

```
> assume(R>0);
> Start := time():
> temp := Warp(Laplacian_2d, Map3,InvMap3, [x1,x2],[R,theta]);
```

$$temp := \frac{D_1(V)}{R\tilde{}} + D_{1,1}(V) + \frac{D_{2,2}(V)}{R\tilde{}^2} = 0$$

```
> time() - Start;
```

$$4.983$$

```
> subs(R = r, temp);
```

$$\frac{D_1(\,V\,)}{r} + D_{1,1}(\,V\,) + \frac{D_{2,2}(\,V\,)}{r^2} = 0$$

```
> readlib(`ptolemy/pde_collect`):
> Map4 := (x1,x2,x3) ->
>   (sqrt(x1^2 + x2^2 + x3^2),
>    arctan(x2,x1),
>    arccos(x3/sqrt(x1^2 + x2^2 + x3^2)));
```

$$Map4 := (\,x1, x2, x3\,) \to \Big(\operatorname{sqrt}(\,x1^2 + x2^2 + x3^2\,), \arctan(\,x2, x1\,), \arccos\left(\frac{x3}{\operatorname{sqrt}(\,x1^2 + x2^2 + x3^2\,)}\right)\Big)$$

```
> InvMap4 := (r,theta,phi) ->
>   (r*cos(theta)*sin(phi), r*sin(theta)*sin(phi), r*cos(phi));
```

$$InvMap4 := (\,r, \theta, \phi\,) \to (\,r\cos(\,\theta\,)\sin(\,\phi\,), r\sin(\,\theta\,)\sin(\,\phi\,), r\cos(\,\phi\,)\,)$$

```
> Laplacian_3d := D[1,1](V) + D[2,2](V) + D[3,3](V) = 0;
```

$$Laplacian\_3d := D_{1,1}(\,V\,) + D_{2,2}(\,V\,) + D_{3,3}(\,V\,) = 0$$

```
> Phi := 'Phi';
```

$$\Phi := \Phi$$

```
> assume(sin(Phi) > 0);
> `ptolemy/SimpProc` := simplify;
```

$$ptolemy/SimpProc := simplify$$

```
> Start := time():
> temp := Warp(Laplacian_3d, Map4,InvMap4, [x1,x2,x3],[R,Theta,Phi]);
> time() - Start;
```

$$temp := 2\,\frac{D_1(\,V\,)}{R\tilde{}} + \frac{D_{3,3}(\,V\,)}{R\tilde{}^2} + \frac{\cos(\,\Phi\tilde{}\,)\,D_3(\,V\,)}{\sqrt{1-\cos(\,\Phi\tilde{}\,)^2}\,R\tilde{}^2} + D_{1,1}(\,V\,) - \frac{D_{2,2}(\,V\,)}{(-1+\cos(\,\Phi\tilde{}\,)^2)\,R\tilde{}^2} = 0$$

$$16.633$$

```
> temp1 := subs(cos(Phi)^2 = 1 - sin(Phi)^2, temp);
```

$$temp1 := 2\,\frac{D_1(\,V\,)}{R\tilde{}} + \frac{D_{3,3}(\,V\,)}{R\tilde{}^2} + \frac{\cos(\,\Phi\tilde{}\,)\,D_3(\,V\,)}{\sqrt{\sin(\,\Phi\tilde{}\,)^2}\,R\tilde{}^2} + D_{1,1}(\,V\,) + \frac{D_{2,2}(\,V\,)}{\sin(\,\Phi\tilde{}\,)^2\,R\tilde{}^2} = 0$$

```
> temp2 := `ptolemy/pde_collect`(temp1,V, simplify, power);
```

$$temp2 := 2\,\frac{D_1(\,V\,)}{R\tilde{}} + \frac{D_{3,3}(\,V\,)}{R\tilde{}^2} + \frac{\cos(\,\Phi\tilde{}\,)\,D_3(\,V\,)}{\sin(\,\Phi\tilde{}\,)\,R\tilde{}^2} + D_{1,1}(\,V\,) + \frac{D_{2,2}(\,V\,)}{\sin(\,\Phi\tilde{}\,)^2\,R\tilde{}^2}$$

$$= 0$$

```
> subs(R=r, Theta=theta, Phi=phi, temp2);
```

$$2\,\frac{D_1(\,V\,)}{r} + \frac{D_{3,3}(\,V\,)}{r^2} + \frac{\cos(\,\phi\,)\,D_3(\,V\,)}{\sin(\,\phi\,)\,r^2} + D_{1,1}(\,V\,) + \frac{D_{2,2}(\,V\,)}{\sin(\,\phi\,)^2\,r^2} = 0$$

End of Maple Worksheet

Method of Implementation

Essentially all of the algorithmic operations of this procedure are performed by the low level procedure **fast_map**, which is described on page 240. The sole function of the `Warp` procedure is to assist interactive users in constructing the arguments required by **fast_map**. Since directly calling the procedure **fast_map** is potentially much faster and always at least a little faster, programmers are encouraged to call **fast_map** directly. However, the use of **Warp** is encouraged for interactive use, because it checks for several subtle errors.

Not all of the functionality provided by **fast_map** may be directly accessed through **Warp**. The procedure **Warp** assumes that all of the state-variables are functions of the coordinates. When the purpose of the mapping operation is to reduce the dimensionality of the problem, directly calling **fast_map** will be noticeably faster. In addition **Warp** will always use the same output state-variable names as the input state-variable names. When the intent is to perform nontrivial substitutions on the state-variable names as part of the mapping process then **fast_map** must be called directly.

This procedure uses the number of arguments to determine which combination of extra arguments has been specified. If there are exactly three arguments then no extra arguments have been specified; if there are four arguments then the state-variables have been specified but not the coordinate names; if there are five arguments then both sets of coordinate names have been specified but not the state-variables; if there are six coordinates then both the state-variables and both sets of coordinate names have been specified. If the state-variables are not specified they are determined by calling **free_var**. Similarly if the coordinate names are not specified they are determined by extracting the argument names from the procedures specified by `Map` and `InvMap`.

Next the procedure checks that the specified or defaulted coordinate names are lists of names and that there is no overlap between the coordinate names in the original domain and in the mapped-to domain. It then checks that there are no global variables with the same name as any of the coordinate names. Next it checks that the specified (or defaulted) state-variables are of type *collection(name)*.

Care is taken to ensure that the argument `NewCoordDepend` to **fast_map** specifies the minimum set of dependencies on the original coordinates. This is done by lexically checking for coordinate names in each component of the result of applying **Map** to the coordinate names.

It is possible for an input coordinate name to appear in one of the coordinates of **Map** but for the result not to actually depend on the coordinate. This is rarely a problem, since this can happen only if some simplification exists that would eliminate some of the coordinate name from the representation of the map. A great deal of extra care is applied when simplifying maps. In cases where the simplification process applied during the construction of the map was not adequate to eliminate unneeded coordinates from the map definition but the user possesses problem specific insights that can be used to prove that some of the coordinate functions do not depend of all of the coordinates that appear in their definitions, it would be better to use this insight to cause a better simplification of the map before calling **Warp**.

Dependencies

As mentioned in the introduction to this section, **Warp** may invoke **free_var** in order to construct an informed guess of the set state-variables. It will also invoke **fast_map** in order to perform the actual mapping.

MakeRecMap

This procedure will construct a map (and its inverse) from one oriented parallelepiped to another.

MakeRecMap(`Domain`: *RecDomainType*, `Map`: ***name***, `InvMap`: ***name***)
MakeRecMap(`Domain`: *RecDomainType*, `Map`: ***name***, `InvMap`: ***name***,
`NewCoord`: *list(name)*)
MakeRecMap(`Domain`: *RecDomainType*, `Map`: ***name***, `InvMap`: ***name***,
`NewDomain`: *RecDomainType*)

The global variable `ptolemy/SimpProc` indicates the simplification procedure to be applied to the algebraic expressions used to create the map and its inverse.

Description

The argument `Domain` describes the mapped-from domain. The arguments `Map` and `InvMap` specify the variable names to which the results will be assigned.

The coordinate names in the mapped-from domain are extracted from `Domain`. If the optional argument `NewCoord` is specified, it indicates the coordinate names to be used in the mapped-to domain. If the optional argument `NewDomain` is specified, it indicates both the mapped-to domain and the coordinate names to be used in the mapped-to domain. When the coordinate names of the mapped-to domain are not specified the coordinate names of the mapped-from domain are used. If the mapped-to domain is not specified, then the oriented unit cube (hyper-cube, square, or interval) such that each coordinate ranging from zero to one is used.

Because the results are expressed as procedures, the coordinate names (in either domain) are of limited importance, as long as they are unique and of the correct type (i.e., the Maple type ***name***). Their primary significance is the human readability of the printed form of the map. However, other procedures that make use of the map may (in some circumstances) extract and use these variable names in contexts in which they assume additional significance. For example **Warp** will assume that the argument names of the map are the coordinate names used in the equation(s) being wrapped, if coordinate names are not explicitly specified. For both of these reasons it is often better for interactive users to explicitly provide meaningful coordinate names.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy,MakeRecMap);
```

$$[\mathit{MakeRecMap}]$$

```
> Domain := [[x1,x2], -W..W, 0..H];
```

$$\mathit{Domain} := [[\mathit{x1}, \mathit{x2}], -\mathrm{W}..\mathrm{W}, 0..H]$$

```
> MakeRecMap(Domain, 'Map1','Map2');
> eval(Map1); eval(Map2);
```

$$(x1, x2) \to \left(\frac{1}{2}\frac{x1 + \mathrm{W}}{\mathrm{W}}, \frac{x2}{H}\right)$$

$$(x1, x2) \to (\mathrm{W}(2\,x1 - 1), H\,x2)$$

```
> MakeRecMap(Domain, 'Map1','Map2', [y1,y2]);
> eval(Map1); eval(Map2);
```

$$(x1, x2) \to \left(\frac{1}{2}\frac{x1 + \mathrm{W}}{\mathrm{W}}, \frac{x2}{H}\right)$$

$$(y1, y2) \to (\mathrm{W}(2\,y1 - 1), H\,y2)$$

```
> MakeRecMap(Domain, 'Map1','Map2', [[y1,y2], -1..1, -1..1]);
> eval(Map1); eval(Map2);
```

$$(x1, x2) \to \left(\frac{x1}{\mathrm{W}}, \frac{2\,x2 - H}{H}\right)$$

$$(y1, y2) \to \left(\mathrm{W}\,y1, \frac{1}{2}H(y2 + 1)\right)$$

```
> MakeRecMap(Domain, 'Map1','Map2', [[y1,y2], -alpha1..beta1, -alpha2..beta2]);
> eval(Map1); eval(Map2);
```

$$(x1, x2) \to \left(-\frac{1}{2}\frac{-\beta1\,x1 - \beta1\,\mathrm{W} - \alpha1\,x1 + \alpha1\,\mathrm{W}}{\mathrm{W}}, \frac{x2\,\beta2 + x2\,\alpha2 - \alpha2\,H}{H}\right)$$

$$(y1, y2) \to \left(\frac{\mathrm{W}(2\,y1 + \alpha1 - \beta1)}{\beta1 + \alpha1}, \frac{H(y2 + \alpha2)}{\beta2 + \alpha2}\right)$$

End of Maple Worksheet

Method of Implementation

Without loss of generality (WOLG) let $[x_1, \dots, x_n]$ be the coordinates of the mapped-from domain and let $[y_1, \dots, y_n]$ be the coordinates of the mapped-to domain. Then the mapped-from domain can be expressed as the cross product of the intervals $x_i = [\ell_i, h_i]$ for constant ℓ's and h's where $i \in \{1, \dots, n\}$. Similarly, the mapped-to domain can be expressed as the cross product of the intervals $y_i = [a_i, b_i]$ for the same set of i.

Then the map is defined by

$$y_i = \frac{(b_i - a_i)(x_i - \ell_i)}{h_i - \ell_i} + a_i$$

Because each coordinate function depends only on the corresponding input coordinate, the inverse map can be computed simply by inverting each coordinate function. The result is

$$x_i = \frac{(h_i - \ell_i)(y_i - a_i)}{b_i - a_i} + \ell_i$$

These equations are directly implemented.

Restrictions

The mapped-to domain must be finite. It would be possible to relax this restriction, but a special check would have to be made and a different algorithm would have to be used for unbounded "ends." This might seem advantageous since the normal use of **PTOLEMY** is to first map from the actual domain onto a finite parallelepiped and then to map from the finite parallelepiped onto the infinite parallelepiped. However, doing this mapping in one step would have the adverse affect of fixing the asymptotics in the unbounded direction. Even though the second map will usually be a product component of maps of the form

$$\phi_i(z_i) = \ln\left(\frac{y_i - a}{b - y_i}\right)$$

the ability to change this second map is critical when the performance of this standard map is inadequate.

Dependencies

This procedure is not dependent on any other part of the **PTOLEMY** system.

MakeTradMap

This procedure will construct a map from an arbitrary traditional domain to a parallelepiped.

MakeTradMap(`Domain:` *TradDomainType*, `Map:` *name*, `InvMap:` *name*)
MakeTradMap(`Domain:` *TradDomainType*, `Map:` *name*, `InvMap:` *name*,
`NewCoord:` *list(name)*)
MakeTradMap(`Domain:` *TradDomainType*, `Map:` *name*, `InvMap:` *name*,
`NewDomain:` *RecDomainType*)

The global variable `ptolemy/SimpProc` indicates the simplification procedure to be applied to the algebraic expressions used to create the map and the inverse map.

The calling sequence is the same as for **MakeRecMap** described on page 76. Of course, the allowed class of domains is much more general.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, MakeTradMap);
```

$$[\,MakeTradMap\,]$$

```
> D1 := [[x1,x2], x2=0..1, x1=0..2-x2];
```

$$D1 := [\,[\,x1, x2\,], x2 = 0..1, x1 = 0..2 - x2\,]$$

```
> ptolemy[PlotDomBound](D1);
```

See Figure 3.3.

Maple Worksheet Continued on Next Page

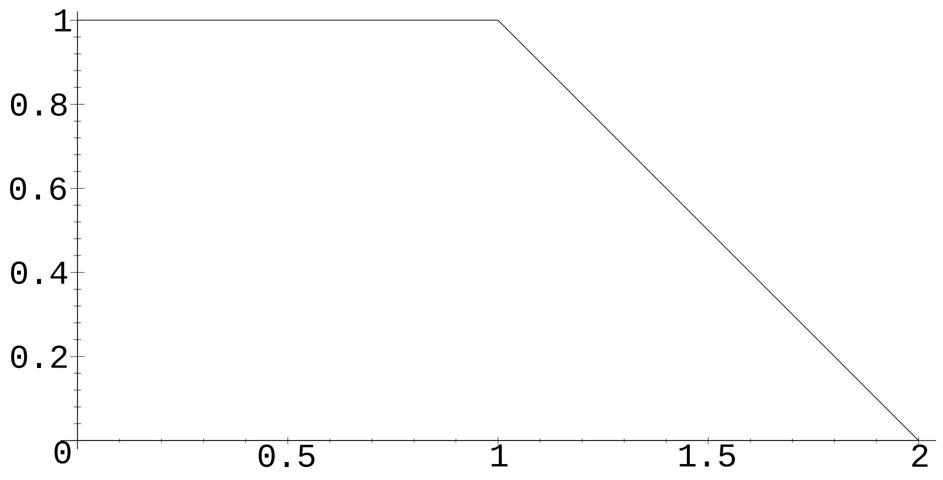

Figure 3.3: A Trapezoidal Domain

```
> MakeTradMap(D1, 'Map', 'InvMap');
> eval(Map); eval(InvMap);
```

$$(\,x1, x2\,) \to \left(-\frac{x1}{-2+x2}, x2\right)$$

$$(\,x1, x2\,) \to (\,-(\,-2+x2\,)\,x1, x2\,)$$

```
> MakeTradMap(D1, 'Map', 'InvMap', [z1,z2]);
> eval(Map); eval(InvMap);
```

$$(\,x1, x2\,) \to \left(-\frac{x1}{-2+x2}, x2\right)$$

$$(\,z1, z2\,) \to (\,-(\,-2+z2\,)\,z1, z2\,)$$

```
> MakeTradMap(D1, 'Map', 'InvMap', [[z1,z2], 0..2, 0..1]);
> eval(Map); eval(InvMap);
```

$$(\,x1, x2\,) \to \left(-2\,\frac{x1}{-2+x2}, x2\right)$$

$$(\,z1, z2\,) \to \left(-\frac{1}{2}\,(\,-2+z2\,)\,z1, z2\right)$$

```
> D2 := [[x,y,z], x=0..1, y=x-1..2-x, z=0..2+x+y];
```

$$D2 := [\,[\,x, y, z\,], x = 0..1, y = x-1..2-x, z = 0..2+x+y\,]$$

```
> Options := axes=boxed, labels=[x,y,z], grid=[15,15], orientation=[-30,50];
```

$$Options := axes = boxed, labels = [\,x, y, z\,], grid = [\,15, 15\,],$$
$$orientation = [\,-30, 50\,]$$

```
> ptolemy[PlotDomBound](D2, Options);
```
See Figure 3.4.

```
> MakeTradMap(D2, 'Map', 'InvMap');
> eval(Map); eval(InvMap);
```

$$(\,x, y, z\,) \to \left(x, \frac{-y+x-1}{-3+2\,x}, \frac{z}{2+x+y}\right)$$

$$(\,x, y, z\,) \to (\,x, 3\,y-2\,y\,x+x-1, -(\,-1-2\,x-3\,y+2\,y\,x\,)\,z\,)$$

End of Maple Worksheet

Method of Implementation

WOLG let $[x_1, \ldots, x_n]$ be the coordinate names of the original domain and let $[y_1, \ldots, y_n]$ be the coordinates of the mapped-to domain. Then construct $i[j]$'s such that the coordinate range specifications appear in the order $[x_{i[1]}, \ldots, x_{i[n]}]$. Then a more mathematical

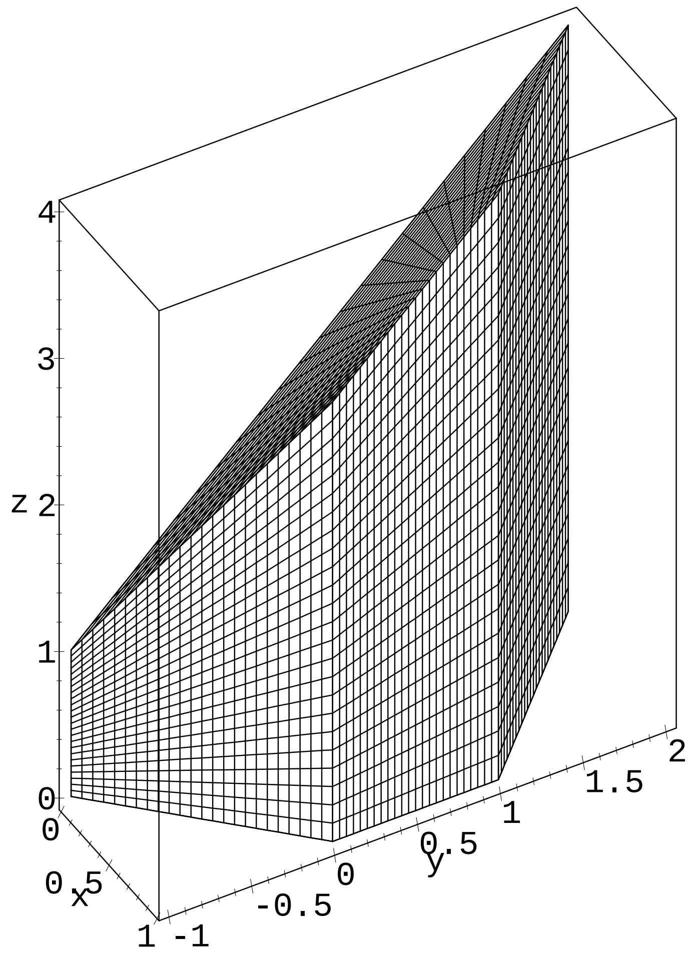

Figure 3.4: A Stylized Modern Office Building

form of the domain specification is:

$$\begin{aligned}
x_{i[1]} &\in [\ell_1, h_1]\\
x_{i[2]} &\in [\ell_2(x_{i[1]}), h_2(x_{i[1]})]\\
&\vdots\\
x_{i[n]} &\in [\ell_n(x_{i[1]}, \ldots, x_{i[n-1]}), h_n(x_{i[1]}, \ldots, x_{i[n-1]})]
\end{aligned}$$

where ℓ_1 to ℓ_n and h_1 to h_n are functions which give the lower and upper bounds of the corresponding range specifications.

Now let the correspondence between the x's and y's be such that $y_{i[j]}$ is a linear parameterization (from zero to one) of the interval $[\ell_{i[j]}, h_{i[j]}]$ for each dimension. This implies that the map is:

$$\begin{aligned}
y_{i[1]} &= \frac{x_{i[1]} - \ell_{i[1]}}{h_{i[1]} - \ell_{i[1]}}\\
y_{i[2]} &= \frac{x_{i[2]} - \ell_{i[2]}(x_{i[1]})}{h_{i[2]}(x_{i[1]}) - \ell_{i[2]}(x_{i[1]})}\\
&\vdots\\
y_{i[n]} &= \frac{x_{i[n]} - \ell_{i[n]}(x_{i[1]}, \ldots, x_{i[n]})}{h_{i[n]}(x_{i[1]}, \ldots, x_{i[n-1]}) - \ell_{i[n]}(x_{i[1]}, \ldots, x_{i[n-1]})}.
\end{aligned}$$

When each y_i actually varies from a_i to b_i, the result is

$$\begin{aligned}
y_{i[1]} &= \frac{(b_{i[1]} - a_{i[1]})(x_{i[1]} - \ell_{i[1]})}{h_{i[1]} - \ell_{i[1]}} + a_{i[1]}\\
y_{i[2]} &= \frac{(b_{i[2]} - a_{i[2]})(x_{i[2]} - \ell_{i[2]}(x_{i[1]}))}{h_{i[2]}(x_{i_{i[1]}}) - \ell_{i[2]}(x_{i[1]})} + a_{i[2]}\\
&\vdots\\
y_{i[n]} &= \frac{(b_{i[n]} - a_{i[n]})(x_{i[n]} - \ell_{i[2]}(x_{i_{i[1]}}, \ldots, x_{i[n]}))}{h_{i[2]}(x_{i[1]}, \ldots, x_{i[n-1]}) - \ell_{i[2]}(x_{i[1]}, \ldots, x_{i[n-1]})} + a_{i[n]}.
\end{aligned} \tag{3.1}$$

Because for each j, $y_{i[j]}$ depends only on the coordinates $x_{i[k]}$ for $k \in \{1, \ldots, j\}$ and the dependency of $y_{i[k]}$ on $x_{i[j]}$ is linear, the entire system may be inverted by substitution. The result is in the same form as Equation 3.1, but with the a's and b's swapped with the role of the ℓ's and h's.

$$\begin{aligned}
x_{i[1]} &= \frac{(h_{i[1]} - \ell_{i[1]})(y_{i[1]} - a_{i[1]})}{b_{i[1]} - a_{i[1]}} + \ell_{i[1]}\\
x_{i[2]} &= \frac{(h_{i[1]}(x_0) - \ell_{i[1]}(x_{i[1]}))(y_{i[1]} - a_{i[2]})}{b_{i[2]} - a_{i[2]}} + \ell_{i[1]}(x_0)\\
&\vdots\\
x_{i[n]} &= \frac{(h_{i[n]}(x_0, \ldots, x_{n-1})) - \ell_{i[n]}(x_0, \ldots, x_{n-1}))(y_{i[n]} - a_{i[n]})}{b_{i[n]} - a_{i[n]}} + \ell_{i[n]}(x_0, \ldots, x_{n-1})
\end{aligned} \tag{3.2}$$

The straightforward implementation of Equation 3.2 would be to construct a symbolic representation of Equation 3.2, then substituting the result for $x_{i[1]}$ into the results for $x_{i[2]}$ through $x_{i[n]}$, then substituting this result for $x_{i[2]}$ into the results for $x_{i[3]}$ through $x_{i[n]}$, and so on. Finally all of these results would need to be correctly ordered. The implementation actually used in **MakeTradMap** is less direct in order to make the total number of substitutions required linear in n, instead of quadratic. Specifically, the symbolic form of $x_{i[n]}$ is substituted into the list of the coordinate names, then the symbolic form of $x_{i[n-1]}$ is substituted into this result, and so on. This sequence of substitutions is actually performed with a single `subs` command.

Restrictions

Just as for **MakeRecMap** the mapped-to domain must be finite. See Section 3.2 for a justification of this restriction.

Dependencies

This procedure is not dependent on any other part of the **PTOLEMY** system.

MakeMappedMap

This procedure will extract and manipulate the map (and its inverse) from one mapped domain type.

MakeMappedMap(`Domain:` *MappedDomainType*, `Map:` *name*, `InvMap:` *name*)
MakeMappedMap(`Domain:` *MappedDomainType*, `Map:` *name*, `InvMap:` *name*, `NewCoord:` *list(name)*)
MakeMappedMap(`Domain:` *MappedDomainType*, `Map:` *name*, `InvMap:` *name*, `NewDomain:` *MappedDomainType*)

The global variable `ptolemy/SimpProc` indicates the simplification procedure to be applied to the algebraic expressions used to create the map and its inverse.

The calling sequence is the same as for **MakeRecMap** described on page 76. Except that the default mapped-to domain is not necessary the cross product of the intervals [0, 1]; instead the default domain is the parallelepiped used to define the original domain.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy,MakeMappedMap,PlotDomBound);
```

$$[\mathit{MakeMappedMap}, \mathit{PlotDomBound}]$$

```
> Domain :=
>   [[x,y], [1/2..1, 0..Pi],
>    (x,y) -> (sqrt(x^2 + y^2), arctan(y,x)),
>    (r,theta) -> (r*cos(theta),r*sin(theta))];
```

$$\mathit{Domain} := \left[[x,y], \left[\frac{1}{2}..1, 0..\pi\right], (x,y) \rightarrow (\mathrm{sqrt}(x^2+y^2), \arctan(y,x)), (r,\theta) \rightarrow (r\cos(\theta), r\sin(\theta))\right]$$

```
> type(Domain,MappedDomainType);
```

$$\mathit{true}$$

```
> PlotDomBound(Domain);
```

See Figure 3.5

```
> MakeMappedMap(Domain, 'Map','InvMap');
> eval(Map); eval(InvMap);
```

$$(x,y) \rightarrow (\mathrm{sqrt}(x^2+y^2), \arctan(y,x))$$

$$(r,\theta) \rightarrow (r\cos(\theta), r\sin(\theta))$$

```
> MakeMappedMap(Domain, 'Map','InvMap', [z1,z2]);
> eval(Map); eval(InvMap);
```

$$(x,y) \rightarrow (\mathrm{sqrt}(x^2+y^2), \arctan(y,x))$$

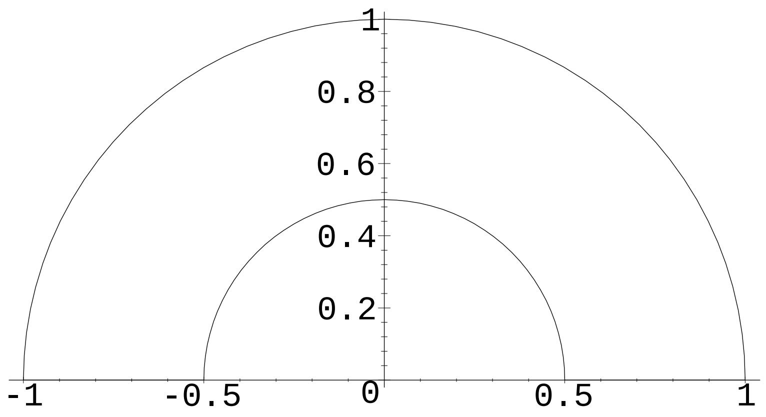

Figure 3.5: An Arc Sector Domain

Maple Worksheet Continued from Previous Page

$$(\,z1, z2\,) \to (\,z1\cos(\,z2\,), z1\sin(\,z2\,)\,)$$

```
> MakeMappedMap(Domain, 'Map','InvMap', [[z1,z2], 0..1, 0..1]);
> eval(Map); eval(InvMap);
```

$$(\,x, y\,) \to \left(2\sqrt{x^2+y^2}-1, \frac{\arctan(\,y, x\,)}{\pi}\right)$$

$$(\,z1, z2\,) \to \left(\frac{1}{2}(\,z1+1\,)\cos(\,\pi\,z2\,), \frac{1}{2}(\,z1+1\,)\sin(\,\pi\,z2\,)\right)$$

End of Maple Worksheet

Method of Implementation

If the optional argument `NewDomain` is not specified then the map used to define the domain is the map returned by the procedure. If the map's argument names are not the same as the coordinate names they will be changed so that they are. Similarly if the optional argument `NewCoord` is specified but the inverse-map's argument names do not match then the argument names of the inverse-map will be changed so that they do match.

If the optional argument `NewDomain` is specified then the process is the same but the result is composed with a map of the type formed by **MakeRecMap** which maps the parallelepiped used to define the domain to the requested output parallelepiped.

Restrictions

Just as for **MakeRecMap** the mapped-to domain must be finite. See Section 3.2 for a justification of this restriction.

Dependencies

This procedure is not dependent on any other part of the **Ptolemy** system.

ToRec

This procedure maps an entire problem specification from its natural domain to a parallelepiped domain.

ToRec(`Prob`: *ProbType*, `Coord`: *list(name)*, `MapInfo`: *name*)
ToRec(`Prob`: *ProbType*, `Coord`: *list(name)*, `MapInfo`: *name*,
    `OutRec`: *RecDomainStruct*)

The argument `Prob` defines the PDE to be mapped, the argument `Coord` defines the coordinate names in the mapped-to domain, the argument `MapInfo` specifies the name of the variable to which the map information will be assigned, and the optional argument `OutRec` defines the mapped-to domain. If the mapped to domain is not specified it defaults to the unit cube ranging from zero to one in each dimension, unless `Prob` is of type *MappedProbType*. In this case the mapped-to domain defaults to the parallelepiped used to define the problem domain.

Example Over Rectangular Domain

When `Prob` is of type *RecDomainType* the problem is already specified over a parallelepiped domain. The procedure **ToRec** will still map the problem from one rectangular domain to another. This was not the intended use of **ToRec**, but it is important to maintain consistent behavior for all of the subtypes of *ProbType*.

Start of Maple Worksheet

```
> with(ptolemy,ToRec);
```

$$[\,ToRec\,]$$

```
> Domain := [[x1,x2], 0..W, 0..H];
```

$$Domain := [\,[\,x1, x2\,], 0..\mathrm{W}, 0..H\,]$$

```
> Possian := D[1,1](V) + D[2,2](V) = (1-exp(x1)) * (1 - exp(x2));
```

$$Possian := D_{1,1}(\,V\,) + D_{2,2}(\,V\,) = (\,1 - e^{x1}\,)\,(\,1 - e^{x2}\,)$$

```
> BC := { seq(seq([i,End,V=0], End=[LOW,HIGH]), i=1..2) };
```

$$BC := \{[\,1, LOW, V = 0\,], [\,1, HIGH, V = 0\,], [\,2, LOW, V = 0\,],\\ [\,2, HIGH, V = 0\,]\}$$

```
> RecProb := [Possian, BC, Domain, [T, [0,0]] ];
```

$$RecProb := \Big[D_{1,1}(\,V\,) + D_{2,2}(\,V\,) = (\,1 - e^{x1}\,)\,(\,1 - e^{x2}\,), \{[\,1, LOW, V = 0\,],\\ [\,1, HIGH, V = 0\,], [\,2, LOW, V = 0\,], [\,2, HIGH, V = 0\,]\},\\ [\,[\,x1, x2\,], 0..\mathrm{W}, 0..H\,], [\,T, [\,0, 0\,]\,]\Big]$$

```
> type(RecProb,ProbType);
```

$$true$$

```
> ToRec(RecProb, [y1,y2], 'MapInfo');
```

$$\left[\frac{D_{1,1}(V)}{\mathrm{W}^2} + \frac{D_{2,2}(V)}{H^2} = (e^{(\mathrm{W}\, y1)} - 1)(e^{(H\, y2)} - 1), \{[1, LOW, V = 0], [1, HIGH, V = 0], [2, LOW, V = 0], [2, HIGH, V = 0]\}, [[y1, y2], 0..1, 0..1], [T, [0, 0]]\right]$$

```
> eval(MapInfo);
```

$$\left[(x1, x2) \to \left(\frac{x1}{\mathrm{W}}, \frac{x2}{H}\right), (y1, y2) \to (\mathrm{W}\, y1, H\, y2)\right]$$

End of Maple Worksheet

Example Over Traditional Domain

Consider the two-dimensional problem defined over the traditional domain

$$x_1 = [0, 2] \quad \text{and} \quad x_2 = [0, 1 + {}^1\!/\!_2\, x_1]$$

See Figure 3.6 from the example Maple session at the end of this subsection for an illustration of this domain. Assume that the governing equation over this domain is the Laplacian, i.e.,

$$\frac{\partial^2 T}{\partial x_1^2} + \frac{\partial^2 T}{\partial x_2^2} = 0.$$

Also assume that the boundary constraints are $T = 1$ along the left edge of the domain, $T = 0$ along the right edge of the domain, and $\nabla T \cdot \hat{n} = 0$, where $\hat{n}$ is the unit normal, along the top and bottom boundaries.

The traditional map from this domain to the unit square $[0, 1]^2$ is

$$\begin{aligned} y_1 &= {}^1\!/\!_2\, x_1 & \qquad x_1 &= 2y_1 \\ y_2 &= \frac{2x_2}{2 + x_1} & \qquad x_2 &= y_2(1 + y_1) \end{aligned}$$

Applying the multidimensional form of The Chain Rule yield,

$$\frac{\partial T}{\partial x_1} = \frac{\partial T}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial T}{\partial y_2}\frac{\partial y_2}{\partial x_1} \tag{3.3}$$

and

$$\begin{aligned} \frac{\partial^2 T}{\partial x_1^2} = {} & \\ & \frac{\partial^2 T}{\partial y_1^2}\left(\frac{\partial y_1}{\partial x_1}\right)^2 + \frac{\partial^2 T}{\partial y_1 \partial y_2}\left(\frac{\partial y_2}{\partial x_1}\right)\left(\frac{\partial y_1}{\partial x_1}\right) + \frac{\partial T}{\partial y_1}\frac{\partial^2 y_1}{\partial x_1^2} + \\ & \frac{\partial^2 T}{\partial y_2 \partial y_1}\left(\frac{\partial y_1}{\partial x_1}\right)\left(\frac{\partial y_2}{\partial x_1}\right) + \frac{\partial^2 T}{\partial y_2^2}\left(\frac{\partial y_2}{\partial x_1}\right)^2 + \frac{\partial T}{\partial y_2}\frac{\partial^2 y_2}{\partial x_1^2} \end{aligned} \tag{3.4}$$

Since

$$\frac{\partial y_1}{\partial x_1} = \frac{1}{2} \qquad \frac{\partial y_2}{\partial x_1} = \frac{-2x_2}{(2+x_1)^2} = \frac{-y_2}{2(1+y_1)}$$
$$\frac{\partial^2 y_1}{\partial x_1^2} = 0 \qquad \frac{\partial^2 y_2}{\partial x_1^2} = \frac{4x_2}{(2+x_1)^3} = \frac{y_2}{2(1+y_1)}$$

it follows that

$$\frac{\partial^2 T}{\partial x_1^2} = \frac{1}{4}\frac{\partial^2 T}{\partial y_1^2} - \frac{y_2}{2(1+y_1)}\frac{\partial^2 T}{\partial y_1 \partial y_2} + \frac{y_2^2}{4(1+y_1)^2}\frac{\partial^2 T}{\partial y_2 \partial y_1} + \frac{y_2}{2(1+y_1)}\frac{\partial T}{\partial y_2}$$

The expansion of $\partial^2 T/\partial x_2^2$ is similar to Equation 3.4, but since

$$\begin{aligned} &\frac{\partial y_1}{\partial x_2} = 0 \qquad \frac{\partial y_2}{\partial x_2} = \frac{2}{2+x_1} = \frac{1}{1+y_1} \\ &\frac{\partial^2 y_1}{\partial x_2^2} = 0 \qquad \frac{\partial^2 y_2}{\partial x_2^2} = 0 \end{aligned} \tag{3.5}$$

all the terms are zero except the term involving $\partial^2 T/\partial y_2^2$. It follows that

$$\frac{\partial^2 T}{\partial x_2^2} = \frac{1}{(1+y_1)^2}\frac{\partial^2 T}{\partial y_2^2} \tag{3.6}$$

Combining Equation 3.5 and Equation 3.6 yields the mapped form of the Laplacian.

The boundary constraints $T = 1$ and $T = 0$ are unaffected by the mapping. Along the lower boundary $\hat{n}$ equals the vector $(0, -1)$ and along the upper boundary $\hat{n}$ equals the vector $(-1/\sqrt{5}, 2/\sqrt{5})$. As a result these boundary constraints may be rewritten as

$$-\frac{\partial T}{\partial x_2} = 0$$
$$-\frac{1}{\sqrt{5}}\frac{\partial T}{\partial x_1} + \frac{2}{\sqrt{5}}\frac{\partial T}{\partial x_2} = 0$$

Using an expansion analogous to Equation 3.3, i.e.,

$$\frac{\partial T}{\partial x_2} = \frac{\partial T}{\partial y_1}\frac{\partial y_1}{\partial x_2} + \frac{\partial T}{\partial y_2}\frac{\partial y_2}{\partial x_2} \tag{3.7}$$

the lower boundary constraint becomes

$$\frac{-1}{1+y_1}\frac{\partial T}{\partial y_2} = 0$$

Using Equation 3.3 and Equation 3.7, the upper boundary constraint becomes

$$\frac{-1}{2\sqrt{5}}\frac{\partial T}{\partial y_1} + \frac{2(4-y_2)}{2\sqrt{5}(1+y_1)}\frac{\partial T}{\partial y_1} = 0$$

Since $y_2 = 1$ along this entire boundary, this may be rewritten as

$$\frac{-1}{2\sqrt{5}}\frac{\partial T}{\partial y_1} + \frac{5}{2\sqrt{5}(1+y_1)}\frac{\partial T}{\partial y_1} = 0$$

The following example uses **ToRec** to produce these results. Notice that the upper and lower boundary constraints have been scaled by $\sqrt{5}$ and -1 (in the problem specification), respectively, to simplify the algebra. This simplification is not material to the example.

Start of Maple Worksheet

```
> with(ptolemy,ToRec);
```

$$[\, \mathit{ToRec} \,]$$

```
> Domain := [[x1,x2], x1=0..2, x2=0..1+x1/2];
```

$$\mathit{Domain} := \left[[\, \mathit{x1}, \mathit{x2} \,], \mathit{x1} = 0..2, \mathit{x2} = 0..1 + \frac{1}{2}\,\mathit{x1} \right]$$

```
> ptolemy[PlotDomBound](Domain, scaling=constrained);
```

See Figure 3.6

```
> Laplacian := D[1,1](T) + D[2,2](T) = 0;
```

$$\mathit{Laplacian} := D_{1,1}(\,T\,) + D_{2,2}(\,T\,) = 0$$

```
> BC := {[1,LOW, T=1], [1,HIGH, T=0],
>   [2,LOW, D[2](T)=0], [2,HIGH, -D[1](T) + 2*D[2](T) = 0]};
```

$$\mathit{BC} := \Big\{ [\,1, \mathit{LOW}, T = 1\,], [\,1, \mathit{HIGH}, T = 0\,], [2, \mathit{LOW}, D_2(\,T\,) = 0]\,,$$
$$[2, \mathit{HIGH}, -D_1(\,T\,) + 2\, D_2(\,T\,) = 0] \Big\}$$

```
> TradProb := [Laplacian, BC, Domain, [T, [0,0]]];
```

$$\mathit{TradProb} := \Big[D_{1,1}(\,T\,) + D_{2,2}(\,T\,) = 0, \Big\{ [\,1, \mathit{LOW}, T = 1\,], [\,1, \mathit{HIGH}, T = 0\,],$$

Maple Worksheet Continued on Next Page

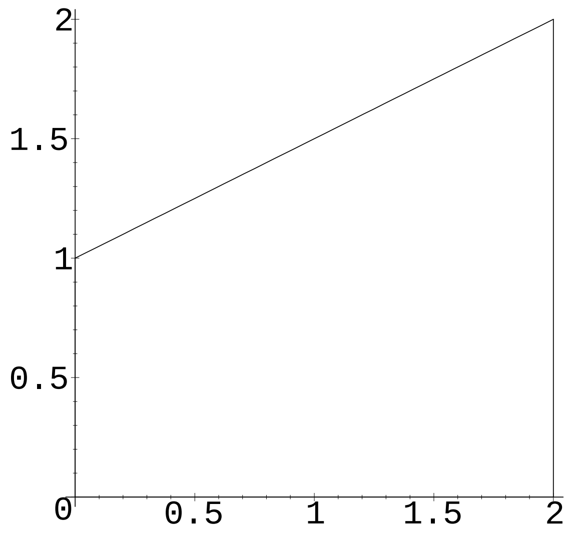

Figure 3.6: An Trapezoidal Domain

$$[2, LOW, D_2(T) = 0], [2, HIGH, -D_1(T) + 2\,D_2(T) = 0]\Big\},$$
$$\left[[x1, x2], x1 = 0..2, x2 = 0..1 + \frac{1}{2}\,x1\right], [T, [0, 0]]\Big]$$

```
> type(TradProb,ProbType);
```

true

```
> start := time(): ToRec(TradProb, [y1,y2], 'MapInfo'); time() - start;
```

$$\left[\frac{1}{4}\,\frac{(y2^2+4)\,D_{2,2}(T)}{(1+y1)^2} - \frac{1}{2}\,\frac{D_{1,2}(T)\,y2}{1+y1} + \frac{1}{4}\,D_{1,1}(T) + \frac{1}{2}\,\frac{y2\,D_2(T)}{(1+y1)^2} = 0, \Big\{\right.$$
$$[1, LOW, T = 1], [1, HIGH, T = 0], \left[2, LOW, \frac{D_2(T)}{1+y1} = 0\right],$$
$$\left[2, HIGH, \frac{5}{2}\,\frac{D_2(T)}{1+y1} - \frac{1}{2}\,D_1(T) = 0\right]\Big\}, [[y1, y2], 0..1, 0..1],$$
$$\left.[T, [0, 0]]\right]$$

4.367

```
> eval(MapInfo);
```

$$\left[(x1, x2) \rightarrow \left(\frac{1}{2}\,x1, 2\,\frac{x2}{2+x1}\right), (y1, y2) \rightarrow (2\,y1, (1+y1)\,y2)\right]$$

```
> ToRec(TradProb, [y1,y2], 'MapInfo', [-1..1, -1..1]);
```

$$\left[\frac{(y2^2+2\,y2+17)\,D_{2,2}(T)}{(3+y1)^2} - 2\,\frac{D_{1,2}(T)\,(y2+1)}{3+y1} + D_{1,1}(T)\right.$$
$$+ 2\,\frac{(y2+1)\,D_2(T)}{(3+y1)^2} = 0, \Big\{[1, LOW, T = 1], [1, HIGH, T = 0],$$
$$\left[2, HIGH, 10\,\frac{D_2(T)}{3+y1} - D_1(T) = 0\right], \left[2, LOW, 4\,\frac{D_2(T)}{3+y1} = 0\right]\Big\},$$
$$\left.[[y1, y2], -1..1, -1..1], [T, [0, 0]]\right]$$

```
> eval(MapInfo);
```

$$\left[(x1, x2) \rightarrow \left(x1 - 1, -\frac{-4\,x2 + 2 + x1}{2+x1}\right),\right.$$
$$\left.(y1, y2) \rightarrow \left(1 + y1, \frac{1}{4}\,(3+y1)\,(y2+1)\right)\right]$$

End of Maple Worksheet

Example Over Mapped Domain

This subsection illustrates the use of **ToRec** with a *MappedProbType*.

Start of Maple Worksheet

```
> with(ptolemy,ToRec);
```

$$[\,ToRec\,]$$

```
> Laplacian := D[1,1](T) + D[2,2](T) = 0;
```

$$Laplacian := D_{1,1}(\,T\,) + D_{2,2}(\,T\,) = 0$$

```
> Map := (x,y) -> (sqrt(x^2 + y^2), arctan(y,x));
```

$$Map := (\,x, y\,) \to (\,\mathrm{sqrt}(\,x^2 + y^2\,), \arctan(\,y, x\,)\,)$$

```
> InvMap := (r,theta) -> (r*cos(theta), r*sin(theta));
```

$$InvMap := (\,r, \theta\,) \to (\,r\cos(\,\theta\,), r\sin(\,\theta\,)\,)$$

```
> Domain := [[x,y],  [1/2..1, -Pi/4..Pi/4], eval(Map), eval(InvMap)];
```

$$Domain := \left[[\,x, y\,], \left[\frac{1}{2}..1, -\frac{1}{4}\pi..\frac{1}{4}\pi\right],\right.$$
$$\left.(\,x, y\,) \to (\,\mathrm{sqrt}(\,x^2 + y^2\,), \arctan(\,y, x\,)\,), (\,r, \theta\,) \to (\,r\cos(\,\theta\,), r\sin(\,\theta\,)\,)\right]$$

```
> type(Domain,MappedDomainType);
```

$$true$$

```
> ptolemy[PlotDomBound](Domain, 0..1, tickmarks=[6,7]);
```

See Figure 3.7

```
> BC := {[1,LOW, T=1], [1,HIGH, T=0],
>   [2,LOW, -D[1](T) - D[2](T) = 0], [2,HIGH, -D[1](T) + D[2](T)= 0]};
```

Maple Worksheet Continued on Next Page

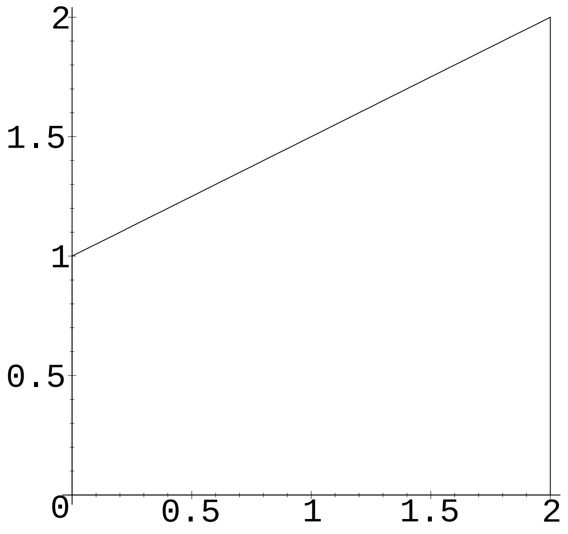

Figure 3.7: An Arc Sector Domain

$$BC := \Big\{[1, LOW, T = 1], [1, HIGH, T = 0], [2, LOW, -D_1(T) - D_2(T) = 0], [2, HIGH, -D_1(T) + D_2(T) = 0]\Big\}$$

```
> MappedProb := [Laplacian, BC, Domain, [T, [0,0]]];
```

$$MappedProb := \Big[D_{1,1}(T) + D_{2,2}(T) = 0, \Big\{[1, LOW, T = 1], [1, HIGH, T = 0], [2, LOW, -D_1(T) - D_2(T) = 0], [2, HIGH, -D_1(T) + D_2(T) = 0]\Big\}, \Big[[x, y], \Big[\frac{1}{2}..1, -\frac{1}{4}\pi..\frac{1}{4}\pi\Big], (x, y) \to (\mathrm{sqrt}(x^2 + y^2), \arctan(y, x)), (r, \theta) \to (r\cos(\theta), r\sin(\theta))\Big], [T, [0, 0]]\Big]$$

```
> type(MappedProb,ProbType);
```

$$true$$

```
> ToRec(MappedProb, [r,theta], 'MapInfo');
```

$$\Big[\frac{r^2\,(\cos(\theta)^2 + \sin(\theta)^2)\,D_1(T)}{(r^2\,(\cos(\theta)^2 + \sin(\theta)^2))^{3/2}} + D_{1,1}(T) + \frac{D_{2,2}(T)}{(\cos(\theta)^2 + \sin(\theta)^2)\,r^2} = 0, \Big\{\Big[2, HIGH, \frac{\sqrt{2}\,D_2(T)}{r} = 0\Big], \Big[2, LOW, -\frac{\sqrt{2}\,D_2(T)}{r} = 0\Big], [1, LOW, T = 1], [1, HIGH, T = 0]\Big\}, \Big[[r, \theta], \frac{1}{2}..1, -\frac{1}{4}\pi..\frac{1}{4}\pi\Big], [T, [0, 0]]\Big]$$

```
> eval(MapInfo);
```

$$[(x, y) \to (\mathrm{sqrt}(x^2 + y^2), \arctan(y, x)), (r, \theta) \to (r\cos(\theta), r\sin(\theta))]$$

```
> `ptolemy/SimpProc` := simplify;
```

$$ptolemy/SimpProc := simplify$$

```
> ToRec(MappedProb, [r,theta], 'MapInfo');
```

$$\Big[\frac{\mathrm{csgn}(r)\,D_1(T)}{r} + D_{1,1}(T) + \frac{D_{2,2}(T)}{r^2} = 0, \Big\{\Big[2, HIGH, \frac{\sqrt{2}\,D_2(T)}{r} = 0\Big], \Big[2, LOW, -\frac{\sqrt{2}\,D_2(T)}{r} = 0\Big], [1, LOW, T = 1], [1, HIGH, T = 0]\Big\}, \Big[[r, \theta], \frac{1}{2}..1, -\frac{1}{4}\pi..\frac{1}{4}\pi\Big], [T, [0, 0]]\Big]$$

```
> eval(MapInfo);
```

$$[(x, y) \to (\mathrm{sqrt}(x^2 + y^2), \arctan(y, x)), (r, \theta) \to (r\cos(\theta), r\sin(\theta))]$$

```
> assume(r>0);
> ToRec(MappedProb, [r,theta], 'MapInfo');
```

$$\left[\frac{D_1(T)}{r\tilde{}} + D_{1,1}(T) + \frac{D_{2,2}(T)}{r\tilde{}^2} = 0, \left\{\left[2, LOW, -\frac{\sqrt{2}\,D_2(T)}{r\tilde{}} = 0\right], \left[2, HIGH, \frac{\sqrt{2}\,D_2(T)}{r\tilde{}} = 0\right], [1, LOW, T = 1], [1, HIGH, T = 0]\right\}, \left[[r\tilde{}, \theta], \frac{1}{2}..1, -\frac{1}{4}\pi..\frac{1}{4}\pi\right], [T, [0, 0]]\right]$$

```
> eval(MapInfo);
```

$$[(x, y) \to (\mathrm{sqrt}(x^2 + y^2), \arctan(y, x)), (\text{`}r\tilde{}\text{`}, \theta) \to (\text{`}r\tilde{}\text{`}\cos(\theta), \text{`}r\tilde{}\text{`}\sin(\theta))]$$

```
> assume(z1>0);
> ToRec(MappedProb, [z1,z2], 'MapInfo', [0..1, 0..1]);
```

$$\left[4\frac{D_1(T)}{1 + z1\tilde{}} + 4D_{1,1}(T) + 16\frac{D_{2,2}(T)}{\pi^2(z1\tilde{}^2 + 2\,z1\tilde{} + 1)} = 0, \left\{[1, LOW, T = 1], [1, HIGH, T = 0], \left[2, LOW, -4\frac{\sqrt{2}\,D_2(T)}{\pi(1 + z1\tilde{})} = 0\right], \left[2, HIGH, 4\frac{\sqrt{2}\,D_2(T)}{\pi(1 + z1\tilde{})} = 0\right]\right\}, [[z1\tilde{}, z2], 0..1, 0..1], [T, [0, 0]]\right]$$

```
> eval(MapInfo);
```

$$\left[(x, y) \to \left(2\sqrt{x^2 + y^2} - 1, \frac{1}{2}\frac{4\arctan(y, x) + \pi}{\pi}\right), (\text{`}z1\tilde{}\text{`}, z2) \to \left(\frac{1}{2}\sin\left(\frac{1}{2}\pi\,z2 + \frac{1}{4}\pi\right)\text{`}z1\tilde{}\text{`} + \frac{1}{2}\sin\left(\frac{1}{2}\pi\,z2 + \frac{1}{4}\pi\right), -\frac{1}{2}\cos\left(\frac{1}{2}\pi\,z2 + \frac{1}{4}\pi\right)\text{`}z1\tilde{}\text{`} - \frac{1}{2}\cos\left(\frac{1}{2}\pi\,z2 + \frac{1}{4}\pi\right)\right)\right]$$

End of Maple Worksheet

Method of Implementation

The procedure first evaluates the arguments in order to extract information about both the mapped-from and the mapped-to domain. Then, if the domain type used in the problem definition is not a mapped domain the procedure constructs the map between the two domains. The procedure **MakeRecMap** or the procedure **MakeTradMap** is called for this purpose.

Then, the procedure **Warp** is called repeatedly (with the constructed mapping information) to map the governing equation and each of the boundary constraints. Next, the coordinate in the mapped-to domain that is constant along the boundary on which the constraint is applied is replaced with the appropriate value. Finally, **pde_collect** is called on each mapped boundary constraint just in case the specification of the constant coordinate allows for additional simplifications.

Dependencies

This procedure is dependent on **Warp** and **pde_collect**. If the domain of the problem is of type *RecProbType*, then the procedure also depends on **MakeRecMap**; similarly, if the domain is of type *TradDomainType* then the procedure also depends on **Make-TradMap**.

MultiToRec

This procedure maps a multidomain problem onto a collection of rectangular domains.

MultiToRec(`Prob`: *MultiProbType*, `NewCoord`: *list(name)*, `MapInfo`: *name*)
MultiToRec(`Prob`: *MultiProbType*, `NewCoord`: *list(name)*, `MapInfo`: *name*,
`OutRec`: *list({RecDomainStruct, 'DEFAULT'})*)

The argument `Prob` defines the PDE to be mapped to a collection of parallelepipeds, the argument `NewCoord` defines the coordinate names of the mapped-to domain, and the argument `MapInfo` specifies the name of the variable to be assigned to the list of maps used.

The result assigned to `MapInfo` will be of type *list(MapInfoType)*. Unlike the *MapInfoType*'s used to represent the one-dimensional "sinc maps" these *MapInfoType*s do not include the associated weights.

If the optional argument `OutRec` is provided it specifies the parallelepipeds onto which each subdomain is to be mapped. The symbol `DEFAULT` may be used in place of any output domain specifications; this indicates that the mapped-to domain for this subdomain should be the default for the subdomain type.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy,MultiToRec, PlotDomBound);
```

$$[\, MultiToRec, PlotDomBound\,]$$

```
> Domain := [[x1,x2], [x2=0..1, x1=0..2-x2], [x1=1..2, x2=2-x1..2]];
```

$$Domain := [[\,x1, x2\,], [\,x2 = 0..1, x1 = 0..2 - x2\,], [\,x1 = 1..2, x2 = 2 - x1..2\,]]$$

```
> PlotDomBound(Domain);
```

See Figure 3.8

```
> Domain2_Bad :=
>   [[x1,x2], [x2=0..1, x1=0..2-x2], [x1=1..2, x2=2-x1-sin(Pi*x1)/10..2]];
```

$$Domain2\_Bad := \Big[[\,x1, x2\,], [\,x2 = 0..1, x1 = 0..2 - x2\,], \Big[x1 = 1..2, x2 = 2 - x1 - \frac{1}{10}\sin(\,\pi\, x1\,)..2\Big]\Big]$$

```
> PlotDomBound(Domain2_Bad, scaling=constrained);
```

See Figure 3.9

```
> BC := [ {[1,LOW, V=1], [2,LOW,D[2](V)=0], [2,HIGH,D[2](V)=0]},
>   {[1,LOW, D[1](V)=1], [1,HIGH,V=0], [2,HIGH,D[2](V)=0]} ];
```

$$BC := \Big[\{[1, LOW, V = 1], [2, LOW, D_2(\,V\,) = 0], [2, HIGH, D_2(\,V\,) = 0]\},$$

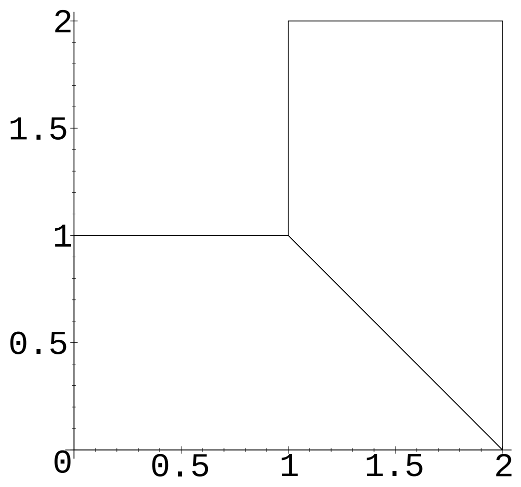

Figure 3.8: Two Subdomains of the ‘L’-Problem

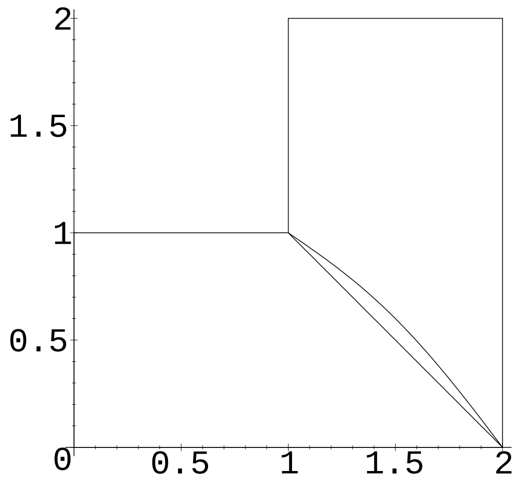

Figure 3.9: Incorrectly Specified Subdomains of the ‘L’-Problem

$$\{[2, HIGH, D_2(V) = 0], [1, LOW, D_1(V) = 1], [1, HIGH, V = 0]\}\Big]$$

```
> Laplacian := D[1,1](V) + D[2,2](V) = 0;
```

$$Laplacian := D_{1,1}(V) + D_{2,2}(V) = 0$$

```
> Prob := [Laplacian, BC, [1,1,HIGH] = [2,2,LOW], Domain, [V, [[0,0], [0,0]]]];
```

$$\begin{aligned}Prob := \Big[&D_{1,1}(V) + D_{2,2}(V) = 0, \Big[\\ &\{[1, LOW, V = 1], [2, LOW, D_2(V) = 0], [2, HIGH, D_2(V) = 0]\},\\ &\{[2, HIGH, D_2(V) = 0], [1, LOW, D_1(V) = 1], [1, HIGH, V = 0]\}\Big],\\ &[1, 1, HIGH] = [2, 2, LOW],\\ &[[x1, x2], [x2 = 0..1, x1 = 0..2 - x2], [x1 = 1..2, x2 = 2 - x1..2]],\\ &[V, [[0, 0], [0, 0]]]\Big]\end{aligned}$$

```
> type(Prob, MultiProbType);
```

$$true$$

```
> Prob_Bad := [Laplacian, BC, [1,1,HIGH] = [2,2,LOW],
>   Domain2_Bad, [V, [[0,0], [0,0]]]];
```

$$\begin{aligned}Prob\_Bad := \Big[&D_{1,1}(V) + D_{2,2}(V) = 0, \Big[\\ &\{[1, LOW, V = 1], [2, LOW, D_2(V) = 0], [2, HIGH, D_2(V) = 0]\},\\ &\{[2, HIGH, D_2(V) = 0], [1, LOW, D_1(V) = 1], [1, HIGH, V = 0]\}\Big],\\ &[1, 1, HIGH] = [2, 2, LOW], \Big[[x1, x2], [x2 = 0..1, x1 = 0..2 - x2],\\ &\Big[x1 = 1..2, x2 = 2 - x1 - \frac{1}{10}\sin(\pi\, x1)..2\Big]\Big], [V, [[0, 0], [0, 0]]]\Big]\end{aligned}$$

```
> type(Prob_Bad, MultiProbType);
```

$$true$$

```
> Start := time():
> RecProb := MultiToRec(Prob, [y1,y2], 'MapInfo');
> time() - Start;
```

$$\begin{aligned}RecProb := \Big[\Big[&\\ &\frac{(y1^2 + 1)\, D_{1,1}(V)}{(-2 + y2)^2} + D_{2,2}(V) + 2\,\frac{y1\, D_1(V)}{(-2 + y2)^2} - 2\,\frac{D_{1,2}(V)\, y1}{-2 + y2} = 0,\\ &2\,\frac{(y2 - 1)\, D_2(V)}{(y1 + 1)^2} + D_{1,1}(V) + \frac{(y2^2 - 2\, y2 + 2)\, D_{2,2}(V)}{(y1 + 1)^2}\\ &- 2\,\frac{D_{1,2}(V)\,(y2 - 1)}{y1 + 1} = 0\Big], \Big[\Big\{\Big[2, LOW, \frac{1}{2} D_1(V)\, y1 + D_2(V) = 0\Big],\\ &[2, HIGH, D_1(V)\, y1 + D_2(V) = 0], [1, LOW, V = 1]\Big\}, \Big\{\\ &[1, HIGH, V = 0], [1, LOW, D_1(V) + (-y2 + 1)\, D_2(V) = 1],\end{aligned}$$

$$\left[2, HIGH, \frac{D_2(V)}{y1+1} = 0\right]\Big\}\Big], \Big\{\Big[$$
$$[1, 1, HIGH, [2]] = [2, 2, LOW, [-1]],$$
$$\left[V = V, \frac{1}{2}\sqrt{2}\, D_2(V) - \frac{\sqrt{2}\, D_1(V)}{-2+y2} = \frac{\sqrt{2}\, D_2(V)}{y1+1} + \frac{1}{2}\sqrt{2}\, D_1(V)\right]\Big]\Big\},$$
$$[[y1, y2], [0..1, 0..1], [0..1, 0..1]], [V, [[0, 0], [0, 0]]]\Big]$$

10.534

```
> type(RecProb, MultiRecProbType);
```

true

```
> MapInfo;
```

$$\left[\left[(x1, x2) \to \left(-\frac{x1}{-2+x2}, x2\right), (y1, y2) \to (-(-2+x2)\, y1, y2)\right], \left[\right.\right.$$
$$(x1, x2) \to \left(x1 - 1, \frac{x2 - 2 + x1}{x1}\right),$$
$$\left.\left.(y1, y2) \to (y1 + 1, y2\, y1 + y2 + 1 - y1)\right]\right]$$

```
> MultiToRec(Prob_Bad, [y1,y2], 'MapInfo');
Error, (in MultiToRec) Sides don't match, [1, 1, HIGH] = [2, 2, LOW]
```

End of Maple Worksheet

Method of Implementation

The operations performed by **MultiToRec** are divided into three parts. The first is to construct maps from each subdomain to the appropriate output domain and then use these maps to map the governing equation for each domain and all of the the boundary constraints. This part is equivalent to the implementation of **ToRec**. The second part is to construct the coupling orientation information for each of the couplings. The third part is to construct the coupling equations and map each half these coupling equations according to the map for the subdomain in which that half is applied. In the final result the coupling equations for each coupling are combined with the coupling orientation information to form a *CoupleEqType*.

Coupling Orientation Information After mapping, the boundary will have dimensionality one less then the problem. For pedagogical reasons the coordinates of the boundary could be named something other than the coordinate of either mapped-to domain. For example if the mapped-to coordinate are (y_1, y_2, y_3) then the boundary coordinate might be thought of as (y_a, y_b). The coupling orientation information tells how to relate boundary coordinates to the mapped-to coordinate of each domain participating in the coupling. In terms of the example, the coupling orientation information tells how to relate (y_a, y_b) to (y_1, y_2, y_3) in each of the domains that participate in the coupling.

This is done by computing the boundary in terms of the mapped-to coordinate for each domain. In effect, this is a parameterization of the boundary in terms of the subset of the coordinates in the mapped-to domain that vary along the boundary. Each domain will yield a different parameterization, but the components of each side must be equal since there is only one side shared between two domains. For an n-dimensional problem, this yields n equations in terms of $2(n-1)$ unknowns (i.e., the coordinate of the mapped-to domain in each domain which varies over the boundary in question).

The coordinates of the mapped-to domain do not necessarily match in each subdomain that participates in the coupling. A point on the boundary with a y_1 value equal to k in one subdomain may have the y_1 of the same point equal to $1-k$ in the other subdomain, or the $y1$ values in one subdomain may match the $y2$ values in the other subdomain. This is the point that necessitates the coupling orientation information. The coupling orientation information can be constructed by solving for the y-values in one subdomain in terms of the y-values in the other subdomain.

The procedure **MultiToRec** accomplishes this by 1) using different names for the coordinate names in each subdomain, 2) using Maple's `solve` command to solve for the coordinates of the first subdomain in terms of the coordinate of the second subdomain, and 3) using the coordinate of the first domain as the coordinate of the boundary.

As pointed out previously this system is overconstrained. Nevertheless, as long the boundaries of the two subdomains actually do correspond to one "surface" in space the system will be consistent, and `solve` will be able to solve the system. If, however, the two boundaries of the two subdomains that are supposed to participate in the coupling do not correspond to the same surface then the system of constraints will be inconsistent, which will be detected by `solve`'s inability to solve the system. So a useful benefit of computing the coupling orientation information in this fashion is that the geometric "correctness" of the problem is checked as a side-effect.

Coupling Equations The coupling equations couple the subproblems defined on each subdomain so that the solution is the same as the solution to the original problem defined over the union of the subdomains. Specifically, all of the derivatives up to some order in the direction normal to the boundary must be the same on each side of the boundary. Each directional derivative (i.e., half of a coupling equation) must then be mapped to its respective output domains.

For isotropic equations, the number of coupling equations should be equal to the order of the governing equation, denoted by μ. Since the first coupling equation equates the zeroth order derivatives (i.e., the function values) this means the maximum order derivative that should be equated across the boundary is of order $\mu - 1$.

More generally, the number of coupling equations needs to be equal only to the maximum order of derivative in the governing equation which is not parallel to the boundary at atleast one point along the boundary. Typically, even for anisotropic equations this is the order of the equation. For each boundary **MultiToRec** checks that coordinates are fixed along that boundary, and then determines the maximum order derivative of each state-variable in the governing equation with respect to any nonfixed coordinate. The order of the governing equation (in each dimension) is determined once by calling **pde_order**. So determining the order of the directional derivatives that must be equated can be done in linear time as a function of both the number of dimensions and the complexity of the governing equations.

Determining which coordinates vary on a given boundary is performed by a helper procedure, **varies_over**, which returns a list of boolean values indicating whether or not the coordinate varies over the specified coordinate.

varies_over(`Coord:` *list(name)*, `Struct:` *StructType*, `DimNum:` *posint*, `End:` *EndType*)

Once the maximum order of coupled derivative has been determined, the procedure **unit_normal** is invoked to determine the direction normal to the boundary. The m^{th} order derivative in the direction $(d_1, d_2, \ldots, d_n)$ is simply

$$d_1 \frac{\partial^m V}{\partial x_1^m} + d_2 \frac{\partial^m V}{\partial x_2^m} + \ldots + d_n \frac{\partial^m V}{\partial x_n^m}$$

This expression is then mapped using the procedure **Warp** and the maps for each subdomain. The value of the one coordinate in the mapped-to domain, which is fixed on the boundary, is substituted into the result. Finally, the procedure **pde_collect** is called a second time (the first time implicitly by **Warp**) to simplify this result. These steps are performed for each domain participating in the coupling and the results are equated to form the final mapped coupling equation.

Dependencies

This procedure depends on **Warp** to map the governing equations in each domain, to map the boundary conditions, and to map each half of each coupling equation.

The procedure **pde_collect** is used in an attempt to further simplify mapped boundary constraints and mapped coupling equations after the value of the one fixed coordinate has been substituted into the result.

The procedure **pde_order** is called to determine the maximum order of the governing equation in each dimension.

Finally, the procedure **unit_normal** is called to facilitate constructing the coupling equations.

Chapter 4

Collocation on Parallelepipeds

The procedures described in this chapter perform collocation of PDEs defined over parallelepiped domains.

A significant part of this process is applying the sinc-map to the problem. The sinc-map is the portion of the overall mapping process that is designed to control the region of analyticity and the asymptotic behavior of the solution. These two factors control the rate of convergence of the approximation.

PTOLEMY defines the multidimensional "sinc-map" to be the tensor product of one-dimensional sinc-maps. The primary implication of this design decision is that the exploited region of analyticity (i.e., the portion mapped to the multidimensional analog of $\mathcal{D}_d$) has a certain "boxiness." For any single point in the real domain the portion of the exploited region whose real components match the point in question will form a parallelepiped. In addition a slice of the region of utilization that is aligned with any single complex plane will be the same no matter where the slice is taken in the region. The first restriction is usually of minor practical importance; however, the second restriction often means the exploited region of analyticity must be reduced to that dictated by the worst case slice. However, the simplification allowed by this restriction is dramatic.

Collocation is the process of substituting a parameterized form of the selected approximation into the equation and solving for the parameters which cause the equations to be satisfied at the collocation points. In order for the resulting system of constraints to be properly constrained, the system must have the same number of collocation points as the number parameters in the approximation. In order to actually solve the resulting system of equations a great deal more is required than to merely have the same number of equations as parameters.

In the case of sinc methods the approximation is always a linear combination of the bases. The result is that there is a trivial bijection between the set of bases and the set of parameters. The fact that the number of collocation points must equal the number of parameters suggests that it might be useful to pair collocation points with parameters.

For reasons based on sampling theory **PTOLEMY** always uses collocation points that map to the uniform grid (i.e., after the sinc-map is applied). Strictly interpreted, sampling theory allows for other choices of collocation points but places rather stringent limits on show much the points can spread out or bunch up in the mapped-to domain. **PTOLEMY** assumes uniform sampling because it is much simpler and more natural for the sinc bases. See [12] for an introduction to nonuniform sampling theory.

So a consequence of **PTOLEMY**'s choice of collocation points is that the sinc bases are orthonormal at the collocation points (i.e., in the discreet sense). As a result there is an extremely natural bijection between the subset of bases that are sinc-bases and most of the collocation points. This pairing of collocation points with bases can be easily extended to include all of the spline bases.

This results is a four way isomorphism between 1) collocation points, 2) unknowns in the formulation of the approximation, 3) bases, and 4) elements of the system of equation resulting from the collocation process. Comfort with this isomorphism is key to understanding the collocation process, but it can lead to some confusing descriptions, such as referring to the location of a bases or the state-variable of a collocation point. To help avoid some of this confusion without making the prose overly cumbersome, this manual introduces a new phrase, *collocation event*. Corresponding to every bases is a collocation event, but each collocation event has associated with it not only bases but also a collocation point and both an equation and an unknown in the final system of equations.

In order to improve the efficiency of symbolic manipulation it is important to group bases which behave in the same fashion with respect to the relevant symbolic manipulations (typically differentiation). This allows for the creation of a compact notation for representing and manipulating whole groups of bases. This grouping then leads to a similar grouping of collocation points, equations in the system of equations, and unknowns in the final system of equations.

Finally, for programming purposes is important to "number" (literally label with a tuple) each collocation event and all of the groups of collocation events.

LogRatioMap

This function creates a *MapInfoType* for the standard "log-ratio map."

LogRatioMap(`Low`, `High`, `X`, `Y`)
LogRatioMap(`Low`, `High`, `X`, `Y`, `Zero`)

The argument `Low` specifies the low end of the interval, `High` is the high end of the interval, `X` is the coordinate name, and `Y` is the coordinate name in the mapped-to domain. The optional argument `Zero` indicates the point that maps to zero.

Definitions

The log-ratio map is defined to be

$$\phi(x) := \ln\left(s \cdot \frac{x-\ell}{h-x}\right)$$

where the constant, s, may be thought of as the skew and the constants l and h indicate the low and high ends of the interval, respectively. It follows that

$$\phi^{-1}(z) = \frac{h \cdot e^z - s \cdot \ell}{s + e^z}$$

and that

$$\frac{1}{\phi'}(x) = \frac{(x-\ell)(h-x)}{h-\ell}$$

This particular weighting function has only one factorization

$$\text{Left Factor} = \frac{x-\ell}{h-\ell}$$
$$\text{Right Factor} = \frac{h-x}{h-\ell}$$

The point which maps to zero (called the zero-point) can be found by solving

$$\phi(x_0) = 0 \qquad \Rightarrow \qquad x_0 = \frac{h - s\cdot\ell}{1+s}.$$

This implies that

$$s = \frac{h - x_0}{x_0 - \ell}$$

From this it follows that:

- When $s = 1$ the zero-point will be in the center of the interval.
- When $s > 1$ the zero-point is to the left of the center of the interval. The larger the value of s the closer the zero-point gets to the left edge of the interval; in the limit as the skew goes to infinity the zero-point approaches the left endpoint.
- When $s < 1$ the zero-point is to the right of the center of the interval. In the limit as the s goes to zero (from the right) the zero-point becomes the right endpoint of the interval.

- When $s = c$ (for any positive constant) the zero-point is the same distance from the center of the region as when $s = 1/c$, but on the opposite side of the center.

Changing the zero-point allows the user to balance the constants associated with later truncating the left and right tail of the mapped function.

The intended use of this feature is for functions $f(x)$ such that $f(x) \to a_0(x-a)^\alpha$ as $x \to a^+$ and $f(x) \to b_0(b-x)^\alpha$ as $x \to b^-$, where $a_0 \neq b_0$. In this case when $s = 1$ the mapped function $F(z) := f(\phi^{-1}(z))$ will behave like $F(z) \to a_0 e^{\alpha z}$ as $x \to -\infty$ and $F(z) \to b_0 e^{-\alpha z}$ as $x \to \infty$. Changing s will not change the exponential rate of decay of either tail of F, but it will change the constant term.

It is possible for a specific value of N to shift the zero-point in order to balance the errors due to later truncation of the tails even when rates of decay of the tails are different, for example when $f(x) \to x^\alpha$ as $x \to a^+$, $f(x) \to x^\beta$ as $x \to b^-$ and $\alpha \neq \beta$. By setting s to a symbolic constant that is not replaced by a numerical value until near the end of the setup process it is even possible to avoid redoing most of the symbolic computations whenever N changes. However, this practice may case a loss of simplification and is considered something of a "dirty trick."

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, LogRatioMap);
```

$$[\mathit{LogRatioMap}]$$

```
> LogRatioMap(0,1, x,z);
```

$$\left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (x, 1-x)\right]$$

```
> LogRatioMap(0,1, x,z, 1/2);
```

$$\left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (x, 1-x)\right]$$

```
> LogRatioMap(-a,a, x,z);
```

$$\left[x \to \ln\left(\frac{x+a}{a-x}\right), z \to \frac{a\,e^z - a}{1+e^z}, x \to \left(\frac{1}{2}\,\frac{(x+a)\sqrt{2}}{\sqrt{a}}, \frac{1}{2}\,\frac{(a-x)\sqrt{2}}{\sqrt{a}}\right)\right]$$

```
> LogRatioMap(-a,a, x,z, a/2);
```

$$\left[x \to \ln\left(\frac{1}{3}\,\frac{x+a}{a-x}\right), z \to \frac{a\,e^z - \frac{1}{3}a}{\frac{1}{3} + e^z}, x \to \left(\frac{1}{2}\,\frac{(x+a)\sqrt{2}}{\sqrt{a}}, \frac{1}{2}\,\frac{(a-x)\sqrt{2}}{\sqrt{a}}\right)\right]$$

```
> LogRatioMap(a,b, x,z, c);
```

$$\left[x \to \ln\left(\frac{(b-c)(x-a)}{(c-a)(b-x)}\right), z \to \frac{-b\,e^z\,c + b\,e^z\,a - a\,b + a\,c}{-b + c - e^z\,c + a\,e^z},\right.$$
$$\left. x \to \left(\frac{x-a}{\sqrt{b-a}}, \frac{b-x}{\sqrt{b-a}}\right)\right]$$

End of Maple Worksheet

Mapped Region

The log-ratio map maps symmetric circular portions from above and below the plane to the the strip $\mathcal{D}_d$. The circular portion above the real line has a center at $0 + i\,C$ and a radius of R where

$$C = -\,{}^1\!/\!_2(b-a)\cot(d)$$
$$R = {}^1\!/\!_2(b-a)\csc(d)$$

These regions are illustrated in the Figure 4.1. Notice that the entire complex plane is mapped to the strip $\mathcal{D}_\pi$.

Dependencies

The procedure does not depend on any other part of the **Ptolemy** package. It does, however, use Maple's `procmake` LLF.

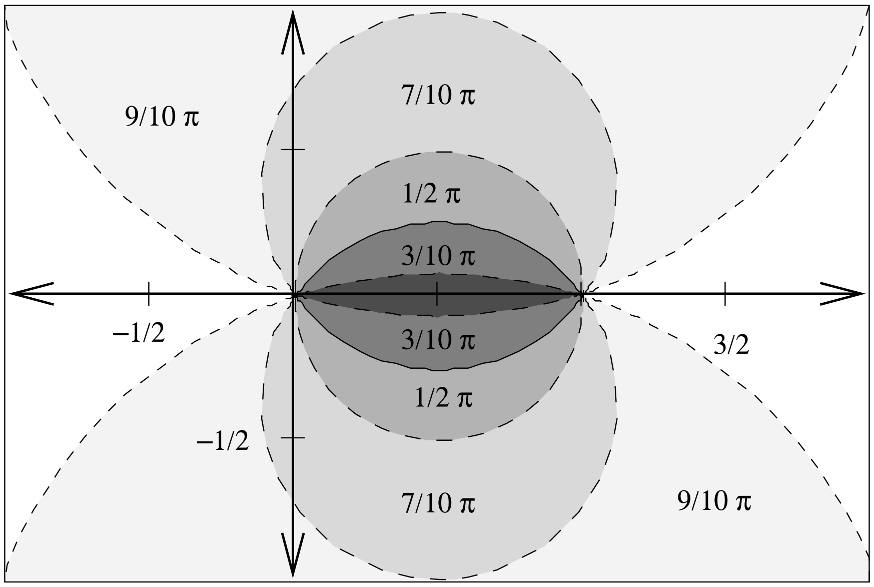

Figure 4.1: The Hierarchy of Regions Mapped by the Log-Ratio Map to the Hierarchy $\mathcal{D}_d$

LogTanMap

This function creates a *MapInfoType* for the "log-tangent map."

LogTanMap(`Low:` *:{numeric, name}*, `High:` *:{numeric, name}*, `Coord:` *:name*, `NewCoord:` *:name*)
LogTanMap(`Low:` *:{numeric, name}*, `High:` *:{numeric, name}*, `Coord:` *:name*, `NewCoord:` *:name*, `Zero:` *:{numeric, name}*)

The argument `Low` specifies the low end of the interval, `High` is the high end of the interval, `X` is the coordinate name, and `Y` is the coordinate name in the mapped-to domain. The optional argument `Zero` indicates the point that maps to zero.

Definition

The log-tangent map is defined to be

$$\phi(x) := \ln\left(s \cdot \tan\left(\frac{\pi}{2} \cdot \frac{x-\ell}{h-\ell}\right)\right),$$

where the constant s may be thought of as the skew and the constants l and h indicate the low and high ends of the interval, respectively. It follows that

$$\phi^{-1}(z) = \frac{2}{\pi}\,(h-\ell)\tan^{-1}(s\,e^{z}) + \ell$$

and that

$$\frac{1}{\phi'}(x) = \frac{2}{\pi}\,(h-\ell)\cos\left(\frac{\pi}{2} \cdot \frac{x-\ell}{h-\ell}\right)\sin\left(\frac{\pi}{2} \cdot \frac{x-\ell}{h-\ell}\right)$$

Unlike the log-ratio weighting function can be factored in many different ways. It has zeros at all of the integers. It would probably be advantageous to construct a factorization such that all of the zeros to the left of the interval are part of the left factor and all of the zeros to the right of the interval are part of the right factor. However, for reasons of symbolic simplicity this is *not* what is done by this procedure. Instead,

$$\text{Left Factor} = \sqrt{\frac{2}{\pi}\,(h-\ell)}\,\sin\left(\frac{\pi}{2} \cdot \frac{x-\ell}{h-\ell}\right)$$

$$\text{Right Factor} = \sqrt{\frac{2}{\pi}\,(h-\ell)}\,\cos\left(\frac{\pi}{2} \cdot \frac{x-\ell}{h-\ell}\right)$$

The point that maps to zero (called the zero-point) can be found by solving

$$\phi(x_0) = 0 \qquad \Rightarrow \qquad x_0 = 2/\pi\,(h-\ell)\tan^{-1}(s) + \ell$$

This in turn implies that

$$s = \tan\left(\frac{\pi}{2} \cdot \frac{x_0-\ell}{h-\ell}\right).$$

The intuitive interpretation and intended use of s is identical to that for the log-ratio map. See page 105 for a discussion of these issues.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy,LogTanMap);
```

$$[\mathit{LogTanMap}]$$

```
> LogTanMap(0,1, x,y);
```

$$\left[x \to \ln\left(\tan\left(\frac{1}{2}\pi x\right)\right),\, y \to 2\,\frac{\arctan(\mathrm{e}^{y})}{\pi},\, x \to \left(\frac{\sqrt{2}\cos\left(\frac{1}{2}\frac{\pi x}{1-x}\right)}{\sqrt{\pi}}, \frac{\sqrt{2}\cos\left(\frac{1}{2}\frac{\pi x}{1-x}\right)}{\sqrt{\pi}}\right)\right]$$

```
> LogTanMap(-1,1, x,y, -1/2);
```

$$\left[x \to \ln\left(3\tan\left(\frac{1}{4}\pi(x+1)\right)\right),\, y \to 4\,\frac{\arctan(3\,\mathrm{e}^{y}-1)}{\pi},\, x \to \left(2\,\frac{\cos\left(\frac{1}{2}\frac{\pi(x+1)}{1-x}\right)}{\sqrt{\pi}}, 2\,\frac{\cos\left(\frac{1}{2}\frac{\pi(x+1)}{1-x}\right)}{\sqrt{\pi}}\right)\right]$$

End of Maple Worksheet

Mapped Region

This map is very similar to the log-ratio map described on page 105, except that it only maps regions in the complex plane to strips $\mathcal{D}_d$ for $d \le \pi/2$. For smaller values of d the difference is that the log-tangent map uses more of the the complex plane directly above and directly below the real interval.

The domains mapped to strips of various width are shown in Figure 4.2.

For problems where the region of analyticity exploited by the log-tangent map is more desirable then the region exploited by the log-ratio map, using the log-tangent will result in a slight improvement in the rate of convergence. If the approximation error for the log-ratio map is of the form $c_r e^{-c\sqrt{N}}$ (where N is the number of sample points), and $d \le \pi/2$, then the approximation error for the log-tangent will be of the form $c_t e^{-c\sqrt{N}}$ where $c_t < c_r$. It is doubtful that this effect would be dramatic for most physically based problems. This is in part because most of the vertical strip exploited by the log-tangent map that is not exploited by the log-ratio map is mapped to regions very close to the boundary of $\mathcal{D}_{\pi/2}$. This is visualized in Figure 4.3. Even worse, when the log-ratio map can be used for $d > \pi/2$ the result will be a better exponential convergence rate than can be archived for by the log-tangent map.

The log-tangent map is included in this manual, not because of its compelling importance, but rather to illustrate the ease of adding new customized sinc-maps to **PTOLEMY**.

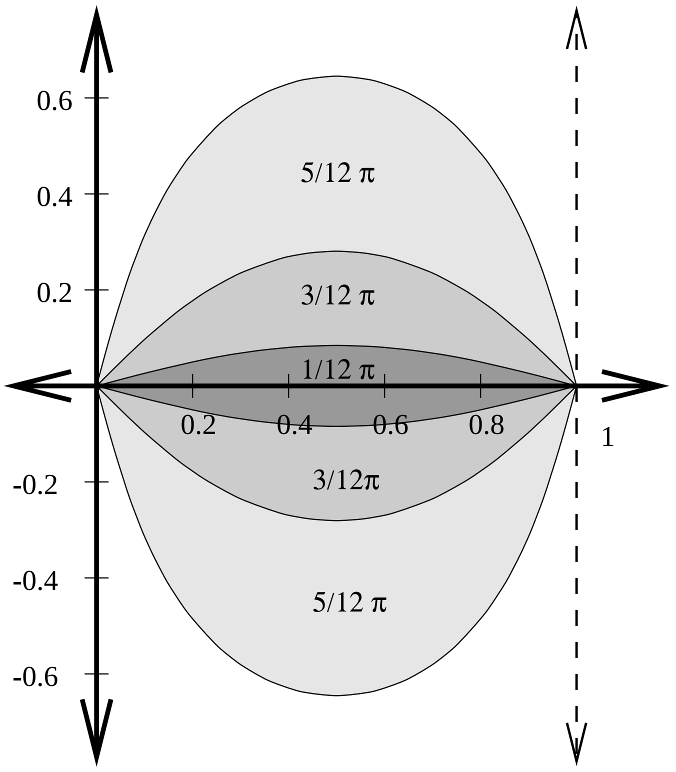

Figure 4.2: The Hierarchy of Regions Mapped by the Log-Tangent Map to the Hierarchy $\mathcal{D}_d$

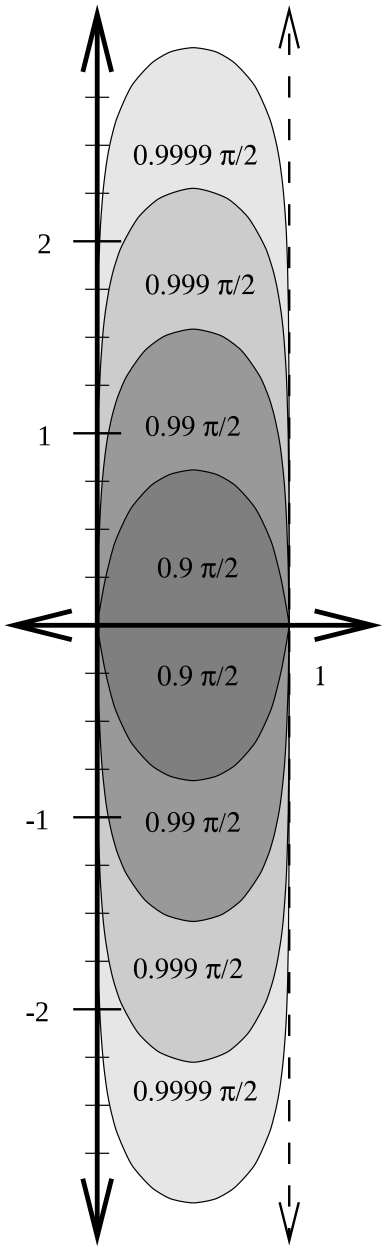

Figure 4.3: The Logarithmic Growth of $\mathcal{D}_d$ as a Function of the Size of the Mapped Region for the Log-Tangent Map

Dependencies

The procedure does not depend on any other part of the **PTOLEMY** package. It does, however, use Maple's `procmake` LLF.

LogSinhMap

This procedure creates a *MapInfoType* for various forms of the "log-sinh map."

LogSinhMap(`Edge:` {*numeric, name*}, `End:` *EndType*, `X:` *name*, `Z:` *name*)
LogSinhMap(`Edge:` {*numeric, name*}, `End:` *EndType*, `X:` *name*, `Z:` *name*,
`DistToZero:` {*algebraic*})

The argument `Edge` specifies the location of the edge of the semiinfinite interval (which is the domain of the map). The `End` argument specifies rather `Edge` specifies the low end or the high end of the interval. The arguments `X` and `Y` are the coordinate names in the original domain and the mapped-to domain, respectively.

The optional argument `DistToZero` specifies the distance between the edge of the interval and the point that maps to zero. If this value is not specified it defaults to $\sinh^{-1}(1) \approx 0.8814$. Maximal symbolic simplicity of the result occurs when this argument is some whole number multiple of $\sinh^{-1}(1)$.

Definitions

When `End = LOW`, the log-sinh map is defined as

$$\phi(x) = \ln\big(\sinh\big(\gamma(x-a)\big)\big) \tag{4.1}$$

where $\gamma := \sinh^{-1}(1)/\ell$, a is the location of the edge, and ℓ is the distance between the point that maps to zero and the edge.

From this definition it follows that

$$\phi^{-1}(z) = (1/\gamma)\sinh^{-1}(e^{z}) + a$$

and that

$$\frac{1}{\phi'}(x) = (1/\gamma)\tanh\big(\gamma(x-a)\big)$$

Just as in the case of the log-tangent map there is more than one way to factor this weighting function. However, unlike the weight for the log-tangent map there appears to be only one reasonable factorization

$$\text{Left Factor} = \frac{1}{\sqrt{\gamma}}\sinh\big(\gamma(x-a)\big)$$
$$\text{Right Factor} = \frac{1}{\sqrt{\gamma}}\operatorname{sech}\big(\gamma(x-a)\big)$$

All of these equations assume that $\ell \in R^{+}$.

All of these formula can be used irrespective of rather the semiinfinite interval is negative going or positive going, but these formula probably should not be used for negative going intervals (i.e., when `Edge = HIGH`). This is because when a is actually the high edge of the interval the equations map a to negative infinity (instead of positive infinity). This causes a confusing inversion of the low and high ends.

So when `Edge = HIGH` the procedure uses the definition

$$\phi(x) := -\ln\left(\sinh\left(\frac{\sinh^{-1}(1)}{\ell}\cdot(b-x)\right)\right)$$

where b is now used to represent the edge. (So just as in the **LogRatioMap** a represents the low end of the interval and b represents the high end of the interval, but here sometimes the low end is the edge and sometimes the high end is). Letting $\hat{\gamma} := \sinh^{-1}/(b - x_0)$ reduces this to a form similar to Equation 4.1, i.e.,

$$\phi(x) := -\ln\big(\sinh\big(\hat{\gamma}\,(b - x)\big)\big), \tag{4.2}$$

The inverse of this map is

$$\phi^{-1}(z) = b - (1/\hat{\gamma})\sinh^{-1}(e^z)$$

and the resulting nullifier function is

$$\frac{1}{\phi'}(x) = (1/\hat{\gamma})\tanh\big(\hat{\gamma}(b - x)\big)$$

This minus sign on Equation 4.2 can be removed by converting the `sinh` function to a `csch` function. However, since both `ln(x)` and `csch(x)` have singularities at $x = 0$ Maple struggles to correctly evaluate and symbolically manipulate $\ln \circ \operatorname{csch}$ near the origin. In some cases the symbolic problems can be overcome by using the `assume` mechanism. Because this mechanism is relatively week and expressions derived from the sinc-maps are extensively manipulated by **PTOLEMY** this solution is explicitly not recommended. So although the use `csch` and `arcccsh` makes the expressions appear cleaner, the result is much less useful.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, LogSinhMap);
```

$$[\mathit{LogSinhMap}]$$

```
> LogSinhMap(0,LOW, x,z);
```

$$[x \to \ln(\sinh(x)), z \to \operatorname{arcsinh}(e^z), x \to (\sinh(x), \operatorname{sech}(x))]$$

```
> LogSinhMap(0,LOW, x,z, 1);
```

$$\Bigg[x \to \ln(\sinh(\operatorname{arcinh}(1)\,x)), z \to \frac{\operatorname{arcsinh}(e^z)}{\operatorname{arcinh}(1)}, x \to \\ \left(\sqrt{\frac{1}{\operatorname{arcinh}(1)}}\sinh(\operatorname{arcinh}(1)\,x), \sqrt{\frac{1}{\operatorname{arcinh}(1)}}\operatorname{sech}(\operatorname{arcinh}(1)\,x)\right)\Bigg]$$

```
> LogSinhMap(1,HIGH, x,z);
```

$$[x \to -\ln(-\sinh(-1 + x)), z \to 1 - \operatorname{arcsinh}(e^{(-z)}), \\ x \to (\sinh(1 - x), \operatorname{sech}(1 - x))]$$

```
> LogSinhMap(A,LOW, x,z, B);
```

$$\left[x \to \ln\left(\sinh\left(\frac{\operatorname{arcinh}(1)(x-A)}{B}\right)\right), z \to \frac{\operatorname{arcsinh}(e^z)B}{\operatorname{arcinh}(1)} + A, x \to \left(\sqrt{\frac{B}{\operatorname{arcinh}(1)}}\sinh\left(\frac{\operatorname{arcinh}(1)(x-A)}{B}\right), \sqrt{\frac{B}{\operatorname{arcinh}(1)}}\operatorname{sech}\left(\frac{\operatorname{arcinh}(1)(x-A)}{B}\right)\right)\right]$$

```
> LogSinhMap(B,HIGH, x,z, A);
```

$$\left[x \to -\ln\left(\sinh\left(\frac{\operatorname{arcinh}(1)(B-x)}{A}\right)\right), z \to B - \frac{\operatorname{arcsinh}(e^{(-z)})A}{\operatorname{arcinh}(1)}, x \to \left(\sqrt{\frac{A}{\operatorname{arcinh}(1)}}\sinh\left(\frac{\operatorname{arcinh}(1)(B-x)}{A}\right), \sqrt{\frac{A}{\operatorname{arcinh}(1)}}\operatorname{sech}\left(\frac{\operatorname{arcinh}(1)(B-x)}{A}\right)\right)\right]$$

End of Maple Worksheet

Mapped Regions

The log-sinh map maps what is sometimes referred to as a bullet shaped region to strips about the real line. These regions are illustrated in Figure 4.4.

Dependencies

The procedure does not depend on any other part of the **PTOLEMY** package. It does, however, use Maple's `procmake` LLF.

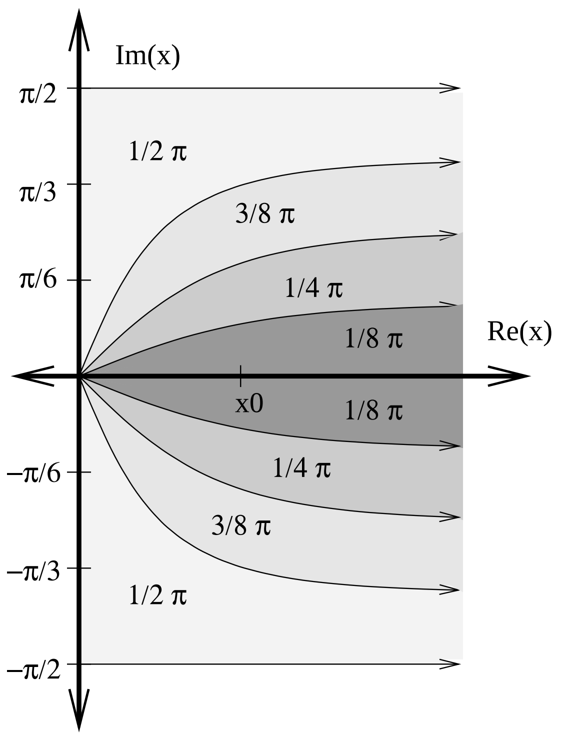

Figure 4.4: The Hierarchy of Regions Mapped by the Log-Sinh Map to the Hierarchy $\mathcal{D}_d$

CollocateRec

This procedure applies collocation to a PDE defined over a parallelepiped domain.

CollocateRec(`Prob`: *RecProbType*, `NewCoord`: *list(name)*, `ExtraBases`: *name*)
CollocateRec(`Prob`: *RecProbType*, `NewCoord`: *list(name)*, `ExtraBases`: *name*,
`MapInfp`: *list(MapInfoType)*)

This procedure converts the problem specified by the argument `Prob` to `SB`-notation. This conversion is equivalent to performing collocation. A key part of this process is mapping the domain to an infinite domain. The argument `NewCoord` specifies the coordinate names in the mapped-to domain. The argument `ExtraBases` specifies the name of a variable that will be assigned a doubly indexed table of extra bases functions. The first index of this table will be the state-variable name and the second index will be an integer indicating the dimension. The entry will be a list of the extra bases components for this dimension.

The procedure **CollocateRec** does not apply any weighting. Weighting only affects the conditioning of the problem, not the solution. As a result, it seems more appropriate to apply weighting when the problem is converted to a matrix equation. (See the section titled "SToLinKron" on page 146 for a discussion of converting `SB`-notation to a matrix equation.)

A Simple Example

Consider Poisson's equation where V is the state-variable, x_1 and x_2 are the coordinates, and k is a constant,

$$\frac{\partial^2 V}{\partial x_1^2} + \frac{\partial^2 V}{\partial x_2^2} = K$$

Assume that this equation is applied over the region $[0, W] \times [0, H]$ and that $V = 0$ on the boundary of the region and that only a zeroth order approximation is required (i.e., an approximation to the solution, but not an approximation to the solution's derivatives, is required).

A reasonable choice of sinc maps is

$$z_1 = \phi_1(x_1) = \ln\left(\frac{x_1}{W - x_1}\right) \quad \text{and} \quad z_2 = \phi_2(x_2) = \ln\left(\frac{x_2}{H - x_2}\right)$$

The inverses of these maps are

$$x_1 = \frac{We^{z_1}}{1 + e^{z_1}} \quad \text{and} \quad x_2 = \frac{He^{z_1}}{1 + e^{z_1}}.$$

The sinc bases in each dimension are

$$\operatorname{sinc}(k, h_1) \circ \phi_1(x_1) \quad \text{and} \quad \operatorname{sinc}(\ell, h_2) \circ \phi_2(x_2)$$

where $k \in \{-N_1, \ldots, N_1\}$ and $\ell \in \{-N_2, \ldots, N_2\}$.

Because the boundary conditions are homogeneous there is no need for any spline bases in this example. If the boundary constraints had been $V = f(x_1, x_2)$ then spline bases would be required. In both cases the boundary constraints are zeroth order. **PTOLEMY** could check to see if the boundary constraint implies that any of the derivatives are of the state-variables are identically zero along any boundary and avoid using the corresponding splines. For zeroth order constraints this is relatively easy, but for systems of higher order constraints this requires quite a bit of extra code. The value of this complication is limited because: 1) a few unnecessary boundary splines do not increase the order of the final matrix system significantly, 2) in realistic engineering problems homogeneous constraints are reasonably uncommon, and 3) when symbolic constants are used within the constraints it is often not possible to determine rather a particular spline bases is necessary without knowing the values of the symbolic constants. As a result this version of **PTOLEMY** only uses the *order of the boundary constraints* for determining the spline bases. Sometimes, as in this example, this causes a small amount of extra work, but it is always sufficient.

So for this problem the spline bases in the x_1 dimension are $1 - x_1$ and x_1 and the spline bases in the x_2 dimension are $1 - x_2$ and x_2. The boundary constraints will cause the coefficients associated with these bases to be identically zero, but **PTOLEMY** still includes them in the problem formulation.

The resulting approximation of V is:

$$\begin{aligned}
\tilde{V} = {} & C[-(N_1+1), -(N_2+1)](W - x_1)(H - x_2) + \\
& \sum_{k=-N_1}^{N_1} C[k, -(N_2+1)]\big(\operatorname{sinc}(k, h_1) \circ \phi_1(x_1)\big)(H - x_2) + \\
& C[N_1+1, -(N_2+1)]x_1(H - x_2) + \\
& \sum_{\ell=-N_2}^{N_2} C[-(N_1+1), \ell](W - x_1)\big(\operatorname{sinc}(\ell, h_2) \circ \phi_2(x_2)\big) + \\
& \sum_{k=-N_1}^{N_1} \sum_{\ell=-N_2}^{N_2} C[k, l]\big(\operatorname{sinc}(k, h_1) \circ \phi_1(x_1)\big)\big(\operatorname{sinc}(\ell, h_2) \circ \phi_2(x_2)\big) + \\
& \sum_{\ell=-N_2}^{N_2} C[N_1+1, \ell]x_1\big(\operatorname{sinc}(\ell, h_2) \circ \phi_2(x_2)\big) + \\
& C[-(N_1+1), N_2+1] \cdot (W - x_1)x_2 + \\
& \sum_{k=-N_1}^{N_1} C[k, N_2+1]\big(\operatorname{sinc}(k, h_1) \circ \phi_1(x_1)\big)x_2 + \\
& C[N_1+1, N_2+1] \cdot x_1 x_2
\end{aligned}$$

where C is a $(2N_1 + 3) \times (2N_2 + 3)$ array of constants indexed over $-(N_1 + 1)..N_1 + 1$

by $-(N_2+1)..N_2+1$. The derivative of $\tilde{V}$ with respect to x_1 is

$$\begin{aligned}
\frac{\partial^2 \tilde{V}}{\partial x_1^2} = & -C[-(N_1+1),-(N_2+1)]\cdot(H-x_2)+\\
& \sum_{k=-N_1}^{N_1} C[k,-(N_2+1)]\phi'(x_1)\big(\operatorname{sinc}'(k,h_1)\circ\phi_1(x_1)\big)(H-x_2)+\\
& C[N_1+1,-(N_2+1)](H-x_2)+\\
& \sum_{\ell=-N_2}^{N_2} -C[-(N_1+1),\ell]\big(\operatorname{sinc}(\ell,h_2)\circ\phi_2(x_2)\big)+\\
& \sum_{k=-N_1}^{N_1}\sum_{\ell=-N_2}^{N_2} C[k,l]\phi'(x_1)\big(\operatorname{sinc}'(k,h_1)\circ\phi_1(x_1)\big)(\operatorname{sinc}(\ell,h_2)\circ\phi_2(x_2))+\\
& \sum_{\ell=-N_2}^{N_2} C[N_1+1,\ell]\big(\operatorname{sinc}(\ell,h_2)\circ\phi_2(x_2)\big)-\\
& C[-(N_1+1),N_2+1]x_2+\\
& \sum_{k=-N_1}^{N_1} C[k,N_2+1]\phi'(x_1)\big(\operatorname{sinc}'(k,h_1)\circ\phi_1(x_1)\big)x_2+\\
& C[N_1+1,N_2+1]x_2
\end{aligned}$$

where ϕ' and sinc' represent the derivatives of ϕ and sinc. Similarly the second derivative is

$$\begin{aligned}
\frac{\partial \tilde{V}}{\partial x_1} = & \sum_{k=-N_1}^{N_1} C[k,-(N_2+1)](\phi'(x_1))^2(\operatorname{sinc}''(k,h_1)\circ\phi_1(x_1))(H-x_2)+\\
& \sum_{k=-N_1}^{N_1} C[k,-(N_2+1)]\phi''(x_1)(\operatorname{sinc}'(k,h_1)\circ\phi_1(x_1))(H-x_2)+\\
& \sum_{k=-N_1}^{N_1}\sum_{\ell=-N_2}^{N_2} C[k,l](\phi'(x_1))^2(\operatorname{sinc}''(k,h_1)\circ\phi_1(x_1))(\operatorname{sinc}(\ell,h_2)\circ\phi_2(x_2))+\\
& \sum_{k=-N_1}^{N_1}\sum_{\ell=-N_2}^{N_2} C[k,l]\phi''(x_1)(\operatorname{sinc}'(k,h_1)\circ\phi_1(x_1))(\operatorname{sinc}(\ell,h_2)\circ\phi_2(x_2))+\\
& \sum_{k=-N_1}^{N_1} C[k,-N_2-1](\phi_1'(x_1))^2(\operatorname{sinc}''(k,h_1)\circ\phi_1(x_1))x_2+\\
& \sum_{k=-N_1}^{N_1} C[k,-N_2-1]\phi_1''(x_1)(\operatorname{sinc}'(k,h_1)\circ\phi_1(x_1))x_2
\end{aligned} \tag{4.3}$$

The derivatives of ϕ_1 are

$$\phi_1'(x_1)=\frac{W}{x_1(W-x_1)} \quad \text{and} \quad \phi_1''(x_1)=\frac{W(W-2x_1)}{x_1^2(W-x_1)^2}$$

Substituting $x_1 = \phi_1^{-1}(z_1)$ into this equation yields

$$\begin{aligned}\phi_1'(x_1) &= \frac{W}{x_1(W-x_1)} = \frac{(1+e^{z_1})^2}{W\,e^{2z_1}} \\ \phi_1''(x_1) &= \frac{W(W-2x_1)}{x_1^2(W-x_1)^2} = \frac{(1+e^{z_1})^3(1-e^{z_1})}{W\,e^{2z_1}}\end{aligned} \tag{4.4}$$

Substituting Equation 4.4, z_1 for $\phi_1(x_1)$, and z_2 for $\phi_2(x_2)$ into Equation 4.3 yields:

$$\begin{aligned}\frac{\partial\tilde{V}}{\partial x_1} = &\sum_{k=-N_1}^{N_1} C[k,-(N_2+1)]\frac{(1+e^{z_1})^4\,\mathrm{sinc}''(k,h_1)(z_1)}{W^2e^{2z_1}}\cdot\frac{H}{1+e^{z_2}} + \\ &\sum_{k=-N_1}^{N_1} C[k,-(N_2+1)]\frac{(1+e^{z_1})^3(1-e^{z_1})\,\mathrm{sinc}'(k,h_1)(z_1)}{W^2e^{2z_1}}\frac{H}{1+e^{z_2}} + \\ &\sum_{k=-N_1}^{N_1}\sum_{\ell=-N_2}^{N_2} C[k,l]\frac{(1+e^{z_1})^4\,\mathrm{sinc}''(k,h_1)}{W^2e^{2z_1}}\mathrm{sinc}(\ell,h_2)(z_2) + \\ &\sum_{k=-N_1}^{N_1}\sum_{\ell=-N_2}^{N_2} C[k,l]\frac{(1+e^{z_1})^3(1-e^{z_1})\,\mathrm{sinc}'(k,h_1)(z_1)}{W^2e^{2z_1}}\cdot\mathrm{sinc}(\ell,h_2)(z_2) + \\ &\sum_{k=-N_1}^{N_1} C[k,N_2+1]\cdot\frac{(1+e^{z_1})^4\,\mathrm{sinc}''(k,h_1)(z_1)}{W^2e^{2z_1}}\cdot\frac{H\,e^{z_2}}{1+e^{z_2}} + \\ &\sum_{k=-N_1}^{N_1} C[k,N_2+1]\cdot\frac{(1+e^{z_1})^4\,\mathrm{sinc}''(k,h_1)(z_1)}{W^2e^{2z_1}}\cdot\frac{H\,e^{z_2}}{1+e^{z_2}}\end{aligned}$$

The expression for $\partial^2\tilde{V}/\partial x_2^2$ is similar.

In addition to the governing equation, the problem's boundary constraints must be incorporated into the collocation equations. The boundary constraints are handled by overriding the constraints derived from the governing equation with special equations at the "boundary collocation points." In this case, this means that $\tilde{V}$ must be zero along the outer ring of collocation points.

Consider collocation point number $(-(N_1+1), -(N_2+1))$. All of the (two-dimensional) bases are zero at this point except the bases $(1-x_1)(1-x_2)$, $x_1(1-x_2)$, $(1-x_1)x_2$, and $x_1\,x_2$. So the "override equation" at this point is:

$$\begin{aligned}&C[-(N_1+1),-(N_2+1)](1-x_1)(1-x_2) + \\ &\quad C[N_1+1,-(N_2+1)]x_1(1-x_2) + \\ &\quad C[-(N_1+1),N_2+1](1-x_1)x_2 + C[N_1+1,N_2+1]x_1\,x_2 = 0\end{aligned} \tag{4.5}$$

Of the four nonzero bases, the $x_1(1-x_2)$, $(1-x_1)x_2$, and $x_1\,x_2$ are very close to zero at this point. As a result, it is not uncommon to approximate Equation 4.5 with

$$(1-x_1)(1-x_2) = 0.$$

PTOLEMY allows this type of approximation to the full collocation to be used by providing an optional argument to **collocate_bound**. However, this is not the default behavior; none of the examples in this section include this additional approximation. Equation 4.5 will also be the "override equation" for the points $(N_1+1, -(N_2+1))$, $(-(N_1+1), N_2+1)$, and (N_1+1, N_2+1).

Now consider the points numbered $(-(N_1+1), -N_2), \ldots, (-(N_1+1), N_2)$. At each of these points the only nonzero (two-dimensional) bases are:

$$\begin{gathered}(1-x_1)(1-x_2), \quad x_1(1-x_2), \quad (1-x_1)x_2, \quad x_1\,x_2,\\ (1-x_1)\operatorname{sinc}(\ell, h_2)\circ\phi(x_2), \quad \text{and} \quad x_1\operatorname{sinc}(\ell, h_2)\circ\phi(x_2)\end{gathered}$$

for $\ell \in \{-N_2, \ldots, N_2\}$. So the "overriding equation" will be:

$$\begin{aligned}&C[-(N_1+1), -(N_2+1)](1-x_1)(1-x_2) +\\ &\quad C[N_1+1, -(N_2+1)]x_1(1-x_2) +\\ &\quad C[-(N_1+1), N_2+1](1-x_1)x_2 + C[N_1+1, N_2+1]x_1\,x_2 +\\ &\quad \sum_{\ell=-N_2}^{N_2} C[-N_1-1, \ell](1-x_1)\operatorname{sinc}(\ell, h_2)\circ\phi(x_2) +\\ &\quad \sum_{\ell=-N_2}^{N_2} C[-N_1-1, \ell]x_1\operatorname{sinc}(\ell, h_2)\circ\phi(x_2) = 0\end{aligned} \tag{4.6}$$

This is also the override equation for the points $(N_1+1, -N_2), \ldots, (N_1+1, N_2)$. However, a different but strictly analogous equation is applied at the points along the upper and lower edges of the domain.

The following example uses **CollocateRec** to compute this result. Especially notice how compact the SB-notation is and that the computation takes of the order of 10 seconds on an average, circa 1994-95, workstation-class machine.

Start of Maple Worksheet

```
> with(ptolemy, CollocateRec);
```

$$[\,CollocateRec\,]$$

```
> Sys := D[1,1](V) + D[2,2](V) = K;
```

$$Sys := D_{1,1}(\,V\,) + D_{2,2}(\,V\,) = K$$

```
> Bound := {[1,LOW, V=0], [1,HIGH, V=0], [2,LOW, V=0], [2,HIGH, V=0]};
```

$$\begin{aligned}&Bound := \{[1, LOW, V=0], [1, HIGH, V=0], [2, LOW, V=0],\\ &\quad [2, HIGH, V=0]\}\end{aligned}$$

```
> Domain := [[x1,x2], 0..1, 0..1];
```

$$Domain := [\,[\,x1, x2\,], 0..1, 0..1\,]$$

```
> OrderInfo := {[V, [0,0]]};
```

$$OrderInfo := \{\,[\,V, [\,0, 0\,]\,]\,\}$$

```
> Prob := [Sys, Bound, Domain, OrderInfo];
```

$$Prob := \Big[D_{1,1}(\,V\,) + D_{2,2}(\,V\,) = K, \{[1, LOW, V = 0], [1, HIGH, V = 0], [2, LOW, V = 0], [2, HIGH, V = 0]\}, [[\,x1, x2\,], 0..1, 0..1], \{[\,V, [0, 0]\,]\}\Big]$$

```
> type(Prob, RecProbType);
```

$$true$$

```
> start := time():
> SProb := CollocateRec(Prob, [z1,z2], 'ExtraBases');
```

$$SProb := \Bigg[\frac{D^{(2)}(\,V\_S1\,)\,(\,1 + e^{z1}\,)^4\,V\_S2}{(\,e^{z1}\,)^2} + \frac{D(\,V\_S1\,)\,(\,e^{z1} - 1\,)\,(\,1 + e^{z1}\,)^3\,V\_S2}{(\,e^{z1}\,)^2} + \frac{D^{(2)}(\,V\_S1\,)\,(\,1 + e^{z1}\,)^4\,\%1}{(\,e^{z1}\,)^2} + \frac{D(\,V\_S1\,)\,(\,e^{z1} - 1\,)\,(\,1 + e^{z1}\,)^3\,\%1}{(\,e^{z1}\,)^2} + \frac{V\_S1\,D^{(2)}(\,V\_S2\,)\,(\,1 + e^{z2}\,)^4}{(\,e^{z2}\,)^2} + \frac{V\_S1\,D(\,V\_S2\,)\,(\,e^{z2} - 1\,)\,(\,1 + e^{z2}\,)^3}{(\,e^{z2}\,)^2} + \frac{\%2\,D^{(2)}(\,V\_S2\,)\,(\,1 + e^{z2}\,)^4}{(\,e^{z2}\,)^2} + \frac{\%2\,D(\,V\_S2\,)\,(\,e^{z2} - 1\,)\,(\,1 + e^{z2}\,)^3}{(\,e^{z2}\,)^2} = K, \{$$

$$[\%2\;V\_S2 + \%2\,\%1 = 0, V, [-1, 0]], [\%2\,\%1 = 0, V, [1, 1]], [\%2\,\%1 + V\_S1\,\%1 = 0, V, [0, 1]], [\%2\,\%1 = 0, V, [-1, -1]], [\%2\,\%1 + V\_S1\,\%1 = 0, V, [0, -1]], [\%2\,\%1 = 0, V, [-1, 1]], [\%2\;V\_S2 + \%2\,\%1 = 0, V, [1, 0]], [\%2\,\%1 = 0, V, [1, -1]]\},$$

$$[\,z1, z2\,], \Bigg[\bigg[x1 \to \ln\left(\frac{x1}{1 - x1}\right), z1 \to \frac{e^{z1}}{1 + e^{z1}}, x1 \to (\,x1, 1 - x1\,)\bigg], \bigg[x2 \to \ln\left(\frac{x1}{1 - x1}\right), z2 \to \frac{e^{z1}}{1 + e^{z1}}, x2 \to (\,x1, 1 - x1\,)\bigg]\Bigg], [[\,V, [0, 0]\,]]\Bigg]$$

$$\%1 := \mathrm{V\_B2}\left(\frac{1}{1 + e^{z2}}, \frac{e^{z2}}{1 + e^{z2}}\right)$$

$$\%2 := \mathrm{V\_B1}\left(\frac{1}{1 + e^{z1}}, \frac{e^{z1}}{1 + e^{z1}}\right)$$

```
> time() - start;
```

$$8.450$$

```
> eval(ExtraBases);
```

table([
$(\,V, 1\,) = [\,1 - x1, x1\,]$
$(\,V, 2\,) = [\,1 - x2, x2\,]$
])

```
> type(SProb, SProbType);
```

$$true$$

End of Maple Worksheet

This same example can be easily redone with a different choice of map. Trying many different maps on the same problem would be prohibitive if setup were being done by hand. Notice that the computational time is more than slightly longer. This is due to the use of the "more heavyweight" simplification procedure. Also notice that whereas the result is more complicated it is still surprisingly similar. Each coefficient is a low order rational function in terms of e^{z_1} and/or e^{z_2}. This form is the direct result of the asymptotics of the sinc-map which is the same in both examples.

Start of Maple Worksheet

```
> with(ptolemy, CollocateRec, LogTanMap);
```

$$[\,CollocateRec, LogTanMap\,]$$

```
> Sys := D[1,1](V) + D[2,2](V) = K;
```

$$Sys := D_{1,1}(\,V\,) + D_{2,2}(\,V\,) = K$$

```
> Bound := {[1,LOW, V=0], [1,HIGH, V=0], [2,LOW, V=0], [2,HIGH, V=0]};
```

$$Bound := \{[\,1, LOW, V = 0\,], [\,1, HIGH, V = 0\,], [\,2, LOW, V = 0\,], [\,2, HIGH, V = 0\,]\}$$

```
> Domain := [[x1,x2], 0..1, 0..1];
```

$$Domain := [\,[\,x1, x2\,], 0..1, 0..1\,]$$

```
> OrderInfo := {[V, [0,0]]};
```

$$OrderInfo := \{\,[\,V, [\,0, 0\,]\,]\,\}$$

```
> Prob := [Sys, Bound, Domain, OrderInfo];
```

$$Prob := \Big[D_{1,1}(\,V\,) + D_{2,2}(\,V\,) = K, \{[\,1, LOW, V = 0\,], [\,1, HIGH, V = 0\,], [\,2, LOW, V = 0\,], [\,2, HIGH, V = 0\,]\}, [\,[\,x1, x2\,], 0..1, 0..1\,], \{\,[\,V, [\,0, 0\,]\,]\,\}\Big]$$

```
> type(Prob, RecProbType);
```

$$true$$

```
> Map1 := LogTanMap(0,W, x1,z1);
```

$$Map1 := \left[x1 \to \ln\left(\tan\left(\frac{1}{2}\,\frac{\pi\,x1}{\mathrm{W}}\right)\right), z1 \to 2\,\frac{\mathrm{W}\arctan(\,\mathrm{e}^{z1}\,)}{\pi}, x1 \to \left(\sqrt{2}\sqrt{\frac{\mathrm{W}}{\pi}}\cos\left(\frac{1}{2}\,\frac{\pi\,x1}{\mathrm{W}-x1}\right), \sqrt{2}\sqrt{\frac{\mathrm{W}}{\pi}}\cos\left(\frac{1}{2}\,\frac{\pi\,x1}{\mathrm{W}-x1}\right)\right)\right]$$

```
> Map2 := LogTanMap(0,H, x2,z2);
```

$$Map2 := \left[x2 \to \ln\left(\tan\left(\frac{1}{2}\,\frac{\pi\,x2}{H}\right)\right),\, z2 \to 2\,\frac{H\arctan(\,e^{z2}\,)}{\pi},\right.$$
$$\left. x2 \to \left(\sqrt{2}\,\sqrt{\frac{H}{\pi}}\cos\left(\frac{1}{2}\,\frac{\pi\,x2}{H-x2}\right),\, \sqrt{2}\,\sqrt{\frac{H}{\pi}}\cos\left(\frac{1}{2}\,\frac{\pi\,x2}{H-x2}\right)\right)\right]$$

```
> `ptolemy/SimpProc` := simplify;
```

$$ptolemy/SimpProc := simplify$$

```
> Start := time():
> SProb := CollocateRec(Prob, [z1,z2], 'ExtraBases', [Map1,Map2]);
```

$$SProb := \left[\frac{1}{4}\,\frac{(\,1+e^{(\,2\,z1\,)}\,)^2\,\pi^2\,e^{(\,-2\,z1\,)}\,\%2\,D^{(\,2\,)}(\,V\_S1\,)}{W^2}\right.$$
$$-\,\frac{1}{4}\,\frac{D(\,V\_S1\,)\,(\,1+e^{(\,2\,z1\,)}\,)\,\pi^2\,(\,-1+e^{(\,-2\,z1\,)}\,)\,\%2}{W^2}$$
$$+\,\frac{1}{4}\,\frac{(\,1+e^{(\,2\,z2\,)}\,)^2\,\pi^2\,e^{(\,-2\,z2\,)}\,D^{(\,2\,)}(\,V\_S2\,)\,V\_S1}{H^2}$$
$$-\,\frac{1}{4}\,\frac{V\_S1\,D(\,V\_S2\,)\,(\,1+e^{(\,2\,z2\,)}\,)\,\pi^2\,(\,-1+e^{(\,-2\,z2\,)}\,)}{H^2}$$
$$+\,\frac{1}{4}\,\frac{(\,1+e^{(\,2\,z2\,)}\,)^2\,\pi^2\,e^{(\,-2\,z2\,)}\,D^{(\,2\,)}(\,V\_S2\,)\,\%1}{H^2}$$
$$-\,\frac{1}{4}\,\frac{\%1\,D(\,V\_S2\,)\,(\,1+e^{(\,2\,z2\,)}\,)\,\pi^2\,(\,-1+e^{(\,-2\,z2\,)}\,)}{H^2}$$
$$+\,\frac{1}{4}\,\frac{(\,1+e^{(\,2\,z1\,)}\,)^2\,\pi^2\,e^{(\,-2\,z1\,)}\,V\_S2\,D^{(\,2\,)}(\,V\_S1\,)}{W^2}$$
$$-\,\frac{1}{4}\,\frac{D(\,V\_S1\,)\,(\,1+e^{(\,2\,z1\,)}\,)\,\pi^2\,(\,-1+e^{(\,-2\,z1\,)}\,)\,V\_S2}{W^2}$$
$$+\,V\_B1\left(-\frac{1}{2}\,\%3,\,\frac{1}{2}\,\%3\right)\,\%2 + \%1\,V\_B2\left(-\frac{1}{2}\,\%4,\,\frac{1}{2}\,\%4\right)$$
$$+\,V\_S1\,V\_B2\left(-\frac{1}{2}\,\%4,\,\frac{1}{2}\,\%4\right) + V\_B1\left(-\frac{1}{2}\,\%3,\,\frac{1}{2}\,\%3\right)\,V\_S2 = K, \{$$
$$[\,\%1\,\%2 + V\_S1\,\%2 = 0, V, [\,0,1\,]\,], [\,\%1\,\%2 = 0, V, [\,-1,1\,]\,],$$
$$[\,\%1\,\%2 + V\_S1\,\%2 = 0, V, [\,0,-1\,]\,], [\,\%1\,\%2 = 0, V, [\,1,1\,]\,],$$
$$[\,\%1\,\%2 = 0, V, [\,-1,-1\,]\,], [\,\%1\,\%2 = 0, V, [\,1,-1\,]\,],$$
$$[\,\%1\,\%2 + \%1\;V\_S2 = 0, V, [\,-1,0\,]\,],$$
$$[\,\%1\,\%2 + \%1\;V\_S2 = 0, V, [\,1,0\,]\,]\}, [\,z1, z2\,], \Big[\Big[$$
$$x1 \to \ln\left(\tan\left(\frac{1}{2}\,\frac{\pi\,x1}{W}\right)\right),\, z1 \to 2\,\frac{W\arctan(\,e^{z1}\,)}{\pi},$$
$$x1 \to \left(\sqrt{2}\,\sqrt{\frac{W}{\pi}}\cos\left(\frac{1}{2}\,\frac{\pi\,x1}{W-x1}\right),\, \sqrt{2}\,\sqrt{\frac{W}{\pi}}\cos\left(\frac{1}{2}\,\frac{\pi\,x1}{W-x1}\right)\right)\Big],\,\Big[$$
$$x2 \to \ln\left(\tan\left(\frac{1}{2}\,\frac{\pi\,x2}{H}\right)\right),\, z2 \to 2\,\frac{H\arctan(\,e^{z1}\,)}{\pi},$$
$$x2 \to \left(\sqrt{2}\,\sqrt{\frac{H}{\pi}}\cos\left(\frac{1}{2}\,\frac{\pi\,x2}{H-x2}\right),\, \sqrt{2}\,\sqrt{\frac{H}{\pi}}\cos\left(\frac{1}{2}\,\frac{\pi\,x2}{H-x2}\right)\right)\Big]\Big],$$

$$[[V,[0,0]]]\Big]$$

$$\%1 := \mathrm{V\_B1}\left(\frac{1}{1+e^{z1}}, \frac{e^{z1}}{1+e^{z1}}\right)$$

$$\%2 := \mathrm{V\_B2}\left(\frac{1}{1+e^{z2}}, \frac{e^{z2}}{1+e^{z2}}\right)$$

$$\%3 := \frac{\pi^2\,(e^{z1}-1)\,(1+e^{(2\,z1)})}{\mathrm{W}^2\,(1+3\,e^{z1}+3\,e^{(2\,z1)}+e^{(3\,z1)})}$$

$$\%4 := \frac{\pi^2\,(e^{z2}-1)\,(1+e^{(2\,z2)})}{H^2\,(1+3\,e^{z2}+3\,e^{(2\,z2)}+e^{(3\,z2)})}$$

```
> time() - Start;
```

$$13.066$$

End of Maple Worksheet

Less Trivial Boundary Constraints

The example in this subsection solves the same problem but with a mixture of zeroth order and first order boundary constraints. Specifically, $\nabla V \cdot \hat{n} = 1$ along the left and right edge. Although this problem may be physically meaningless, it is straightforward to collocate. It illustrates several ways in which the complexity of this problem grows compared to the previous problem.

Notice that the order of the actual approximation must be greater than the requested order of the approximation in order to accurately approximate the partial with respect to x_1 of the solution on the left and right edges of the domain. This causes the sinc bases to be slightly more complex, but more significantly, it causes the boundary splines to be higher order. These higher order boundary splines yield both more complicated terms and more terms in the governing equation than in the previous example. Also, the corner regions now have two different override equations.

In spite of all theses extra complications the operation still takes roughly the same amount of time. This is because the bulk of the time is spent simplifying intermediate results and the amount of effort required for this part of the problem is not substantially different in the three examples of this section.

Start of Maple Worksheet

```
> with(ptolemy, CollocateRec, LogRatioMap);
```

$$[\,\mathit{CollocateRec}, \mathit{LogRatioMap}\,]$$

```
> Sys := D[1,1](V) + D[2,2](V) = K;
```

$$\mathit{Sys} := D_{1,1}(\,V\,) + D_{2,2}(\,V\,) = K$$

```
> BC :=
>   [[1,LOW, D[1](V) = 1], [1,HIGH, D[1](V) = -1],
>    [2,LOW, V=x1*(1-x1)], [2,HIGH, V=sin(Pi*x1)/Pi]];
```

$$BC := \Big[[1, LOW, D_1(V) = 1], [1, HIGH, D_1(V) = -1], [2, LOW, V = x1\,(1 - x1)], \Big[2, HIGH, V = \frac{\sin(\pi\, x1)}{\pi}\Big]\Big]$$

$$BC := \Big[[1, LOW, D_1(V) = 1], [1, HIGH, D_1(V) = -1], [2, LOW, V = 0], [2, HIGH, V = 0]\Big]$$

```
> Coords := [x1,x2]; NewCoords := [z1,z2];
```

$$Coords := [x1, x2]$$

$$NewCoords := [z1, z2]$$

```
> Domain := [[x1,x2], 0..1, 0..1];
```

$$Domain := [[x1, x2], 0..1, 0..1]$$

```
> OrderInfo := {[V, [0,0]]};
```

$$OrderInfo := \{[V, [0, 0]]\}$$

```
> Prob := [Sys, BC, Domain, OrderInfo];
```

$$Prob := \Big[D_{1,1}(V) + D_{2,2}(V) = K, \Big[[1, LOW, D_1(V) = 1], [1, HIGH, D_1(V) = -1], [2, LOW, V = x1\,(1 - x1)], \Big[2, HIGH, V = \frac{\sin(\pi\, x1)}{\pi}\Big]\Big], [[x1, x2], 0..1, 0..1], \{[V, [0, 0]]\}\Big]$$

```
> type(Prob, RecProbType);
```

$$true$$

```
> start := time():
> SProb := CollocateRec(Prob, NewCoords, 'ExtraBases');
```

$$SProb := \Big[-2\ V\_S1\ V\_S2 - 2\ V\_S1\ \%1 + \mathrm{V\_B1}\left(6\,\frac{e^{z1}-1}{1+e^{z1}}, 2\,\frac{-2+e^{z1}}{1+e^{z1}}, -6\,\frac{e^{z1}-1}{1+e^{z1}}, 2\,\frac{-1+2\,e^{z1}}{1+e^{z1}}\right)\%1 + \mathrm{V\_B1}\left(6\,\frac{e^{z1}-1}{1+e^{z1}}, 2\,\frac{-2+e^{z1}}{1+e^{z1}}, -6\,\frac{e^{z1}-1}{1+e^{z1}}, 2\,\frac{-1+2\,e^{z1}}{1+e^{z1}}\right)V\_S2 + \frac{\%2\,\mathrm{D}(V\_S2)\,(e^{z2}-1)\,(1+e^{z2})^3}{(e^{z2})^2} + \frac{(1+e^{z1})^2\,\%1\,D^{(2)}(V\_S1)}{e^{z1}} - \frac{(e^{z1}-1)\,(1+e^{z1})\,\%1\,\mathrm{D}(V\_S1)}{e^{z1}} + \frac{e^{z1}\,(1+e^{z2})^4\,D^{(2)}(V\_S2)\,V\_S1}{(1+e^{z1})^2\,(e^{z2})^2} + \frac{e^{z1}\,(e^{z2}-1)\,(1+e^{z2})^3\,\mathrm{D}(V\_S2)\,V\_S1}{(1+e^{z1})^2\,(e^{z2})^2} + \frac{\%2\,D^{(2)}(V\_S2)\,(1+e^{z2})^4}{(e^{z2})^2} + \frac{(1+e^{z1})^2\,V\_S2\,D^{(2)}(V\_S1)}{e^{z1}} - \frac{(e^{z1}-1)\,(1+e^{z1})\,V\_S2\,\mathrm{D}(V\_S1)}{e^{z1}} =$$

$$K, \left\{ [\%3\,\%1 + \%3\; V\_S2 = -1, V, [2,0]], \right.$$

$$\left[\%2\,\%1 = \frac{\sin\left(\frac{\pi\, e^{z1}}{1+e^{z1}}\right)}{\pi}, V, [-1,1] \right],$$

$$[\%3\,\%1 + \%3\; V\_S2 = 1, V, [-2,0]],$$

$$\left[\%2\,\%1 + \frac{e^{z1}\,\%1\; V\_S1}{(1+e^{z1})^2} = \frac{e^{z1}}{(1+e^{z1})^2}, V, [0,-1] \right],$$

$$\left[\%2\,\%1 = \frac{e^{z1}}{(1+e^{z1})^2}, V, [1,-1] \right], [\%3\,\%1 = -1, V, [2,-1]],$$

$$[\%3\,\%1 = 1, V, [-2,1]], \left[\%2\,\%1 = \frac{\sin\left(\frac{\pi\, e^{z1}}{1+e^{z1}}\right)}{\pi}, V, [1,1] \right],$$

$$[\%3\,\%1 = -1, V, [2,1]],$$

$$\left[\%2\,\%1 + \frac{e^{z1}\,\%1\; V\_S1}{(1+e^{z1})^2} = \frac{\sin\left(\frac{\pi\, e^{z1}}{1+e^{z1}}\right)}{\pi}, V, [0,1] \right],$$

$$\left. [\%3\,\%1 = 1, V, [-2,-1]], \left[\%2\,\%1 = \frac{e^{z1}}{(1+e^{z1})^2}, V, [-1,-1] \right] \right\},$$

$$[z1, z2], \left[\left[x1 \to \ln\left(\frac{x1}{1-x1}\right), z1 \to \frac{e^{z1}}{1+e^{z1}}, x1 \to (x1, 1-x1) \right], \right.$$

$$\left. \left[x2 \to \ln\left(\frac{x1}{1-x1}\right), z2 \to \frac{e^{z1}}{1+e^{z1}}, x2 \to (x1, 1-x1) \right] \right], [[V, [1,0]]] \Bigg]$$

$$\%1 := \mathrm{V\_B2}\left(\frac{1}{1+e^{z2}}, \frac{e^{z2}}{1+e^{z2}}\right)$$

$$\%2 := \mathrm{V\_B1}\left(\frac{3\,e^{z1}+1}{(1+e^{z1})^3}, \frac{e^{z1}}{(1+e^{z1})^3}, \frac{(e^{z1})^2\,(3+e^{z1})}{(1+e^{z1})^3}, -\frac{(e^{z1})^2}{(1+e^{z1})^3}\right)$$

$$\%3 := \mathrm{V\_B1}\left(-6\,\frac{e^{z1}}{(1+e^{z1})^2}, -\frac{-1+2\,e^{z1}}{(1+e^{z1})^2}, 6\,\frac{e^{z1}}{(1+e^{z1})^2}, \frac{e^{z1}\,(-2+e^{z1})}{(1+e^{z1})^2}\right)$$

```
> time() - start;
```

$$11.833$$

```
> type(SProb, SProbType);
```

$$\mathit{true}$$

```
> eval(ExtraBases);
```

table([

$$(V, 1) =$$

$$\left[(2\,x1+1)\,(-1+x1)^2, x1\,(-1+x1)^2, -x1^2\,(-3+2\,x1), x1^2\,(-1+x1)\right]$$

$$(V, 2) = [1-x2, x2]$$

])

End of Maple Worksheet

Method of Implementation

The procedure first determines the minimum order of approximation required to both apply all of the boundary conditions and meet the minimum requested order of approximation. This is done by calling **required_order**. In doing this, the procedure is careful to consider only the order of the derivatives occurring in boundary constraints that are perpendicular to the boundary.

Next, **CollocateRec** calls the procedure **collocate_main** to collocate the governing equation. Then it calls the procedure **collocate_bound** to form sets of collocated boundary constraints. Finally, it calls **assign_bound** on each set of override equations to assign each equation to a specific group of collocation points.

Dependencies

If maps are not explicitly specified via the optional argument `MapInfo` this procedure will call **LogRatioMap** to construct the maps.

The procedure will call **required_order** to determine what order of approximation is required in order to ensure that all of the boundary constraints are well approximated. It then calls **make_bases** to construct the main bases and the extra bases for each domain.

Finally, the procedures **collocate_main** and **collocate_bound** are called to perform the primary collocation operation.

CollocateMultiRec

This procedure is equivalent to **CollocateRec** except that it requires a problem of type *RecMultiProbType*, instead of *RecProbType*.

CollocateMultiRec(`Prob`: *RecMultiProbType*, `NewCoord`: *list(name)*,
`ExtraBases`: *name*)
CollocateMultiRec(`Prob`: *RecMultiProbType*, `NewCoord`: *list(name)*,
`ExtraBases`: *name*, `MapInfo`: *list(list(MapInfoType))*)

This procedure converts the problem specified by the argument `Prob` to `SB`-notation. This conversion is equivalent to collocation. As part of this process each rectangular domain must be mapped to an infinite domain. The argument `NewCoord` specifies the coordinate names in the mapped-to domain. The argument `ExtraBases` specifies the name of a variable that will be assigned a doubly index table of extra bases functions. The first index of this table will be integers corresponding to the "domain number," i.e., the sequence number within the list of subdomains.

If the optional argument `MapInfo` is specified then it specifies the mapping to be used. The outer list ranges over the subdomains and the inner list ranges over the dimensions. If the map is not explicitly specified the procedure will construct the required log-ratio-map.

This procedure does not apply any weighting. Weighting affects only the conditioning of the problem, not the solution. As a result, it seemed more appropriate to apply weighting when the problem is converted to a matrix equation. (See the section titled "MultiSToLinKron" on page 152 for information on conversion to a matrix equation.)

The L-Problem Example

Continuing the example from the section titled "MultiToRec" on page 96; the author hand checked this result and found a bug not "exploited" by any previously tested examples. However, hand checking this result is extremely tedious that can take a full day of effort; for most readers it is not a recommended exercise.

Start of Maple Worksheet

```
> with(ptolemy,MultiToRec);
```

$$[\mathit{MultiToRec}]$$

```
> Domain1 := [[x1,x2], x2=0..1, x1=0..2-x2];
```

$$\mathit{Domain1} := [[\,x1, x2\,], x2 = 0..1, x1 = 0..2 - x2\,]$$

```
> Domain2 := [[x1,x2], x1=1..2, x2=2-x1..2];
```

$$\mathit{Domain2} := [[\,x1, x2\,], x1 = 1..2, x2 = 2 - x1..2\,]$$

```
> BC := [ {[1,LOW, V=1], [2,LOW,D[2](V)=0], [2,HIGH,D[2](V)=0]},
>   {[1,LOW, D[1](V)=0], [2,HIGH,D[2](V)=0], [1,HIGH,V=0]} ];
```

$$BC := \Big[\{[1, LOW, V = 1], [2, LOW, D_2(\,V\,) = 0], [2, HIGH, D_2(\,V\,) = 0]\},$$

$$\{[1, HIGH, V = 0], [2, HIGH, D_2(V) = 0], [1, LOW, D_1(V) = 0]\}\Big]$$

```
> Laplacian := D[1,1](V) + D[2,2](V) = 0;
```

$$Laplacian := D_{1,1}(V) + D_{2,2}(V) = 0$$

```
> Prob := [Laplacian, BC, [1,1,HIGH] = [2,2,LOW],
>   [Domain1,Domain2], [V, [[0,0], [0,0]]]];
```

$$Prob := \Big[D_{1,1}(V) + D_{2,2}(V) = 0, \Big[$$
$$\{[1, LOW, V = 1], [2, LOW, D_2(V) = 0], [2, HIGH, D_2(V) = 0]\},$$
$$\{[1, HIGH, V = 0], [2, HIGH, D_2(V) = 0], [1, LOW, D_1(V) = 0]\}\Big],$$
$$[1, 1, HIGH] = [2, 2, LOW], [[[x1, x2], x2 = 0..1, x1 = 0..2 - x2],$$
$$[[x1, x2], x1 = 1..2, x2 = 2 - x1..2]], [V, [[0, 0], [0, 0]]]\Big]$$

```
> type(Prob,MultiProbType);
```

$$true$$

```
> RecProb := MultiToRec(Prob, [[y1,y2], [y1,y2]], 'MapInfo');
```

$$RecProb := \Big[\Big[$$
$$-2\,\frac{y1\,D_{1,2}(V)}{-2 + y2} + \frac{(1 + y1^2)\,D_{1,1}(V)}{(-2 + y2)^2} + 2\,\frac{y1\,D_1(V)}{(-2 + y2)^2} + D_{2,2}(V) = 0,$$
$$-2\,\frac{(y2 - 1)\,D_{1,2}(V)}{y1 + 1} + D_{1,1}(V) + 2\,\frac{(y2 - 1)\,D_2(V)}{(y1 + 1)^2}$$
$$+ \frac{(2 + y2^2 - 2\,y2)\,D_{2,2}(V)}{(y1 + 1)^2} = 0\Big], \Big[\Big\{$$
$$\Big[2, LOW, \frac{1}{2}\,D_1(V)\,y1 + D_2(V) = 0\Big],$$
$$[2, HIGH, D_1(V)\,y1 + D_2(V) = 0], [1, LOW, V = 1]\Big\}, \Big\{$$
$$[1, LOW, (1 - y2)\,D_2(V) + D_1(V) = 0], \Big[2, HIGH, \frac{D_2(V)}{y1 + 1} = 0\Big],$$
$$[1, HIGH, V = 0]\Big\}\Big], \Big\{\Big[[1, 1, HIGH, [2]] = [2, 2, LOW, [-1]],$$
$$\Big[V = V, \frac{1}{2}\,\sqrt{2}\,D_2(V) - \frac{\sqrt{2}\,D_1(V)}{-2 + y2} = \frac{\sqrt{2}\,D_2(V)}{y1 + 1} + \frac{1}{2}\,\sqrt{2}\,D_1(V)\Big]\Big]\Big\},$$
$$[[y1, y2], [0..1, 0..1], [0..1, 0..1]], [V, [[0, 0], [0, 0]]]\Big]$$

```
> type(RecProb, RecMultiProbType);
```

$$true$$

```
> MapInfo;
```

$$\Big[\Big[(x1, x2) \to \Big(-\frac{x1}{-2 + x2}, x2\Big), (y1, y2) \to (-(-2 + x2)\,y1, y2)\Big], \Big[$$

$$(\,x1, x2\,) \to \left(x1 - 1, \frac{x2 - 2 + x1}{x1}\right),$$

$$(\,y1, y2\,) \to (\,y1 + 1, y2\; y1 + y2 + 1 - y1\,)\Big]\Big]$$

```
> with(ptolemy, CollocateMultiRec);
```

$$[\,\mathit{CollocateMultiRec}\,]$$

```
> Start := time():
> SProb := CollocateMultiRec(RecProb,[z1,z2], ’ExtraBases’):
```

$$\mathit{SProb} := \Big[\Big[- \frac{\left(-\mathrm{e}^{z1} - 2\,(\,\mathrm{e}^{z1}\,)^2 + 2\,(\,\mathrm{e}^{z1}\,)^3 - 1\right)(\,1 + \mathrm{e}^{z2}\,)^2\,\%1\,\mathrm{D}(\,V\_S1\,)}{(\,1 + \mathrm{e}^{z1}\,)\,\mathrm{e}^{z1}\,(\,2 + \mathrm{e}^{z2}\,)^2}$$

$$+ 2\,\frac{\mathrm{e}^{z1}\,(\,1 + \mathrm{e}^{z2}\,)\,\%3\,\mathrm{D}(\,V\_S1\,)}{(\,1 + \mathrm{e}^{z1}\,)\,(\,2 + \mathrm{e}^{z2}\,)} + 2\,\frac{\mathrm{e}^{z1}\,(\,1 + \mathrm{e}^{z2}\,)\,\%2\,\%3}{(\,1 + \mathrm{e}^{z1}\,)\,(\,2 + \mathrm{e}^{z2}\,)} + \%4\,\%5$$

$$+ 2\,\frac{\mathrm{e}^{z1}\,(\,1 + \mathrm{e}^{z2}\,)\,\mathrm{D}(\,V\_S2\,)\,\%2}{(\,1 + \mathrm{e}^{z1}\,)\,(\,2 + \mathrm{e}^{z2}\,)} + \frac{(\,1 + \mathrm{e}^{z2}\,)^2\,D^{(\,2\,)}(\,V\_S2\,)\,\%4}{\mathrm{e}^{z2}}$$

$$- \frac{(\,\mathrm{e}^{z2} - 1\,)\,(\,1 + \mathrm{e}^{z2}\,)\,\%4\,\mathrm{D}(\,V\_S2\,)}{\mathrm{e}^{z2}}$$

$$- \frac{\mathrm{e}^{z1}\,\left(-2 + (\,\mathrm{e}^{z2}\,)^2 - \mathrm{e}^{z2} + 2\,\mathrm{e}^{z1}\,\mathrm{e}^{z2}\right)(\,1 + \mathrm{e}^{z2}\,)\,\mathrm{D}(\,V\_S2\,)\;V\_S1}{(\,1 + \mathrm{e}^{z1}\,)^2\,\mathrm{e}^{z2}\,(\,2 + \mathrm{e}^{z2}\,)}$$

$$+ \frac{\mathrm{e}^{z2}\,\%8\,D^{(\,2\,)}(\,V\_S1\,)\;V\_S2}{(\,2 + \mathrm{e}^{z2}\,)^2\,\mathrm{e}^{z1}} + \frac{\mathrm{e}^{z1}\,(\,1 + \mathrm{e}^{z2}\,)^2\,D^{(\,2\,)}(\,V\_S2\,)\;V\_S1}{\mathrm{e}^{z2}\,(\,1 + \mathrm{e}^{z1}\,)^2}$$

$$- 2\,\frac{\mathrm{e}^{z1}\,(\,\mathrm{e}^{z1} - 1\,)\,(\,1 + \mathrm{e}^{z2}\,)\,\%3\;V\_S1}{(\,1 + \mathrm{e}^{z1}\,)^2\,(\,2 + \mathrm{e}^{z2}\,)} + \frac{\mathrm{e}^{z1}\,\%5\;V\_S1}{(\,1 + \mathrm{e}^{z1}\,)^2}$$

$$+ \frac{\mathrm{e}^{z2}\,\%8\,\%7\;V\_S2}{(\,2 + \mathrm{e}^{z2}\,)^2\,(\,1 + \mathrm{e}^{z1}\,)^2} - 2\,\frac{\left((\,\mathrm{e}^{z2}\,)^2 - 2\right)\,\mathrm{e}^{z1}\,\%2\;V\_S2}{(\,2 + \mathrm{e}^{z2}\,)^2\,(\,1 + \mathrm{e}^{z1}\,)} -$$

$$\left(-\mathrm{e}^{z2} - 4\,(\,\mathrm{e}^{z1}\,)^2 - \mathrm{e}^{z1}\,\mathrm{e}^{z2} + 2\,(\,\mathrm{e}^{z1}\,)^2\,(\,\mathrm{e}^{z2}\,)^2 + 2\,(\,\mathrm{e}^{z1}\,)^3\,\mathrm{e}^{z2}\right)\,\mathrm{D}(\,V\_S1\,)\;V\_S2$$

$$\Big/\left((\,2 + \mathrm{e}^{z2}\,)^2\,\mathrm{e}^{z1}\,(\,1 + \mathrm{e}^{z1}\,)\right) + 2\,\frac{\mathrm{e}^{z1}\,(\,1 + \mathrm{e}^{z2}\,)\,\mathrm{D}(\,V\_S2\,)\,\mathrm{D}(\,V\_S1\,)}{(\,1 + \mathrm{e}^{z1}\,)\,(\,2 + \mathrm{e}^{z2}\,)}$$

$$- 2\,\%4\;V\_S2 + 2\,\frac{\mathrm{e}^{z1}\,(\,1 + \mathrm{e}^{z2}\,)^2\,\%2\,\%1}{(\,2 + \mathrm{e}^{z2}\,)^2\,(\,1 + \mathrm{e}^{z1}\,)}$$

$$- 2\,\frac{\left(1 + \mathrm{e}^{z1} + 3\,(\,\mathrm{e}^{z1}\,)^2\right)(\,1 + \mathrm{e}^{z2}\,)^2\;V\_S1\,\%1}{(\,2 + \mathrm{e}^{z2}\,)^2\,(\,1 + \mathrm{e}^{z1}\,)^2} + 2\,\Big($$

$$-2\,(\,\mathrm{e}^{z1}\,)^2\,\mathrm{e}^{z2} + (\,\mathrm{e}^{z1}\,)^2\,(\,\mathrm{e}^{z2}\,)^2 - 2\,\mathrm{e}^{z1} - 2\,(\,\mathrm{e}^{z1}\,)^2 - 6\,\mathrm{e}^{z1}\,\mathrm{e}^{z2} - 2\,(\,\mathrm{e}^{z2}\,)^2\,\mathrm{e}^{z1} - \mathrm{e}^{z2}$$

$$\Big)\;V\_S2\;V\_S1\Big/\left((\,2 + \mathrm{e}^{z2}\,)^2\,(\,1 + \mathrm{e}^{z1}\,)^2\right) + \frac{(\,1 + \mathrm{e}^{z2}\,)^2\,\%8\,\%1\,D^{(\,2\,)}(\,V\_S1\,)}{(\,2 + \mathrm{e}^{z2}\,)^2\,\mathrm{e}^{z1}}$$

$$+ \frac{(\,1 + \mathrm{e}^{z2}\,)^2\,\%8\,\%7\,\%1}{(\,2 + \mathrm{e}^{z2}\,)^2\,(\,1 + \mathrm{e}^{z1}\,)^2} = 0,$$

$$- \frac{\left(\mathrm{e}^{z1}\,\mathrm{e}^{z2} + 2\,(\,\mathrm{e}^{z1}\,)^2\,\mathrm{e}^{z2} - 2\,\mathrm{e}^{z1} - \mathrm{e}^{z2}\right)(\,1 + \mathrm{e}^{z1}\,)\,\mathrm{D}(\,V\_S1\,)\;V\_S2}{(\,1 + \mathrm{e}^{z2}\,)^2\,\mathrm{e}^{z1}\,(\,2\,\mathrm{e}^{z1} + 1\,)}$$

$$- 2\,\frac{(\,1 + \mathrm{e}^{z1}\,)^2\,\%4\,\%3}{(\,1 + \mathrm{e}^{z2}\,)\,(\,2\,\mathrm{e}^{z1} + 1\,)^2} + 2\,\frac{(\,1 + \mathrm{e}^{z1}\,)\,\%3\,\mathrm{D}(\,V\_S1\,)}{(\,2\,\mathrm{e}^{z1} + 1\,)\,(\,1 + \mathrm{e}^{z2}\,)}$$

$$- 2\,\frac{\left(2\,(\,\mathrm{e}^{z1}\,)^2 - 1\right)\,\%3\;V\_S1}{(\,1 + \mathrm{e}^{z2}\,)\,(\,2\,\mathrm{e}^{z1} + 1\,)^2} -$$

$$\left(-2\,e^{z1}-2\,e^{z2}+4\,(e^{z1})^2\,e^{z2}+(e^{z2})^2\,e^{z1}+(e^{z2})^3\,e^{z1}\right)\mathrm{D}(V\_S2)\,V\_S1\Big/\Big($$
$$(2\,e^{z1}+1)^2\,(1+e^{z2})\,e^{z2}\Big)-2\,\frac{\left(3+e^{z2}+(e^{z2})^2\right)(1+e^{z1})^2\,\%4\,V\_S2}{(1+e^{z2})^2\,(2\,e^{z1}+1)^2}$$
$$+\frac{(1+e^{z1})^2\,e^{z2}\,V\_S2\,D^{(2)}(V\_S1)}{e^{z1}\,(1+e^{z2})^2}+\frac{e^{z1}\,\%6\,D^{(2)}(V\_S2)\,V\_S1}{(2\,e^{z1}+1)^2\,e^{z2}}$$
$$-2\,\frac{(e^{z2}-1)(1+e^{z1})\,\%2\,V\_S2}{(1+e^{z2})^2\,(2\,e^{z1}+1)}+\frac{e^{z1}\,\%6\,\%5\,V\_S1}{(1+e^{z2})^2\,(2\,e^{z1}+1)^2}$$
$$+\frac{(1+e^{z1})^2\,\%6\,D^{(2)}(V\_S2)\,\%4}{(2\,e^{z1}+1)^2\,e^{z2}}+\frac{e^{z2}\,V\_S2\,\%7}{(1+e^{z2})^2}$$
$$+\frac{(1+e^{z1})^2\,\%1\,D^{(2)}(V\_S1)}{e^{z1}}-\frac{(e^{z1}-1)(1+e^{z1})\,\%1\,\mathrm{D}(V\_S1)}{e^{z1}}$$
$$-\frac{\left(-2+2\,e^{z2}+(e^{z2})^2+(e^{z2})^3\right)(1+e^{z1})^2\,\mathrm{D}(V\_S2)\,\%4}{(1+e^{z2})\,e^{z2}\,(2\,e^{z1}+1)^2}$$
$$+2\,\frac{(1+e^{z1})\,\%2\,\%3}{(2\,e^{z1}+1)(1+e^{z2})}+2\,\frac{(1+e^{z1})\,\mathrm{D}(V\_S2)\,\%2}{(2\,e^{z1}+1)(1+e^{z2})}+\%7\,\%1$$
$$-2\,V\_S1\,\%1+2\,\frac{(1+e^{z1})\,\mathrm{D}(V\_S2)\,\mathrm{D}(V\_S1)}{(2\,e^{z1}+1)(1+e^{z2})}$$
$$+\frac{(1+e^{z1})^2\,\%6\,\%4\,\%5}{(1+e^{z2})^2\,(2\,e^{z1}+1)^2}-2$$
$$\left(-1+2\,e^{z1}+2\,e^{z2}+2\,(e^{z1})^2+6\,e^{z1}\,e^{z2}+2\,(e^{z1})^2\,e^{z2}+(e^{z2})^2\,e^{z1}\right)V\_S2$$
$$V\_S1\Big/\left((1+e^{z2})^2\,(2\,e^{z1}+1)^2\right)=0\Big],\Big[\Big\{$$
$$\left[\frac{\%2\,\%1\,e^{z1}}{1+e^{z1}}+\%3\,\%4=0,V,[2,1]\right],$$
$$\left[\frac{\%2\,\%1\,e^{z1}}{1+e^{z1}}+\%3\,\%4=0,V,[-2,2]\right],[\%4\,\%1=1,V,[-2,1]],$$
$$\left[\frac{1}{2}\,\frac{\%2\,\%1\,e^{z1}}{1+e^{z1}}+\%3\,\%4=0,V,[-2,-2]\right],[\%4\,\%1=1,V,[-2,-1]],$$
$$\left[\frac{1}{2}\,\frac{\%2\,\%1\,e^{z1}}{1+e^{z1}}+\%3\,\%4=0,V,[2,-1]\right],\Big[\frac{1}{2}\,\frac{\%2\,\%1\,e^{z1}}{1+e^{z1}}+\%3\,\%4$$
$$+\frac{e^{z1}\,\%3\,V\_S1}{(1+e^{z1})^2}+\frac{1}{2}\,\frac{e^{z1}\,\%1\,\mathrm{D}(V\_S1)}{1+e^{z1}}-\frac{1}{2}\,\frac{(e^{z1}-1)\,e^{z1}\,\%1\,V\_S1}{(1+e^{z1})^2}=0,$$
$$V,[0,-2]\Big],\Big[\%3\,\%4+\frac{\%2\,\%1\,e^{z1}}{1+e^{z1}}+\frac{e^{z1}\,\%1\,\mathrm{D}(V\_S1)}{1+e^{z1}}+\frac{e^{z1}\,\%3\,V\_S1}{(1+e^{z1})^2}$$
$$-\frac{(e^{z1}-1)\,e^{z1}\,\%1\,V\_S1}{(1+e^{z1})^2}=0,V,[0,2]\Big],$$
$$\left[\%4\,\%1+\frac{e^{z2}\,V\_S2\,\%4}{(1+e^{z2})^2}=1,V,[-2,0]\right]\Big\},\Big\{$$
$$\left[\frac{(1+e^{z1})\,\%3\,\%4}{2\,e^{z1}+1}=0,V,[2,2]\right],[\%4\,\%1=0,V,[2,1]],$$
$$\left[\%4\,\%1+\frac{e^{z2}\,V\_S2\,\%4}{(1+e^{z2})^2}=0,V,[2,0]\right],$$
$$\left[\frac{\%3\,\%4}{1+e^{z2}}+\%1\,\%2=0,V,[-2,-1]\right],\Big[\frac{\%3\,\%4}{1+e^{z2}}+\%1\,\%2$$

$$-\frac{(e^{z2}-1)\,V\_S2\,\%4}{(1+e^{z2})^2}+\frac{e^{z2}\,V\_S2\,\%2}{(1+e^{z2})^2}+\frac{\%4\,\mathrm{D}(\,V\_S2\,)}{1+e^{z2}}=0,V,[-2,0]\Bigg],$$

$$\left[\frac{\%3\,\%4}{1+e^{z2}}+\%1\,\%2=0,V,[-2,2]\right],$$

$$\left[\frac{(1+e^{z1})\,\%3\,\%4}{2\,e^{z1}+1}=0,V,[-2,1]\right],[\%4\,\%1=0,V,[2,-1]],$$

$$\left[\frac{(1+e^{z1})\,\%3\,\%4}{2\,e^{z1}+1}+\frac{e^{z1}\,\%3\,V\_S1}{(2\,e^{z1}+1)(1+e^{z1})}=0,V,[0,2]\right]\Bigg\}\Bigg],\Bigg[$$

$$[1,1,HIGH,[2]]=[2,2,LOW,[-1]],\Bigg[$$

$$[\%4\,\%1,V,[1,2]]=[\%4\,\%1,V,[-1,-2]],$$

$$\left[\frac{1}{2}\sqrt{2}\,\%4\,\%3+\frac{\sqrt{2}\,(1+e^{z2})\,\%2\,\%1}{2+e^{z2}},V,[2,-2]\right]=$$

$$\left[\frac{\sqrt{2}\,(1+e^{z1})\,\%3\,\%4}{2\,e^{z1}+1}+\frac{1}{2}\sqrt{2}\,\%2\,\%1,V,[2,-2]\right],$$

$$\left[\%4\,\%1+\frac{e^{z2}\,V\_S2\,\%4}{(1+e^{z2})^2},V,[1,0]\right]=$$

$$\left[\%4\,\%1+\frac{e^{z1}\,\%1\,V\_S1}{(1+e^{z1})^2},V,[0,-1]\right],$$

$$[\%4\,\%1,V,[1,-2]]=[\%4\,\%1,V,[1,-2]],\Bigg[\frac{e^{z2}\sqrt{2}\,V\_S2\,\%2}{(2+e^{z2})(1+e^{z2})}$$

$$+\frac{\sqrt{2}\,(1+e^{z2})\,\%2\,\%1}{2+e^{z2}}+\frac{1}{2}\sqrt{2}\,\%4\,\%3-\frac{1}{2}\,\frac{\sqrt{2}\,(e^{z2}-1)\,V\_S2\,\%4}{1+e^{z2}}$$

$$+\frac{1}{2}\sqrt{2}\,\%4\,\mathrm{D}(\,V\_S2\,),V,[2,0]\Bigg]=\Bigg[\frac{\sqrt{2}\,(1+e^{z1})\,\%3\,\%4}{2\,e^{z1}+1}$$

$$+\frac{1}{2}\sqrt{2}\,\%2\,\%1-\frac{1}{2}\,\frac{\sqrt{2}\,(e^{z1}-1)\,\%1\,V\_S1}{1+e^{z1}}+\frac{e^{z1}\sqrt{2}\,\%3\,V\_S1}{(2\,e^{z1}+1)(1+e^{z1})}$$

$$+\frac{1}{2}\sqrt{2}\,\%1\,\mathrm{D}(\,V\_S1\,),V,[0,-2]\Bigg],$$

$$\left[\frac{1}{2}\sqrt{2}\,\%4\,\%3+\frac{\sqrt{2}\,(1+e^{z2})\,\%2\,\%1}{2+e^{z2}},V,[2,2]\right]=$$

$$\left[\frac{\sqrt{2}\,(1+e^{z1})\,\%3\,\%4}{2\,e^{z1}+1}+\frac{1}{2}\sqrt{2}\,\%2\,\%1,V,[-2,-2]\right]\Bigg]\Bigg],[z1,z2],\Bigg[\Bigg[$$

$$\left[y1\to\ln\left(\frac{y1}{1-y1}\right),z1\to\frac{e^{z1}}{1+e^{z1}},y1\to(y1,1-y1)\right],$$

$$\left[y2\to\ln\left(\frac{y1}{1-y1}\right),z2\to\frac{e^{z1}}{1+e^{z1}},y2\to(y1,1-y1)\right]\Bigg],\Bigg[$$

$$\left[y1\to\ln\left(\frac{y1}{1-y1}\right),z1\to\frac{e^{z1}}{1+e^{z1}},y1\to(y1,1-y1)\right],$$

$$\left[y2\to\ln\left(\frac{y1}{1-y1}\right),z2\to\frac{e^{z1}}{1+e^{z1}},y2\to(y1,1-y1)\right]\Bigg]\Bigg],$$

$$[[V,[[1,1],[1,1]]]]\Bigg]$$

$$\%1 := \mathrm{V\_B2}\left(\frac{3\,e^{z2}+1}{(1+e^{z2})^3}, \frac{e^{z2}}{(1+e^{z2})^3}, \frac{(e^{z2})^2\,(3+e^{z2})}{(1+e^{z2})^3}, -\frac{(e^{z2})^2}{(1+e^{z2})^3}\right)$$
$$\%2 := \mathrm{V\_B1}\left(-6\,\frac{e^{z1}}{(1+e^{z1})^2}, -\frac{2\,e^{z1}-1}{(1+e^{z1})^2}, 6\,\frac{e^{z1}}{(1+e^{z1})^2}, \frac{e^{z1}\,(e^{z1}-2)}{(1+e^{z1})^2}\right)$$
$$\%3 := \mathrm{V\_B2}\left(-6\,\frac{e^{z2}}{(1+e^{z2})^2}, -\frac{2\,e^{z2}-1}{(1+e^{z2})^2}, 6\,\frac{e^{z2}}{(1+e^{z2})^2}, \frac{e^{z2}\,(e^{z2}-2)}{(1+e^{z2})^2}\right)$$
$$\%4 := \mathrm{V\_B1}\left(\frac{3\,e^{z1}+1}{(1+e^{z1})^3}, \frac{e^{z1}}{(1+e^{z1})^3}, \frac{(e^{z1})^2\,(3+e^{z1})}{(1+e^{z1})^3}, -\frac{(e^{z1})^2}{(1+e^{z1})^3}\right)$$
$$\%5 := \mathrm{V\_B2}\left(6\,\frac{e^{z2}-1}{1+e^{z2}}, 2\,\frac{e^{z2}-2}{1+e^{z2}}, -6\,\frac{e^{z2}-1}{1+e^{z2}}, 2\,\frac{2\,e^{z2}-1}{1+e^{z2}}\right)$$
$$\%6 := 2+2\,e^{z2}+(e^{z2})^2$$
$$\%7 := \mathrm{V\_B1}\left(6\,\frac{e^{z1}-1}{1+e^{z1}}, 2\,\frac{e^{z1}-2}{1+e^{z1}}, -6\,\frac{e^{z1}-1}{1+e^{z1}}, 2\,\frac{2\,e^{z1}-1}{1+e^{z1}}\right)$$
$$\%8 := 1+2\,e^{z1}+2\,(e^{z1})^2$$

```
> time() - Start;
```

57.167

```
> type(SProb, MultiSProbType);
```

true

```
> eval(ExtraBases);
```

table([
1 = table([
$(V, 1) =$
$[(2\,y1+1)(y1-1)^2, y1\,(y1-1)^2, -y1^2\,(-3+2\,y1), y1^2\,(y1-1)]$
$(V, 2) =$
$[(2\,y2+1)(y2-1)^2, y2\,(y2-1)^2, -y2^2\,(-3+2\,y2), y2^2\,(y2-1)]$
])
2 = table([
$(V, 1) =$
$[(2\,y1+1)(y1-1)^2, y1\,(y1-1)^2, -y1^2\,(-3+2\,y1), y1^2\,(y1-1)]$
$(V, 2) =$
$[(2\,y2+1)(y2-1)^2, y2\,(y2-1)^2, -y2^2\,(-3+2\,y2), y2^2\,(y2-1)]$
])
])

End of Maple Worksheet

Method of Implementation

The procedure first checks for the presence of the `MapInfo` argument. If it is missing the procedure calls **LogRatioMap** repeatedly to construct its default value.

Next the procedure calls **multi_spec_to_list** (from the ***order_ops*** LLF) in order to organize the order information for each domain as a list of *OrderSpecType*s.

Then the procedure performs a process analogous to that performed by **CollocateRec** for each of the subdomains. The process is complicated by the fact that coupling equations have to be assigned collocation points along with the boundary constraints.

So for each subdomain the first step is to search the entire list of coupling equations and extract equation-halves that are applied in the current domain. For each equation-half applied in the current domain a *BoundTagType* is constructed, where the tag-information is the sequence number of the coupling equation from which the equation half was extracted. After this the list of *BoundTagType*s is combined with the list of *BoundCondType*s from the same domain to make a list of *BoundFormType*s.

This collection of boundary forms is used with the order information for the current subdomain in a call to the **required_order** to determine the minimum order of approximation required for this domain. Then **make_bases** is called to construct the main-bases and the extra-bases for this order. Then **collocate_main** is called to collocate the governing equation for this domain. Next **collocate_bound** is called to collocate all of the boundary forms. Results that were derived from the *BoundCondType* will be of type *OverRideType*; these results are separated from the results that are derived from *BoundTagType*. The former will be used directly to construct the override equations in the final result. The latter are stored in a table, indexed by the domain number, for further processing.

Once this process has been completed for every subdomain, the table of collocated halves of the coupling equations is used to match halves from different domains to form all of the *OverCoupleType*s. This reconstruction is performed by copying each half into a table indexed in such a way that any halves that have the same index match. When an equation half is copied into a table element already containing a equation half, the two halves are combined into an *OverCoupleType*.

Matching Halves of the OverCoupleTypes Each half of a coupling equation is applied in only one region in its subdomain. Determining which region in the other subdomain matches this region is particularly tricky.

To understand the problem it is useful to recognize that the *CupleCondType*s are defined over the entire boundary of a domain whereas *OverCoupleType* is defined for only one region on the boundary. This means that, except when the problem is one dimensional, each coupling equation will result in many *OverCoupleType*s. This is illustrated in Figure 4.5.

The procedure **collocate_bound** takes care of generating many *OverRideForm*s for each *BoundFormType*, so **CollocateMultiRec** does not have to worry about this. However, it is no longer sufficient to match halves that have merely have the same tags. This would match only results originate from the same coupling equation, not necessarily applied in matching regions. Results must originating from the same coupling equation and be applied in matching regions in order to be halves of the same *OverCoupleType*. The regions are matched by converting the region number for the actual subdomain (i.e., the region number returned by **collocate_bound**) to the matching region number in the implicitly defined boundary specified by the *CoupleOrientType*. If two regions map to the same region on this surface then they map to each other.

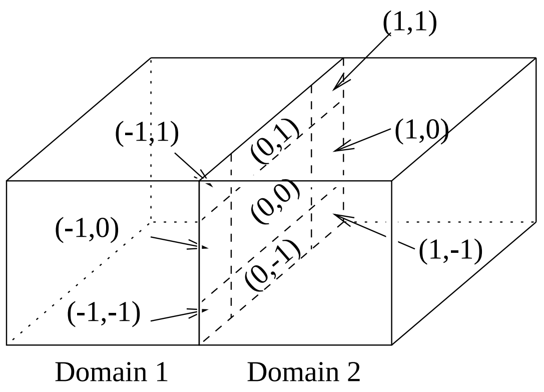

Figure 4.5: The Regions on the Boundary Between Two, Three-Dimensional Domains

Notice that there is one plane between the two domains but this plane is divided into nine regions (i.e., three per dimension). Each domain is divided into 27 regions but most these regions are not shown in order to keep the complexity of the illustration manageable.

Dependencies

If maps are not explicitly specified via the `MapInfo` argument then this procedure will call **LogRatioMap** to construct a map. It also calls the functions **multi_spec_to_list** and **spec_list_to_multi** (from the LLF ***order_ops***) to convert between the type *MultiOrderSpecType*'s and the equivalent list of *OrderSpecType*'s.

The procedure **required_order** is called once for each subdomain to determine the order of approximation required for that domain. Then **make_bases** is called (also once per subdomain) to construct the main-bases and the extra-bases. Finally, the procedures **collocate_main** and **collocate_bound** are called to perform the primary collocation operations.

Chapter 5

Numerical Solutions

This chapter describes procedures designed for obtaining numerical solutions to the system collocation equations. The functionality provided in this version of **PTOLEMY** for numerically solving the matrix system resulting from collocation is limited. Support is provided for expressing linear systems of collocation equations as a block matrix problem using Kronecker product notation. A portable application-independent text-based notation is defined for specifying the type of matrix problems that arise in sinc-collocation. The chapter concludes with a brief description of a collocation of C++ classes and a few C++ programs that can be used to read a problem description in this format, interpret[1] the functions found in the description, and build the double precession matrix described in the file.

These capabilities are limited compared to the large number of tools which a user might reasonably want. Obvious extensions include procedures for linearizing nonlinear problems, solvers and problem description formats for directly solving some nonlinear problems, a resolver that can avoid repeating unnecessary work when only a portion of the matrix description has changed, and a distributed parallel solver.

Provided that a linear solver exists these extra feature could be implemented as obvious extensions. However, such added features represent a significant investment in development time. For these reasons this version of **PTOLEMY** provides only numerical tools for linear problems.

5.1 Linearizahion

A PDE is considered to be linear if it is linear with respect to all of the state-variables and all of the state-variables derivatives and if all of the boundary constraints are linear in the same sense. The coefficients of the various state-variables (or their derivatives) in a linear PDE may be arbitrarily complicated functions of the domain coordinates. As a result a linear PDE will remain linear after it is mapped to a new domain. In addition, collocating a linear

[1]Here the word "interpret" is used in the formal computer science sense of the word and is contrasted with compilation based alternatives.

PDE (defined on a parallelepiped) yields a linear system of algebraic equations which can be rewritten as a matrix problem.

This implies that the collocation of any linear PDE results in a matrix problem. In this case defining "setup" as transforming the PDE into a matrix problem seems easily justified. It is still necessary to solve the matrix problem but it is reasonable to define that as outside the scope of "setup." Since a nonlinear PDEs cannot be reduced to a matrix problem it is less obvious that procedures provided by **PTOLEMY** actually complete the setup problem in these cases. In fact, I maintain that without the addition of a few more features **PTOLEMY** does not perform the "setup" for nonlinear PDEs.

Fortunately the setup of nonlinear problems follows directly from the linear case. There are two ways of solving nonlinear PDEs: 1) apply an iterative solver to the continuous problem and 2) apply an iterative solver to the nonlinear algebraic problem. Either approach can be added to **PTOLEMY**; the former is probably easier and yields more useful information about the solution process while the latter is probably faster. In both cases the iterative solver needs to be able to solve linear equations and evaluate (but not solve) nonlinear expressions. Another way of saying this is that all of the nonlinear solvers iterate and on each iteration they linearize the problem about the current guess of the solution. As a result they only need to "solve" linear problem forms.

5.2 Block Matrices

Block matrices are discussed extensively in contemporary tutorials of matrix computations; see [8] for a focused introduction. They have acquired a central importance in contemporary computing because of their locality and the dominating importance of memory performance in modern computational environments.

PTOLEMY uses a block matrix representation to report the matrix formulation of the problem, but this is primarily done for logical rather than performance reasons. The matrix is organized in such a way that each block contains internal consistency which allows it to be expressed symbolically without the use of case constructs.

Each unknown in the matrix problem is a parameter of the approximation. Because the approximation is a linear combination of the bases, each unknown corresponds directly to a particular bases. These unknowns are grouped in the same way that the bases were grouped during the collocation process; this grouping of the unknowns is in fact a partition of the unknowns and is used to directly define the partition of the columns of the matrix.

In order to facilitate bookkeeping about the system of equations each constraint is placed in the row number that matches the column number of the collocation point at which the constraint was applied. That is corresponding to every row is a column; corresponding to every column is a basis associated with the unknown corresponding to this column, corresponding to every basis is a collocation point, and the equation in every row is the constraint applied at this collocation point. An immediate consequence of this is that the row partition of the matrix is the same as the column partition. A less obvious consequence of this convention is that it facilitates the symbolic expression of the matrix.

One additional complication is that each grouping of unknowns is represented by a single symbol called the "stack variable name." The ordering of the stack-variable names does not totally define the ordering of the unknowns, but it is still necessary to define the order of the unknowns within each group. The convention employed here is that the unknowns within each "stack" are ordered using the natural cartesian ordering of the coordinates of the collocation points associated with each unknown. It is because the intergroup ordering of unknowns is cartesian and the same for both rows and columns that each block of the matrix can be expressed as the Kronecker product of special matrices.

5.3 Kronecker Products and *I*-Notation

The notation used to represent the block matrices is an extension of that which is common in the sinc literature. In addition to a few extensions the notation has undergone a slight translation in order to be expressed using Maple syntax. The commonly used family of special matrices, denoted $I^{(m)}$ (for $m \in \mathbb{Z}$) in the sinc literature, is represented in **PTOLEMY** by the names `I.m`. For example in the matrix $I^{(1)}$ is be represented as `I1`.

The following is a list of the definitions of the special matrix used by **PTOLEMY**:

$$\texttt{Diag(f(z))} := \begin{bmatrix} f(-N\,h) & 0 & & & 0 \\ 0 & f\big(-(N-1)\,h\big) & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & f\big((N-1)\,h\big) & 0 \\ 0 & & & 0 & f(N\,h) \end{bmatrix}$$

$$\texttt{I.m} := \begin{bmatrix} \delta_0^{(m)} & \delta_1^{(m)} & \cdots & \delta_N^{(m)} \\ \delta_{-1}^{(m)} & \delta_0^{(m)} & \cdots & \delta_{N-1}^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{-N}^{(m)} & \delta_{-(N-1)}^{(m)} & \cdots & \delta_0^{(m)} \end{bmatrix}$$

$$\texttt{I.m.\_rev} := \begin{bmatrix} \delta_{-N}^{(m)} & \cdots & \delta_{-1}^{(m)} & \delta_0^{(m)} \\ \delta_{-1}^{(m)} & \cdots & \delta_0^{(m)} & \delta_1^{(m)} \\ \vdots & \ddots & \vdots & \vdots \\ \delta_0^{(m)} & \cdots & \delta_{N-1}^{(m)} & \delta_N^{(m)} \end{bmatrix}$$

where $\delta_k^{(m)}$ is the m^{th}-derivative of the sinc function evaluated at k for $k \in \mathbb{Z}$.

$$\texttt{C\_rev} = \texttt{C} := [\;1 \quad 1 \quad \cdots \quad 1\;]$$

and

$$\mathtt{R} := \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

where z is the relevant coordinate name and h is the step size parameter.

Kronecker product notation is widely used in the sinc-literature, but for the sake of completeness the briefest possible summary of the notation is included here. Give two matrices A and B the Kronecker product of A and B, denoted $A \otimes B$, is the matrix where each element of A, $a_{i,j}$, is replaced by $a_{i,j}\, B$. For example

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \otimes \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} =$$
$$\begin{bmatrix} a_{1,1}\,b_{1,1} & a_{1,1}\,b_{1,2} & a_{1,2}\,b_{1,1} & a_{1,2}\,b_{1,2} & a_{1,3}\,b_{1,1} & a_{1,3}\,b_{1,2} \\ a_{1,1}\,b_{2,1} & a_{1,1}\,b_{2,2} & a_{1,2}\,b_{2,1} & a_{1,2}\,b_{2,2} & a_{1,3}\,b_{2,1} & a_{1,3}\,b_{2,2} \\ a_{2,1}\,b_{1,1} & a_{2,1}\,b_{1,2} & a_{2,2}\,b_{1,1} & a_{2,2}\,b_{1,2} & a_{2,3}\,b_{1,1} & a_{2,3}\,b_{1,2} \\ a_{2,1}\,b_{2,1} & a_{2,1}\,b_{2,2} & a_{2,2}\,b_{2,1} & a_{2,2}\,b_{2,2} & a_{2,3}\,b_{2,1} & a_{2,3}\,b_{2,2} \end{bmatrix}$$

Notice that the Kronecker product is not a communicative operation, but it is associative.

5.4 Parameter Components

The symbolic representation of the matrix block does not specify the size of each block. In addition the matrix definition depends on the sample rate. Before a fully numerical matrix may be instantiated both the block sizes and the sample rates must be specified.

The sample rates are called H-parameters and denoted, $h_1, \ldots, h_n$, where n is the dimension of the domain. The size of each special matrix is determined by the number of samples associated with sinc bases in each dimension of the corresponding subdomain; that is, the number of samples not counting samples corresponding to extra bases. In each dimension the sinc-bases components are numbered from $-N_i$ to N_i for $N_i \in \mathbb{Z}$. The collection of constants $N_1, \ldots, N_n$ are called the N-parameters.

Because the current version of **PTOLEMY** does not construct matrices that interpolate the sinc approximation onto another set of samples, the number of samples and the sampling rate must be the same in corresponding dimensions for the inner group of collocation points as for the group of collocation points along the boundaries of the subdomain. That is in each dimension the N and H parameters must be the same on each boundary and in the interior of the domain.

For a single domain problem this restriction does not result in a unnatural requirement. However, when two subdomains are coupled along a boundary then in order to apply the coupling constraint the parameters must be the same in each of the subdomains participating in the coupling. This can cause a sequence of equivalency constraints across several subdomains which in turn can yield nonobvious constraints on the choice of parameters. For example the five traditional subdomains in Figure 5.1 are coupled in such a way that in Domain 1, N_1 must equal to N_2 and that H_1 must equal to H_2.

Upon careful inspection this is not difficult to deduce, but the opportunity for specifying inconsistent parameter sets is significant. To minimize this problem **Ptolemy** groups all of the parameters that must be equal and requires the user to specify parameters once for each group of equivalent parameters.

Another consequence of the fact that **Ptolemy** does not build matrix blocks that interpolate an approximation at points other than the sinc points is that each subdomain all of the parameters for all of the state-variables must be the same. This combined with the fact that the same equivalency constraints apply to both the N-parameters and the H-parameters implies that all that is required is to determine which domain-dimension pairs must have the same parameters.

Ptolemy determines all of the groups of parameters that must be equivalent by constructing a graph in which the domain-parameter pairs are the *nodes* and the coupling constraints are the *edges*. It then computes all of the graph's components (see [9] for a description of graph components). The node of each component are a group of parameters that must be the same. These components are then ordered and reported to the user who must specify the N and H parameter for each component.

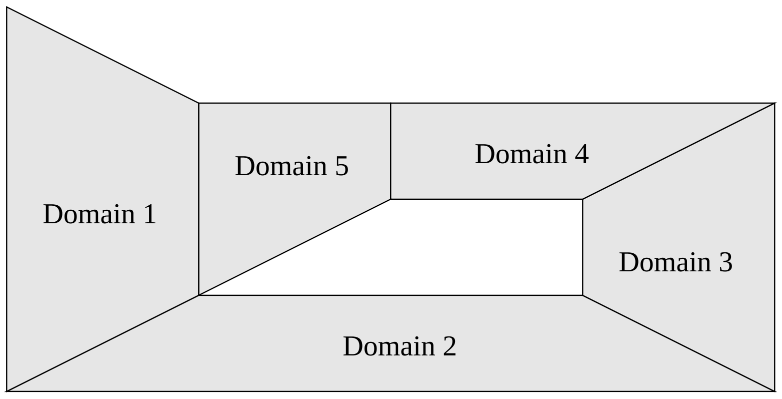

Figure 5.1: Five Coupled Subdomains that Exhibit Interesting Parameter Constraints

Linear

This function checks to see if a problem statement in SB-notation is a well-formed collocation of a linear PDE.

Linear(Prob: {*SProbType, MultiSProbType*})

In order to be a well-formed result of collocating a linear expression an expression must be the linear combination of "terms" such that each term is the collocation of a single state-variable. Collocating a single state-variable in an n-dimensional domain will result in the product of n SB-variables, one from each dimension of the domain.

So the collocation of a linear expression is linear with respect to each SB-variable, but the requirement that the expression be the result of collocating a linear differential expression is also much more restricted than this. For each term all of the SB-variables must be derived from the same state-variable and there must be a one-to-one correspondence between these SB-variables and the dimensions of the domain.

Example

Start of Maple Worksheet

```
> with(ptolemy, Linear, LogRatioMap);
```

$$[\mathit{Linear}, \mathit{LogRatioMap}]$$

```
> Map := LogRatioMap(0,1,y,z);
```

$$\mathit{Map} := \left[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\right]$$

```
> Spline1 := exp(z2) / (1 + exp(z2)); Spline2 := 1 / (1 + exp(z2));
```

$$\mathit{Spline1} := \frac{e^{z2}}{1+e^{z2}}$$

$$\mathit{Spline2} := \frac{1}{1+e^{z2}}$$

```
> Prob1 :=
>    [(D@@2)(V_S1)*V_S2 + D(V_S1)*V_S2 + V_S1*(D@@2)(V_S2) + D(V_S2)*V_S1 = 0,
>     [V_S1*V_B2(Spline1,Spline2) = z1^2, V,[-1,0]], [z1,z2],
>     [Map,Map], [[V,[0,1]]]];
```

$$\mathit{Prob1} := \Big[D^{(2)}(\mathit{V\_S1})\,\mathit{V\_S2} + \mathrm{D}(\mathit{V\_S1})\,\mathit{V\_S2} + \mathit{V\_S1}\,D^{(2)}(\mathit{V\_S2})$$
$$+ \mathrm{D}(\mathit{V\_S2})\,\mathit{V\_S1} = 0,$$
$$\left[\mathit{V\_S1}\,\mathrm{V\_B2}\left(\frac{e^{z2}}{1+e^{z2}}, \frac{1}{1+e^{z2}}\right) = \mathit{z1}^2, V, [-1, 0]\right], [\mathit{z1}, \mathit{z2}], \Big[$$
$$\left[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\right],$$
$$\left[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\right]\Big], [[V, [0, 1]]]\Big]$$

```
> Linear(Prob1);
```

$$\mathit{true}$$

```
> Prob2 := subsop(2 = [V_S1*V_B1=K, V,[-1,0]], Prob1);
```

$$\mathit{Prob2} := \Big[D^{(2)}(\mathit{V\_S1})\,\mathit{V\_S2} + \mathrm{D}(\mathit{V\_S1})\,\mathit{V\_S2} + \mathit{V\_S1}\,D^{(2)}(\mathit{V\_S2}) + \mathrm{D}(\mathit{V\_S2})\,\mathit{V\_S1} = 0, [\mathit{V\_S1}\,\mathit{V\_B1} = K, V, [-1,0]], [z1, z2], \Big[\Big[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\Big], \Big[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\Big]\Big], [[V, [0,1]]]\Big]$$

```
> Linear(Prob2);
```

$$\mathit{false}$$

```
> Prob3 := subsop(1 = (cos(V_S1*V_S2)=0), Prob1);
```

$$\mathit{Prob3} := \Big[\cos(\mathit{V\_S1}\,\mathit{V\_S2}) = 0, \Big[\mathit{V\_S1}\,\mathrm{V\_B2}\left(\frac{e^{z2}}{1+e^{z2}}, \frac{1}{1+e^{z2}}\right) = z1^2, V, [-1,0]\Big], [z1, z2], \Big[\Big[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\Big], \Big[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\Big]\Big], [[V, [0,1]]]\Big]$$

$$\mathit{Prob3} := \Big[\cos(\mathit{V\_S1}\,\mathit{V\_S2}) = 0, \Big[\mathit{V\_S1}\,\mathrm{V\_B2}\left(\frac{e^{z2}}{1+e^{z2}}, \frac{1}{1+e^{z2}}\right) = z1^2, V, [-1,0]\Big], [z1, z2], \Big[\Big[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\Big], \Big[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\Big]\Big], [[V, [0,1]]]\Big]$$

```
> Linear(Prob3);
```

$$\mathit{false}$$

```
> Prob4 := subsop(1 = (V_S1^2*V_S2*D(V_S2)=0), Prob1);
```

$$\mathit{Prob4} := \Big[\mathit{V\_S1}^2\,\mathit{V\_S2}\,\mathrm{D}(\mathit{V\_S2}) = 0, \Big[\mathit{V\_S1}\,\mathrm{V\_B2}\left(\frac{e^{z2}}{1+e^{z2}}, \frac{1}{1+e^{z2}}\right) = z1^2, V, [-1,0]\Big], [z1, z2], \Big[\Big[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\Big], \Big[y \to \ln\left(\frac{y}{1-y}\right), z \to \frac{e^z}{1+e^z}, y \to (y, 1-y)\Big]\Big], [[V, [0,1]]]\Big]$$

```
> Linear(Prob4);
```

false

End of Maple Worksheet

Method of Implementation

This procedure first extracts a list of all state-variables from the "OrderSpec field" in the problem statement. It then constructs a table for each possible `SB`-variable indicating which state-variable and which dimension correspond to this `SB`-variable. Then the procedure extracts the portions of the problem statement which must be the collocation of linear expressions. Finally, it uses the following helper functions to determine if these expressions are all of the correct form.

good_exp(`Exp`: *collectStruct({algebraic, equation})*, `VarSet`: *set(name)*, `Info`: *table*, `Dimen`: *posint*)
good_term(`Term`: *algebraic*, `VarSet`: *set(name)*, `Info`: *table*, `Dimen`: *posint*)
get_var(`Term`: *algebraic*, `VarSet`: *set(name)*)

The procedure **good_exp** checks to see if some structured collection of expressions or equations are all well-formed results of collocating a linear expression of a linear equation. The procedure **good_term** checks to see if the expression specified by the argument `Term` could have been collocated from a coefficient (i.e., an expression independent of all of the state-variables) times a single state-variable. The procedure **get_var** returns a sequence of all of the `SB`-variables which are factors of a potentially linear term. If an `SB`-variable appears as a factor of the term more than once it will appear in the result more than once. If the term could not be linear for some structural reason, **get_var** will return an error.

good_exp This procedure will call itself recursively on each element of a *list*, *set*, or *equation*. If the argument `Exp` is of type *'+'* then it will call **good_term** for each operand in the summation. If `Exp` is of type *'*'*, *function*, *'^'*, *name*, or *numeric* then the procedure will call **good_term**, directly passing it `Exp`. Finally, if `Exp` is some other type the procedure returns `false` without any further processing.

good_term This procedure calls **get_var** to extract the `SB`-variables that are factors in the expression specified by `Term`. If the sequence is empty then the term is valid, because it is the collocation of an expression that is constant with respect to the state-variables. If **get_var** generates an error or if the same `SB`-variable appears more than once in its result then the term is not valid.

If none of these conditions hold then the procedure will need to go through the additional steps of ensuring that all of the `SB`-variables are derived from the same state-variable and that there is a one-to-one correspondence between the `SB`-variables and the dimensions of the problem. In order to perform both of these checks the procedure extracts the state-variable and the dimension corresponding via the `SB`-variable from the table passed to the `Info` argument. This is much faster than extracting this information from the variable names.

get_var This procedure recursively traverses the expression tree constructing the sequence of `SB`-variables as it goes. If at any point it encounters an inherently nonlinear construct it generates an error.

- If the argument `Term` is either of type '*' or '+' then the procedure simply recurses.
- If `Term` is a `D`-operator then the procedure checks to see if the variable to which the `D`-operator is applied is one of the `SB`-variables specified in `VarSet`.
- If `Term` is a function, but was not a `D`-operator, then the procedure checks to see if any of the argument of the function are one of the `SB`-variables specified by **VarSet**. If any of the arguments contained `SB`-variables then `Term` is not linear and the procedure generates an error. Otherwise if the function name is one of the `SB`-variables then it is returned as the result of the procedure.
- If `Term` is a name and the name is one of the `SB`-variables specified by `VarSet` then the procedure returns `Term`.
- If `Term` is of type '^' then **get_var** is applied recursively to each of the operands of the expression. If neither the base of the expression nor the power of the expression contains any `SB`-variables then the procedure returns `NULL`. Otherwise if the power of the expression is of type `nonnegint` then the `SB`-variables contained in the base are repeated the appropriate number of times otherwise the procedure returns an error.
- If `Term` is of type *numeric* then the procedure returns `NULL`
- In all other cases the procedure generates a diagnostic error. Expression involving Maple constructs that do not fall into one of the above cases were not generated by collocation of a linear differential expression.

Dependencies

The procedure **get_var** calls the procedure **isDop** to determine if a function is actually a `D`-operator and if it is, to determine the variable to which the `D`-operator is applied.

SToLinKron

This procedure converts a *linear* problem from `SB`-notation to a matrix problem expressed in Kronecker product notation.

SToLinKron(`Prob`: *SProbType*)

This procedure is largely a textual translator that expands the rather compact `SB`-notation into the more easily evaluated Kronecker product notation. No mapping and very little algebra occurs in this procedure. Nevertheless the procedure specified by `ptolemy/SimpProc` may be explicitly invoked on some of the diagonal "factors" in the output (see the section titled "kron_ops" on page 289 for more details). The result returned by this procedure will be of type ***LinKronType***.

Example Usage

This example starts by directly entering the collocation of a fairly trivial PDE. In its original formulation the PDE would probably have been $\partial^2 V/\partial x_1^2 + \partial^2 V/\partial x_2^2 = K$ (i.e., Poisson's equation) applied over the unit cube $[0, 1] \times [0, 1]$ with the boundary conditions:

$V(x_2) = sin(\pi\, x_2)$	Along the left edge.
$V(x_2) = cos(\pi\, x_2)$	Along the right edge.
$V(x_1) = x_1$	Along the bottom edge.
$V(x_1) = -x_1$	Along the top edge.

These somewhat artificial boundary conditions are shown in Figure 5.2.

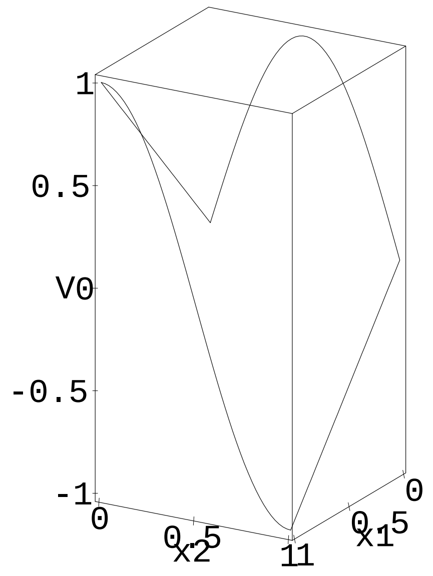

Figure 5.2: Boundary Values for the Example Problem

Start of Maple Worksheet

```
> with(ptolemy,LogRatioMap);
```

$$[\,LogRatioMap\,]$$

```
> Sp11 := 1/(1+exp(z1)); Sp12 := exp(z1)/(1+exp(z1));
```

$$Sp11 := \frac{1}{1+e^{z1}}$$

$$Sp12 := \frac{e^{z1}}{1+e^{z1}}$$

```
> Eq :=
>   V_B1(Sp11, Sp12) *(D@@2)(V_S2)*(1+exp(z2))^4/exp(z2)^2  +
>   V_B1(Sp11, Sp12) * D(V_S2)*(exp(z2)-1)*(1+exp(z2))^3/exp(z2)^2 +
>   V_S1 * (D@@2)(V_S2)*(1+exp(z2))^4/exp(z2)^2 +
>   V_S1 * D(V_S2)*(exp(z2)-1)*(1+exp(z2))^3/exp(z2)^2 = K;
```

$$Eq := \frac{\mathrm{V\_B1}\left(\frac{1}{1+e^{z1}}, \frac{e^{z1}}{1+e^{z1}}\right) D^{(2)}(\,V\_S2\,)\,(\,1+e^{z2}\,)^4}{(\,e^{z2}\,)^2} + \frac{\mathrm{V\_B1}\left(\frac{1}{1+e^{z1}}, \frac{e^{z1}}{1+e^{z1}}\right) \mathrm{D}(\,V\_S2\,)\,(\,e^{z2}-1\,)\,(\,1+e^{z2}\,)^3}{(\,e^{z2}\,)^2} + \frac{V\_S1\, D^{(2)}(\,V\_S2\,)\,(\,1+e^{z2}\,)^4}{(\,e^{z2}\,)^2} + \frac{V\_S1\, \mathrm{D}(\,V\_S2\,)\,(\,e^{z2}-1\,)\,(\,1+e^{z2}\,)^3}{(\,e^{z2}\,)^2} = K$$

```
> f1 := (exp(z1)/(1+exp(z1))^2 + sin(Pi*exp(z2) / (1+exp(z2))))/2;
```

$$f1 := \frac{1}{2}\frac{e^{z1}}{(\,1+e^{z1}\,)^2} + \frac{1}{2}\sin\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right)$$

```
> f2 := (exp(z1)/(1+exp(z1))^2 + cos(Pi*exp(z2) / (1+exp(z2))))/2;
```

$$f2 := \frac{1}{2}\frac{e^{z1}}{(\,1+e^{z1}\,)^2} + \frac{1}{2}\cos\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right)$$

```
> Bound := [
>   [V_B1(1,0)*V_B2(1,0) = f1, V,[-1,-1]],
>   [V_B1(1,0)*V_S2 = sin(Pi*exp(z2) / (1+exp(z2))), V,[-1,0]],
>   [V_B1(1,0)*V_B2(0,1) = f1, V,[-1,1]],
>   [V_S1*V_B2(1,0) =  exp(z1)/(1 + exp(z1)), V,[0,-1]],
>   [V_S1*V_B2(0,1) = -exp(z1) / (1 + exp(z1)) , V,[0,1]],
>   [V_B1(0,1)*V_B2(1,0) = f2, V,[1,-1]],
>   [V_B1(0,1)*V_S2 = cos(Pi*exp(z2) / (1 + exp(z2))), V,[1,0]],
>   [V_B1(0,1)*V_B2(0,1) = f2, V,[1,1]] ];
```

$$Bound := \Bigg[\Bigg[$$

$$\mathrm{V\_B1}(\,1,0\,)\,\mathrm{V\_B2}(\,1,0\,) = \frac{1}{2}\frac{e^{z1}}{(\,1+e^{z1}\,)^2} + \frac{1}{2}\sin\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), V, [\,-1,-1\,]\Bigg]$$

$$, \left[\mathrm{V\_B1}(\,1,0\,)\,V\_S2 = \sin\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), V, [\,-1,0\,]\right],$$

$$\left[\mathrm{V\_B1}(\,1,0\,)\,\mathrm{V\_B2}(\,0,1\,) = \frac{1}{2}\frac{e^{z1}}{(\,1+e^{z1}\,)^2} + \frac{1}{2}\sin\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), V, [\,-1,1\,]\right]$$

$$, \left[V\_S1 \, \mathrm{V\_B2}(1,0) = \frac{e^{z1}}{1+e^{z1}}, V, [0,-1] \right],$$
$$\left[V\_S1 \, \mathrm{V\_B2}(0,1) = -\frac{e^{z1}}{1+e^{z1}}, V, [0,1] \right],$$
$$\left[\mathrm{V\_B1}(0,1)\,\mathrm{V\_B2}(1,0) = \frac{1}{2}\,\frac{e^{z1}}{(1+e^{z1})^2} + \frac{1}{2}\cos\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), V, [1,-1] \right]$$
$$, \left[\mathrm{V\_B1}(0,1)\, V\_S2 = \cos\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), V, [1,0] \right],$$
$$\left[\mathrm{V\_B1}(0,1)\,\mathrm{V\_B2}(0,1) = \frac{1}{2}\,\frac{e^{z1}}{(1+e^{z1})^2} + \frac{1}{2}\cos\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), V, [1,1] \right]$$
$$\Big]$$

```
> Map := LogRatioMap(0,1,x,z);
```

$$Map := \left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (x, 1-x) \right]$$

```
> SProb := [Eq, Bound, [z1,z2], [Map,Map], [V, [0,0]]]:
> type(SProb, SProbType);
```

$$true$$

```
> with(ptolemy,SToLinKron);
```

$$[\,SToLinKron\,]$$

```
> Start := time():
> KronProb := SToLinKron(SProb);
```

$$KronProb := \left[\left[\left[\mathrm{Diag}\left(\frac{1}{z2^2}\right)(I0 \,\&\mathrm{K}\, I2) + \mathrm{Diag}\left(\frac{e^{z2}-1}{(1+e^{z2})\, z2}\right)(I0 \,\&\mathrm{K}\, I1), \right.\right.\right.$$
$$\mathrm{Diag}\left(\frac{1}{z2^2\,(1+e^{z1})}\right)(C \,\&\mathrm{K}\, I2)$$
$$+ \mathrm{Diag}\left(\frac{e^{z2}-1}{(1+e^{z2})\, z2\,(1+e^{z1})}\right)(C \,\&\mathrm{K}\, I1),$$
$$\mathrm{Diag}\left(\frac{e^{z1}}{z2^2\,(1+e^{z1})}\right)(C \,\&\mathrm{K}\, I2)$$
$$\left. + \mathrm{Diag}\left(\frac{(e^{z2}-1)\, e^{z1}}{(1+e^{z2})\, z2\,(1+e^{z1})}\right)(C \,\&\mathrm{K}\, I1), 0, 0, 0, 0, 0, 0 \right],$$
$$[0, \mathrm{Diag}(1)\,(C \,\&\mathrm{K}\, R0), 0, 0, 0, 0, 0, 0, 0],$$
$$[0, 0, \mathrm{Diag}(1)\,(C \,\&\mathrm{K}\, R0), 0, 0, 0, 0, 0, 0],$$
$$[0, 0, 0, \mathrm{Diag}(1)\,(R0 \,\&\mathrm{K}\, C), 0, 0, 0, 0, 0],$$
$$[0, 0, 0, 0, \mathrm{Diag}(1)\,(R0 \,\&\mathrm{K}\, C), 0, 0, 0, 0],$$
$$[0, 0, 0, 0, 0, \mathrm{Diag}(1)\,(C \,\&\mathrm{K}\, C), 0, 0, 0],$$
$$[0, 0, 0, 0, 0, 0, \mathrm{Diag}(1)\,(C \,\&\mathrm{K}\, C), 0, 0],$$
$$[0, 0, 0, 0, 0, 0, 0, \mathrm{Diag}(1)\,(C \,\&\mathrm{K}\, C), 0],$$
$$\left.[0, 0, 0, 0, 0, 0, 0, 0, \mathrm{Diag}(1)\,(C \,\&\mathrm{K}\, C)]\right], [V, V\_B1\_1, V\_B1\_2,$$
$$V\_B2\_1, V\_B2\_2, V\_B12\_1, V\_B12\_2, V\_B12\_3, V\_B12\_4], \Big[K,$$

$$\sin\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), \cos\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), \frac{e^{z1}}{1+e^{z1}}, -\frac{e^{z1}}{1+e^{z1}},$$
$$\frac{1}{2}\frac{e^{z1}}{(1+e^{z1})^2}+\frac{1}{2}\sin\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), \frac{1}{2}\frac{e^{z1}}{(1+e^{z1})^2}+\frac{1}{2}\sin\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right),$$
$$\frac{1}{2}\frac{e^{z1}}{(1+e^{z1})^2}+\frac{1}{2}\cos\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right), \frac{1}{2}\frac{e^{z1}}{(1+e^{z1})^2}+\frac{1}{2}\cos\left(\frac{\pi\, e^{z2}}{1+e^{z2}}\right)\Bigg],$$
$$[z1, z2], [H1, H2], [[1, 1], [1, 2]]\Bigg]$$

```
> time() - Start;
                              .834

> type(KronProb, LinKronType);
                              true
```

End of Maple Worksheet

In this case the complexity of the Kronecker product notation representation of the matrix problem is modest. However, as the problem complexity increases, the complexity of the Kronecker product representation grows more quickly than the complexity of the `SB`-notation. In fact, the results of this procedure will typically be the first to exceed the reasonable limits of human understanding.

In spite of the extent of the output the procedure executes quickly. This is because the operation involves so little math. In fact, even for very complicated problems the execution time of this procedure remains negligible.

Method of Implementation

This procedure first calls **get_SB_var** and **pde_order** to figure out the order of the governing equations. The order of the bases previously used in the collocation process is then subtracted from the order of the governing equation to produce the order of the weight to be applied. From this the actual weights are computed by calling **mapped_weight** (in the ***map_info_ops*** LLF).

Then the procedure uses the order of bases used in collocation process to count the number of extra bases components in each dimension. From this information the procedure constructs the stack variable names.

These names are of the form `V_B.i._.j` where 'i' and 'j' are integers. The value of 'i' indicates the dimension whose bases components are boundary splines. When more than one basis component is a boundary-spline, 'i' will be the concatenation (in ascending order) of each of the dimensions whose bases components is a boundary spline. The value of 'j' indicates the bases group number within the set of bases groups that have the same 'i' value.

For example in a two-dimensional domain the stack variables will be of the form

```
V, V_B1_*, V_B2_*, V_B12_*
```

where '*' represents a range of integers which is determined by knowing the number of extra bases components for each dimension. To make the example more concrete, assume that there are two extra bases components in dimension 1 and that there are four extra bases components in dimension 2. Then the state-variable names will be

```
V, V_B1_1, V_B1_2, V_B2_1, V_B2_2, V_B2_3, V_B2_4,
  V_B12_1, V_B12_2, V_B12_3, V_B12_4
  V_B12_5, V_B12_6, V_B12_7, V_B12_8
```

These stack variable names are constructed by calling **comb_all** (in the ***comb_ops*** LLF) to form a list of all the values of 'i'. Then for each state-variable name the number of bases groups with a given 'i' value is computed. These results are then combined to form the list of stack variables.

Next the procedure converts each governing equation to its Kronecker product format by calling **kron_eq** (from the ***kron_ops*** LLF). Each governing equation is associated with one of the state-variables. This allows a reasonable selection of Kronecker product results for each row of the matrix. Currently the procedure assigns governing equations to state-variables in the order in which the governing equations appears in the problem specification.

Converting override equations into rows of the final matrix is also performed by calling the **kron_eq** procedure. However, this is a bit harder because it is necessary to figure out which row will be replaced by the override equation. The symbolic equation for a block-row resulting from an override equation is kept in a table indexed by the associated stack variable name. So determining which row to assign the result of **kron_eq** is equivalent to constructing the stack variable name associated with the override equation. Both the 'i' and the 'j' parts of the the stack variable name are extracted from the collocation point number by calling **b_point_to_num** (from the ***b_ops*** LLF). An important assumption utilized by this process is that each override equation is collocated at a specific collocation point, which is associated with a particular basis, which is associated with a particular unknown in the matrix problem, which is associated with the row of the same number.

All of this is a matter of convention for **PTOLEMY**. As mentioned in Section 5.2 it would be possible to relax some of these conventions, but this would necessitate multiple procedures analogous to the one performed by **assign_bound**. Perhaps one procedure would be required to assign equations to collocation points, another to assign equations to rows of the matrix, and a third to assign an order to the unknowns. FEM often uses a grid point ordering scheme analogous to this in order to minimize the bandwidth of the resulting matrix problem. However, since the blocks in the matrices produced by **PTOLEMY** are either nearly dense or zero, little could be gained by using intrablock reordering of either columns or rows.

Next the procedure computes the parameter components. This is fairly trivial for a single domain system without any coupling.

To construct the final result the procedure checks each row to see if an override equation exists for that particular row; if not, the equation for the corresponding governing equation is used.

Dependencies

This procedure calls **get_SB_var** to get a list of the S-variables occurring in the governing equation. This is needed to call **pde_order** to determine the order of the governing equations. The procedure **order_subtract** is then used to determine the correct orders for the weights, which are constructed by calling **mapped_weight** (from the ***map_info_ops*** LLF).

The procedure **all_comb** (in the ***comb_ops*** LLF) is used to construct the stack variable names. The core operation of converting from `SB`-notation to Kronecker product notation is performed by the procedure **kron_eq**. Finally, the procedure **b_point_to_num** (in the ***b_ops*** LLF) is called to figure out the stack variable name associated with a specific override equation.

MultiSToLinKron

This procedure converts a *linear* problem defined over multiple subdomains from `SB`-notation to Kronecker product notation.

MultiSToLinKron(`Prob`: *MultiSProbType*)

The return type of this procedure is ***MultiSProbType***. Although no mapping occurs during the execution of **MultiSToLinKron**, the procedure specified by the global variable `ptolemy/SimpProc` may be explicitly invoked on some of the diagonal "factors" in the output (see the section titled "kron_ops" on page 289for more details).

Example Usage

Applied this procedure to the "L"-problem example considered in previous sections results in overwhelming long output. As a result, a highly simplified example is used to illustrate this procedure. The collocation system used here is the one that would result from applying Laplace's Equation over two coupled unit square domains. The state-variable name is V. In addition the left edge of domain 1 is coupled to the bottom edge of domain 2; however, for simplicity only the boundary values and none of the directional derivatives are constrained to be equal across the two domains. If, WOLG, it is assumed that the two domains were $[0,1] \times [0,1]$ and $[1,2] \times [1,2]$, prior to the mapping associated with collocation then Figure 5.3 illustrates the geometry of the problem. If in addition the coordinate names of the original domain where y_1 and y_2

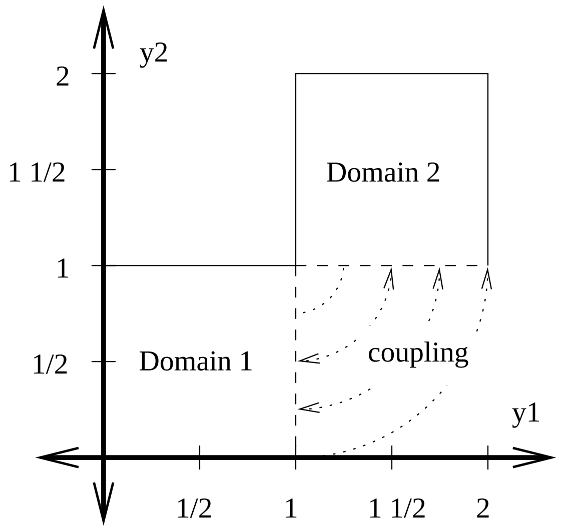

Figure 5.3: The Two Domains Used by the Example Problem

then the boundary values used in this example are

$V(y_2) = 0$	Along the left edge of domain 1
$V(y_1) = y_1^2$	Along the bottom edge of domain 1
$V(y_1) = \sqrt{y_1}$	Along the top edge of domain 1
$V(y_1) = 0$	Along the top edge of domain 2
$V(y_2) = y_2^2$	Along the right edge of domain 2
$V(y_2) = \sqrt{2 - y_2}$	Along the left edge of domain 2

These boundary values are illustrated in Figure 5.4.

Start of Maple Worksheet

```
> with(ptolemy, MultiSToLinKron, LogRatioMap);
```

$$[\mathit{LogRatioMap}, \mathit{MultiSToLinKron}]$$

```
> Sp1 := z -> 1/(1+exp(z)); Sp2 := z -> exp(z)/(1+exp(z));
```

$$\mathit{Sp1} := z \to \frac{1}{1 + e^z}$$

$$\mathit{Sp2} := z \to \frac{e^z}{1 + e^z}$$

```
> Eq :=
>   (D@@2)(V_S1) * (1+exp(z1))^4/exp(z1)^2 * (V_S2 + V_B2(Sp1(z2),Sp2(z2))) +
>   D(V_S1) * (1+exp(z1))^3*(exp(z1)-1)/exp(z1)^2 *
>     (V_S2 + V_B2(Sp1(z2),Sp2(z2))) +
>   (D@@2)(V_S2) * (1+exp(z2))^4/exp(z2)^2 * (V_S1 + V_B1(Sp1(z1),Sp2(z1))) +
>   D(V_S2) * (1+exp(z2))^3*(exp(z2)-1)/exp(z2)^2 *
>     (V_S1 + V_B1(Sp1(z1),Sp2(z1))) = 0;
```

Maple Worksheet Continued on Next Page

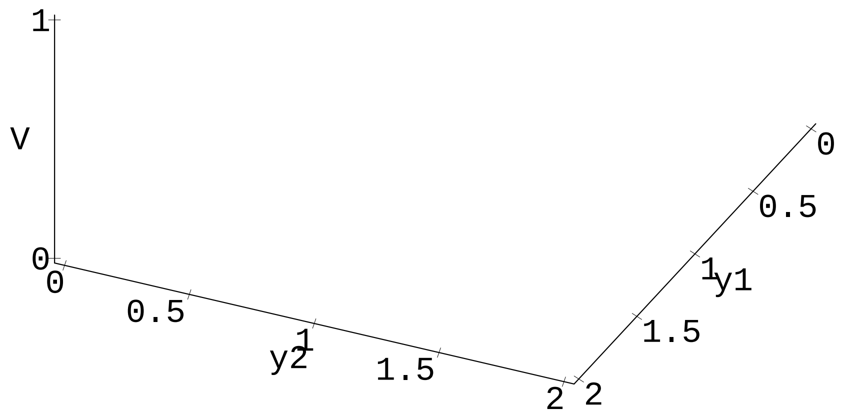

Figure 5.4: The Boundary Values Used in the Example Problem

$$Eq := \frac{D^{(2)}(\,V\_S1\,)\,(\,1+e^{z1}\,)^4\,\left(V\_S2+\mathrm{V\_B2}\left(\frac{1}{1+e^{z2}},\frac{e^{z2}}{1+e^{z2}}\right)\right)}{(\,e^{z1}\,)^2}+\mathrm{D}(\,V\_S1\,)$$
$$(\,1+e^{z1}\,)^3\,(\,e^{z1}-1\,)\,\left(V\_S2+\mathrm{V\_B2}\left(\frac{1}{1+e^{z2}},\frac{e^{z2}}{1+e^{z2}}\right)\right)\Big/(\,e^{z1}\,)^2$$
$$+\frac{D^{(2)}(\,V\_S2\,)\,(\,1+e^{z2}\,)^4\,\left(V\_S1+\mathrm{V\_B1}\left(\frac{1}{1+e^{z1}},\frac{e^{z1}}{1+e^{z1}}\right)\right)}{(\,e^{z2}\,)^2}+$$
$$\mathrm{D}(\,V\_S2\,)\,(\,1+e^{z2}\,)^3\,(\,e^{z2}-1\,)\,\left(V\_S1+\mathrm{V\_B1}\left(\frac{1}{1+e^{z1}},\frac{e^{z1}}{1+e^{z1}}\right)\right)\Big/$$
$$(\,e^{z2}\,)^2=0$$

```
> readlib(`ptolemy/pde_collect`):
> ExpEq := `ptolemy/pde_collect`(Eq, {V_S1, V_S2, V_B1, V_B2});
```

$$ExpEq := \frac{(\,1+e^{z1}\,)^4\,D^{(2)}(\,V\_S1\,)\,V\_S2}{(\,e^{z1}\,)^2}+\frac{(\,1+e^{z1}\,)^3\,(\,e^{z1}-1\,)\,\mathrm{D}(\,V\_S1\,)\,V\_S2}{(\,e^{z1}\,)^2}$$
$$+\frac{(\,1+e^{z2}\,)^4\,D^{(2)}(\,V\_S2\,)\,V\_S1}{(\,e^{z2}\,)^2}+\frac{(\,1+e^{z2}\,)^3\,(\,e^{z2}-1\,)\,\mathrm{D}(\,V\_S2\,)\,V\_S1}{(\,e^{z2}\,)^2}$$
$$+\frac{D^{(2)}(\,V\_S1\,)\,(\,1+e^{z1}\,)^4\,\mathrm{V\_B2}\left(\frac{1}{1+e^{z2}},\frac{e^{z2}}{1+e^{z2}}\right)}{(\,e^{z1}\,)^2}$$
$$+\frac{\mathrm{D}(\,V\_S2\,)\,(\,1+e^{z2}\,)^3\,(\,e^{z2}-1\,)\,\mathrm{V\_B1}\left(\frac{1}{1+e^{z1}},\frac{e^{z1}}{1+e^{z1}}\right)}{(\,e^{z2}\,)^2}$$
$$+\frac{\mathrm{D}(\,V\_S1\,)\,(\,1+e^{z1}\,)^3\,(\,e^{z1}-1\,)\,\mathrm{V\_B2}\left(\frac{1}{1+e^{z2}},\frac{e^{z2}}{1+e^{z2}}\right)}{(\,e^{z1}\,)^2}$$
$$+\frac{D^{(2)}(\,V\_S2\,)\,(\,1+e^{z2}\,)^4\,\mathrm{V\_B1}\left(\frac{1}{1+e^{z1}},\frac{e^{z1}}{1+e^{z1}}\right)}{(\,e^{z2}\,)^2}=0$$

```
> BC1 := [
>   [V_B1(1,0)*V_B2(1,0) = 0, V,[-1,-1]],
>     [V_B1(1,0)*V_S2 = 0, V,[-1,0]],
>     [V_B1(1,0)*V_B2(0,1) = 0, V,[-1,1]],
>   [V_S1*V_B2(1,0) =  Sp2(z1)^2, V,[0,-1]],
>   [V_S1*V_B2(0,1) = Sp2(z1)^(1/2), V,[0,1]]];
```

$$BC1 := \Big[[\,\mathrm{V\_B1}(\,1,0\,)\,\mathrm{V\_B2}(\,1,0\,)=0,V,[\,-1,-1\,]\,],$$
$$[\,\mathrm{V\_B1}(\,1,0\,)\,V\_S2=0,V,[\,-1,0\,]\,],$$
$$[\,\mathrm{V\_B1}(\,1,0\,)\,\mathrm{V\_B2}(\,0,1\,)=0,V,[\,-1,1\,]\,],$$
$$\left[V\_S1\,\mathrm{V\_B2}(\,1,0\,)=\frac{(\,e^{z1}\,)^2}{(\,1+e^{z1}\,)^2},V,[\,0,-1\,]\right],$$
$$\left[V\_S1\,\mathrm{V\_B2}(\,0,1\,)=\sqrt{\frac{e^{z1}}{1+e^{z1}}},V,[\,0,1\,]\right]\Big]$$

```
> BC2 := [
>   [V_B1(1,0)*V_B2(1,0) = 0, V,[-1,1]],
>     [V_B1(1,0)*V_S2 = 0, V,[0,1]],
>     [V_B1(1,0)*V_B2(0,1) = 0, V,[1,1]],
>   [V_S1*V_B2(1,0) =  Sp1(z1)^2, V,[-1,0]],
>   [V_S1*V_B2(0,1) = Sp1(z1)^(1/2), V,[1,0]]];
```

$$\begin{aligned} BC2 := \Big[&[\mathrm{V\_B1}(1,0)\,\mathrm{V\_B2}(1,0) = 0, V, [-1,1]],\\ &[\mathrm{V\_B1}(1,0)\ \mathit{V\_S2} = 0, V, [0,1]],\\ &[\mathrm{V\_B1}(1,0)\,\mathrm{V\_B2}(0,1) = 0, V, [1,1]],\\ &\left[\mathit{V\_S1}\ \mathrm{V\_B2}(1,0) = \frac{1}{(1+\mathrm{e}^{z1})^2}, V, [-1,0]\right],\\ &\left[\mathit{V\_S1}\ \mathrm{V\_B2}(0,1) = \sqrt{\frac{1}{1+\mathrm{e}^{z1}}}, V, [1,0]\right]\Big] \end{aligned}$$

```
> Couple := [[1,1,HIGH,[2]] = [2,2,LOW,[-1]],
>   [[V_B1(0,1)*V_B2(1,0), V,[1,-1]] = [V_B1(0,1)*V_B2(0,1), V,[1,-1]],
>    [V_B1(0,1)*V_S2, V,[1,0]] = [V_S1*V_B2(0,1), V,[0,-1]],
>    [V_B1(0,1)*V_B2(0,1), V,[1,1]] = [V_B1(1,0)*V_B2(0,1), V,[-1,-1]]]];
```

$$\begin{aligned} \mathit{Couple} := [[1,1,\mathit{HIGH},[2]] = [2,2,\mathit{LOW},[-1]],[&\\ &[\mathrm{V\_B1}(0,1)\,\mathrm{V\_B2}(1,0), V, [1,-1]] =\\ &[\mathrm{V\_B1}(0,1)\,\mathrm{V\_B2}(0,1), V, [1,-1]],\\ &[\mathrm{V\_B1}(0,1)\ \mathit{V\_S2}, V, [1,0]] = [\mathit{V\_S1}\ \mathrm{V\_B2}(0,1), V, [0,-1]],\\ &[\mathrm{V\_B1}(0,1)\,\mathrm{V\_B2}(0,1), V, [1,1]] =\\ &[\mathrm{V\_B1}(1,0)\,\mathrm{V\_B2}(0,1), V, [-1,-1]]]] \end{aligned}$$

```
> Map := [[LogRatioMap(0,1,x,z)$2], [LogRatioMap(1,2,x,z)$2]];
```

$$\begin{aligned} \mathit{Map} := \Bigg[\Bigg[&\left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{\mathrm{e}^z}{1+\mathrm{e}^z}, x \to (x, 1-x)\right],\\ &\left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{\mathrm{e}^z}{1+\mathrm{e}^z}, x \to (x, 1-x)\right]\Bigg], \Bigg[\\ &\left[x \to \ln\left(\frac{x-1}{2-x}\right), z \to \frac{2\,\mathrm{e}^z+1}{1+\mathrm{e}^z}, x \to (x-1, 2-x)\right],\\ &\left[x \to \ln\left(\frac{x-1}{2-x}\right), z \to \frac{2\,\mathrm{e}^z+1}{1+\mathrm{e}^z}, x \to (x-1, 2-x)\right]\Bigg]\Bigg] \end{aligned}$$

```
> Prob := [[ExpEq$2], [BC1,BC2], Couple, [z1,z2], Map, [V, [[0,0], [0,0]]]];
```

$$\begin{aligned} \mathit{Prob} := \Bigg[\Bigg[&\frac{(1+\mathrm{e}^{z1})^4\, D^{(2)}(\mathit{V\_S1})\ \mathit{V\_S2}}{(\mathrm{e}^{z1})^2}\\ &+ \frac{(1+\mathrm{e}^{z1})^3\,(\mathrm{e}^{z1}-1)\,\mathrm{D}(\mathit{V\_S1})\ \mathit{V\_S2}}{(\mathrm{e}^{z1})^2} + \frac{(1+\mathrm{e}^{z2})^4\, D^{(2)}(\mathit{V\_S2})\ \mathit{V\_S1}}{(\mathrm{e}^{z2})^2}\\ &+ \frac{(1+\mathrm{e}^{z2})^3\,(\mathrm{e}^{z2}-1)\,\mathrm{D}(\mathit{V\_S2})\ \mathit{V\_S1}}{(\mathrm{e}^{z2})^2} + \frac{D^{(2)}(\mathit{V\_S1})\,(1+\mathrm{e}^{z1})^4\,\%2}{(\mathrm{e}^{z1})^2}\\ &+ \frac{\mathrm{D}(\mathit{V\_S2})\,(1+\mathrm{e}^{z2})^3\,(\mathrm{e}^{z2}-1)\,\%1}{(\mathrm{e}^{z2})^2} \end{aligned}$$

$$+\frac{\mathrm{D}(\mathit{V\_S1})(1+\mathrm{e}^{z1})^3(\mathrm{e}^{z1}-1)\%2}{(\mathrm{e}^{z1})^2}+\frac{D^{(2)}(\mathit{V\_S2})(1+\mathrm{e}^{z2})^4\%1}{(\mathrm{e}^{z2})^2}=0,$$

$$\frac{(1+\mathrm{e}^{z1})^4 D^{(2)}(\mathit{V\_S1})\,\mathit{V\_S2}}{(\mathrm{e}^{z1})^2}+\frac{(1+\mathrm{e}^{z1})^3(\mathrm{e}^{z1}-1)\mathrm{D}(\mathit{V\_S1})\,\mathit{V\_S2}}{(\mathrm{e}^{z1})^2}$$

$$+\frac{(1+\mathrm{e}^{z2})^4 D^{(2)}(\mathit{V\_S2})\,\mathit{V\_S1}}{(\mathrm{e}^{z2})^2}+\frac{(1+\mathrm{e}^{z2})^3(\mathrm{e}^{z2}-1)\mathrm{D}(\mathit{V\_S2})\,\mathit{V\_S1}}{(\mathrm{e}^{z2})^2}$$

$$+\frac{D^{(2)}(\mathit{V\_S1})(1+\mathrm{e}^{z1})^4\%2}{(\mathrm{e}^{z1})^2}+\frac{\mathrm{D}(\mathit{V\_S2})(1+\mathrm{e}^{z2})^3(\mathrm{e}^{z2}-1)\%1}{(\mathrm{e}^{z2})^2}$$

$$\left.+\frac{\mathrm{D}(\mathit{V\_S1})(1+\mathrm{e}^{z1})^3(\mathrm{e}^{z1}-1)\%2}{(\mathrm{e}^{z1})^2}+\frac{D^{(2)}(\mathit{V\_S2})(1+\mathrm{e}^{z2})^4\%1}{(\mathrm{e}^{z2})^2}=0\right]$$

$$,\Big[\Big[[\mathrm{V\_B1}(1,0)\,\mathrm{V\_B2}(1,0)=0,V,[-1,-1]],$$

$$[\mathrm{V\_B1}(1,0)\,\mathit{V\_S2}=0,V,[-1,0]],$$

$$[\mathrm{V\_B1}(1,0)\,\mathrm{V\_B2}(0,1)=0,V,[-1,1]],$$

$$\left[\mathit{V\_S1}\,\mathrm{V\_B2}(1,0)=\frac{(\mathrm{e}^{z1})^2}{(1+\mathrm{e}^{z1})^2},V,[0,-1]\right],$$

$$\left[\mathit{V\_S1}\,\mathrm{V\_B2}(0,1)=\sqrt{\frac{\mathrm{e}^{z1}}{1+\mathrm{e}^{z1}}},V,[0,1]\right]\Big],\Big[$$

$$[\mathrm{V\_B1}(1,0)\,\mathrm{V\_B2}(1,0)=0,V,[-1,1]],$$

$$[\mathrm{V\_B1}(1,0)\,\mathit{V\_S2}=0,V,[0,1]],$$

$$[\mathrm{V\_B1}(1,0)\,\mathrm{V\_B2}(0,1)=0,V,[1,1]],$$

$$\left[\mathit{V\_S1}\,\mathrm{V\_B2}(1,0)=\frac{1}{(1+\mathrm{e}^{z1})^2},V,[-1,0]\right],$$

$$\left[\mathit{V\_S1}\,\mathrm{V\_B2}(0,1)=\sqrt{\frac{1}{1+\mathrm{e}^{z1}}},V,[1,0]\right]\Big]\Big],[$$

$$[1,1,\mathit{HIGH},[2]]=[2,2,\mathit{LOW},[-1]],[$$

$$[\mathrm{V\_B1}(0,1)\,\mathrm{V\_B2}(1,0),V,[1,-1]]=$$

$$[\mathrm{V\_B1}(0,1)\,\mathrm{V\_B2}(0,1),V,[1,-1]],$$

$$[\mathrm{V\_B1}(0,1)\,\mathit{V\_S2},V,[1,0]]=[\mathit{V\_S1}\,\mathrm{V\_B2}(0,1),V,[0,-1]],$$

$$[\mathrm{V\_B1}(0,1)\,\mathrm{V\_B2}(0,1),V,[1,1]]=$$

$$[\mathrm{V\_B1}(1,0)\,\mathrm{V\_B2}(0,1),V,[-1,-1]]]],[\mathit{z1},\mathit{z2}],\Big[\Big[$$

$$\left[x\to\ln\left(\frac{x}{1-x}\right),z\to\frac{\mathrm{e}^z}{1+\mathrm{e}^z},x\to(x,1-x)\right],$$

$$\left[x\to\ln\left(\frac{x}{1-x}\right),z\to\frac{\mathrm{e}^z}{1+\mathrm{e}^z},x\to(x,1-x)\right]\Big],\Big[$$

$$\left[x\to\ln\left(\frac{x-1}{2-x}\right),z\to\frac{2\,\mathrm{e}^z+1}{1+\mathrm{e}^z},x\to(x-1,2-x)\right],$$

$$\left[x\to\ln\left(\frac{x-1}{2-x}\right),z\to\frac{2\,\mathrm{e}^z+1}{1+\mathrm{e}^z},x\to(x-1,2-x)\right]\Big]\Big],$$

$$[V,[[0,0],[0,0]]]\Big]$$

$$\%1 := \mathrm{V\_B1}\left(\frac{1}{1+\mathrm{e}^{z1}}, \frac{\mathrm{e}^{z1}}{1+\mathrm{e}^{z1}}\right)$$
$$\%2 := \mathrm{V\_B2}\left(\frac{1}{1+\mathrm{e}^{z2}}, \frac{\mathrm{e}^{z2}}{1+\mathrm{e}^{z2}}\right)$$

```
> type(Prob,MultiSProbType);
```

$$\mathit{true}$$

```
> with(ptolemy, Linear);
```

$$[\mathit{Linear}]$$

```
> Linear(Prob);
```

$$\mathit{true}$$

```
> Start := time(): MultiSToLinKron(Prob); time() - Start;
```

$$\Bigg[\Bigg[[\%5, \%4, \%3, \%2, \%1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], \%6,$$
$$\left[0, \mathrm{Diag}\left(\frac{(\mathrm{e}^{z2})^2}{(1+\mathrm{e}^{z2})^4}\right) \mathit{I0}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\right],$$
$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],$$
$$\left[0, 0, 0, \mathrm{Diag}\left(\frac{(\mathrm{e}^{z1})^2}{(1+\mathrm{e}^{z1})^4}\right) \mathit{I0}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\right],$$
$$\left[0, 0, 0, 0, \mathrm{Diag}\left(\frac{(\mathrm{e}^{z1})^2}{(1+\mathrm{e}^{z1})^4}\right) \mathit{I0}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\right],$$
$$[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],$$
$$[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],$$
$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],$$
$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],$$
$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \mathit{R0} \mathbin{\&\mathrm{K}} C, 0, 0, 0, 0, 0],$$
$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \mathit{R0} \mathbin{\&\mathrm{K}} C, 0, 0, 0, 0], \%6,$$
$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, C \mathbin{\&\mathrm{K}} \mathit{R0}, 0, 0, 0, 0, 0, 0, 0], \%6,$$
$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0], \%6,$$
$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]\Bigg], [\mathit{V\_1}, \mathit{V\_2},$$
$$\mathit{V\_1\_B1\_1}, \mathit{V\_1\_B1\_2}, \mathit{V\_1\_B2\_1}, \mathit{V\_1\_B2\_2}, \mathit{V\_1\_B12\_1},$$
$$\mathit{V\_1\_B12\_2}, \mathit{V\_1\_B12\_3}, \mathit{V\_1\_B12\_4}, \mathit{V\_2\_B1\_1}, \mathit{V\_2\_B1\_2},$$
$$\mathit{V\_2\_B2\_1}, \mathit{V\_2\_B2\_2}, \mathit{V\_2\_B12\_1}, \mathit{V\_2\_B12\_2}, \mathit{V\_2\_B12\_3},$$
$$\mathit{V\_2\_B12\_4}], \left[0, 0, 0, 0, \frac{(\mathrm{e}^{z1})^2}{(1+\mathrm{e}^{z1})^2}, \sqrt{\frac{\mathrm{e}^{z1}}{1+\mathrm{e}^{z1}}}, 0, 0, 0, 0, \frac{1}{(1+\mathrm{e}^{z1})^2},\right.$$
$$\left.\sqrt{\frac{1}{1+\mathrm{e}^{z1}}}, 0, 0, 0, 0, 0, 0\right], [z1, z2], [\mathit{H1}, \mathit{H2}],$$
$$[[[1, 1], [2, 2]], [[1, 2]], [[2, 1]]]\Bigg]$$
$$\%1 := \mathrm{Diag}\left(\frac{\mathrm{e}^{z2}}{z1^2\,(1+\mathrm{e}^{z2})}\right)(\mathit{I2} \mathbin{\&\mathrm{K}} C)$$

$$+\operatorname{Diag}\left(\frac{(e^{z1}-1)\,e^{z2}}{(1+e^{z1})\,z1\,(1+e^{z2})}\right)(\mathit{I1}\ \&\mathrm{K}\ C)$$

$$\%2 := \operatorname{Diag}\left(\frac{1}{z1^2\,(1+e^{z2})}\right)(\mathit{I2}\ \&\mathrm{K}\ C)$$

$$+\operatorname{Diag}\left(\frac{e^{z1}-1}{(1+e^{z1})\,z1\,(1+e^{z2})}\right)(\mathit{I1}\ \&\mathrm{K}\ C)$$

$$\%3 := \operatorname{Diag}\left(\frac{(e^{z2}-1)\,e^{z1}}{(1+e^{z2})\,z2\,(1+e^{z1})}\right)(C\ \&\mathrm{K}\ \mathit{I1})$$

$$+\operatorname{Diag}\left(\frac{e^{z1}}{z2^2\,(1+e^{z1})}\right)(C\ \&\mathrm{K}\ \mathit{I2})$$

$$\%4 := \operatorname{Diag}\left(\frac{e^{z2}-1}{(1+e^{z2})\,z2\,(1+e^{z1})}\right)(C\ \&\mathrm{K}\ \mathit{I1})$$

$$+\operatorname{Diag}\left(\frac{1}{z2^2\,(1+e^{z1})}\right)(C\ \&\mathrm{K}\ \mathit{I2})$$

$$\%5 := \operatorname{Diag}\left(\frac{(e^{z2})^2}{z1^2\,(1+e^{z2})^4}\right)(\mathit{I2}\ \&\mathrm{K}\ \mathit{I0})$$

$$+\operatorname{Diag}\left(\frac{(e^{z1}-1)\,(e^{z2})^2}{(1+e^{z1})\,z1\,(1+e^{z2})^4}\right)(\mathit{I1}\ \&\mathrm{K}\ \mathit{I0})$$

$$+\operatorname{Diag}\left(\frac{(e^{z1})^2}{(1+e^{z1})^4\,z2^2}\right)(\mathit{I0}\ \&\mathrm{K}\ \mathit{I2})$$

$$+\operatorname{Diag}\left(\frac{(e^{z2}-1)\,(e^{z1})^2}{(1+e^{z2})\,(1+e^{z1})^4\,z2}\right)(\mathit{I0}\ \&\mathrm{K}\ \mathit{I1})$$

$$\%6 := [0,0,0,0,0,0,0,0,0,\%5,\%4,\%3,\%2,\%1,0,0,0,0]$$

1.716

End of Maple Worksheet

Method of Implementation

Just as for the procedure **SToLinKron**, the core operations are performed by procedures from the ***kron_ops*** LLF. However, unlike **SToLinKron**, these core operations no longer represent the bulk of the code. The outline and even much of the logic of **MultiSToLinKron** are the same as for **SToLinKron**, but the extra complications of multiple subdomains and coupling equations is significant.

The key differences between the implementation of this procedure and that of **SToLinKron** are:

- The stack variable names are of the form `V_.Domain._B.i._.j` where `i` and `j` are defined the same as for a single domain problem and `Domain` is an integer indicating the domain number.
- Each call to **kron_eq** or **kron_exp** converts an equation or expression to Kronecker product notation within the context of a single subdomain. This means that only a portion of the block-row is constructed for each call to a procedure from ***kron_ops***.

- Constructing the parameter component list is no longer trivial. The graph indicating which parameters are constrained to be equivalent must be constructed and then this graph must be searched in order to identify all of its components. Fortunately, much of this work is done with the aid of Maple's ***networks*** package.

Partial Row Construction Because the procedures in ***kron_ops*** all operate in the context of a single domain, this procedure constructs intra-subdomain stack variables for each subdomain. These stack variable names are used to construct the portion of each block-row that references the unknowns corresponding to a single subdomain.

For block-rows corresponding to governing equations or boundary constraints all of the unknowns referenced by the block-row will be within a single subdomain. That is, the rest of the blocks in the block-row will be zero. In the case of coupling equations, each side of the equation produces a portion of the block-row. These two portions of the block-row must be embedded among the potential zero-blocks corresponding to unknowns from other subdomains.

Coupling Equations In order to construct override block-rows that correspond to the coupling equations the procedure must first extract the coupling orientation from each *CoupleOverType* in the collocation system. This information indicates the domain of application and the relative orientation of the coordinates on each side of the coupling. Once this information has been extracted the procedure then converts each half of every coupling equation which is part of the current coupling to Kronecker product notation by calling **kron_exp** (form the ***Kron_ops*** LLF). Each half is inserted into the appropriate segment of the full length block-row.

The parameterization of the boundary (i.e., part of the *CoupleOrientType* in each *CoupleOverType*) is used to construct the `RowCoord` argument to **kron_exp**. However, this parameterization information cannot be used directly for the `RowCoord` argument except in the central region of each boundary. In other regions of the boundary some or even all of the coordinates may not vary over the rows of the current block-row. The collocation point specified in each half of the coupling equation is used to determine the region of application, which in turn is used to determine which elements of the parameterization of the boundary must be eliminated from the `RowCoord` argument.

Finally, the collocation point in the domain with the lower domain number is used to determine the row number. The procedure **b_point_to_num** (in the ***b_ops*** LLF) is called to determine the information needed to construct the stack variable name.

Parameter Components As mentioned in the introduction to this subsubsection most of the work of constructing the parameter components is performed by the Maple's ***networks*** package. However, the order in which the **components** procedure reports the components in nondeterministic. This would normally force the user to check the order of the components after every use of this procedure, even if nothing material to the parameter component definition has changed. To prevent this the procedure sorts the components returned by **components** so that the order is consistent between runs.

In order to sort the parameter components it is necessary to first define an order with respect to the parameters. **PTOLEMY** orders parameters descriptions using the domain number as the primary key and the dimension number as the secondary key. Using this ordering all of the parameters within each parameter component are first sorted, then the parameter components are sorted by comparing the "lowest" parameter in each component.

Dependencies

This procedure calls **ptolemy/multi_spec_to_list** from the ***order_ops*** LLF in order to convert the order information into a list of subdomain-specific order specification.

The procedure also calls **get_SB_var** to get a list of `S`-variables occurring in the governing equation so that it can call **pde_order** to determine the order of the governing equations. The procedure **order_subtract** is then used to determine the correct order for the weights, which are constructed by calling **mapped_weight** (from the ***map_info_ops*** LLF).

The procedure **all_comb** (from the ***comb_ops*** LLF) is used to construct the stack variable names.

Each governing equation and each boundary constraint is converted from `SB`-notation to Kronecker product notation by calling the procedure **kron_eq** (from the ***kron_ops*** LLF). Similarly, each half of each coupling equation is converted by calling **kron_exp** (also from the ***kron_ops*** LLF). In the case of boundary constraints and coupling equations the **b_point_to_num** procedure (from the ***b_ops*** LLF) is called to construct information needed to determine the block-row number.

On the chance that it might further simplify the result, the procedure specified by the global variable `ptolemy/SimpProc` is invoked on the sum of the RHSs of the matrix problem produced by the two halves of the coupling equation.

The procedure **cross_prod** is used to construct all parameter descriptions which form the nodes of the graph used to compute the parameter components.

5.8 File Format for Matrix Problem Descriptions

Because the range of options for the numerical solution of matrix problem generated by **PTOLEMY** is so large, it is important that the user have the option of using other systems for this part of the process. To make interfacing with other numerical linear algebra systems more practical **PTOLEMY** defines a portable file format for symbolically describing matrix problems of the form arising in sinc-collocation.

The primary intent of this file-based matrix description is to allow the same problem to be solved in many different numerical environments, some of which have little knowledge of **PTOLEMY**. A significant advantage of this file format is that because it is a symbolic description of the matrix it is typically a much more compact representation than the full numerical instantiation of the matrix.

Expression Format

Each expression must be an element of the grammar defined in Table 5.1. where `<Number>` represents an integer, fixed point, or floating point number, `<Name>` represents one of the 'H'-parameters or one of the coordinate names, and `<Func>` represents a mathematical function name.

Currently, **PTOLEMY**'s standard numerical solver knows about relatively few mathematical functions, but it is easy to define new mathematical functions. The user should not feel constrained by this set of named mathematical functions. However, the user should feel constrained by common sense. The usefulness of this file format is dependent on its portability which is contingent on using widely recognized function names and functions that are common enough for efficient numerical evaluation routines to exist.

It is hoped that the OpenMath project (see [4]) will develop a better solution to this problem. The current proposals from the project include phrase-books for defining translation between different naming schemes that could, with sufficient acceptance, partially solve this problem.

Stack Descriptions

Stacks are vectors whose elements are constructed from a higher- (or potentially higher-) dimensional uniform grid using some cartesian ordering. Stacks are defined by a function over this higher dimensional grid. The file format being

Table 5.1: The Grammar of an Expression

Symbol	Production
`<ExpSeq>`	`<Exp> | <Exp>, <Exp>`
`<Exp>`	`<Term> | <Term> + <Exp>`
`<Term>`	`<Fact> | <Fact>*<Term>`
`<Fact>`	`<SubFact> | <SubFact>^<SubFact>`
`<Sub>`	`<Number> | <Name> | (<Exp>) | <Func>(<ExpSeq>)`

defined in this section allows the expression used to define the stack to be either a single expression or a list of factors of the expression, where each factor varies only with respect to one of the coordinates. The grammar for these two kinds of stack definitions are shown in Table 5.2 where `<WS>` is white space and `<Dim>` is a integer specifying the (one based) dimension number corresponding to the factor. If the dimension number is zero then the factor describes a constant factor, independent of all coordinates.

Matrix Block Descriptions

The grammar for the block definition is shown in Table 5.3 Here `<RowNum>` and `<ColumnNum>` are the (zero based) row and column numbers of the block. The symbol `<Dimen>` is an integer which indicates the number of dimensions that vary along *either* the rows or columns of the block. This is also equivalent to the number of matrices and vectors that appear in the Kronecker product sequence. The `<Type>` symbols in the `<TypeSeq>` construct indicates whether the corresponding matrix in the Kronecker product is square (i.e., `M`) or a vector (i.e., `R` for a row vector and `C` for a column vector).

The symbol `<IOrd>` indicates the order of the 'I'-matrix in the Kronecker product sequence. There should be only one `<IOrd>` in the sequence for each square matrix in the sequence, not necessarily one for every dimension.

The `<Stack>` symbol defines a diagonal matrix to premultiply the other matrices. The optional `<Perm>` symbol indicates a permutation matrix to precede the Kronecker sequence. It indicates the order in which the dimensions are to appear in the cartesian ordering of the rows. The symbol `<Dim>` is an integer, indicating the (zero based) dimension number; there should be one `<Dim>` in the sequence for every dimension that varies over the rows of block.

Matrix Problem Descriptions

The grammar for an entire matrix problem description is shown in Table 5.4. In this grammar `<Coord>` represents a coordinate name and `<HName>` represents the name of an 'H'-parameter. The symbols `<NumMainVar>` and `<NumExtra>` are positive integers indicating the number of main stack variables names and the number of extra stack variable respectively. Each extra stack variable name

Table 5.2: The Grammar for a Stack Description

Symbol	Production
`<Stack>`	`<SimpStack> | <FactStack>`
`<SimpStack>`	`<Exp>`
`<FactStack>`	`[<FactSeq>]`
`<FactSeq>`	`<Fact> | <Fact><WS><Fact>`
`<Fact>`	`<Dim><WS><Exp>`

Table 5.3: The Grammar for a Stack Description

Symbol	Production
`<Block>`	`<ElemNum><WS><DimInfo><WS><MatSpec>`
`<ElemNum>`	`<RowNum><WS><ColumnNum>`
`<DimInfo>`	`<Dimen><WS><TypeSeq>`
`<TypeSeq>`	`<Type> | <Type><WS><TypeSeq>`
`<Type>`	`M | R | C`
`<MatSpec>`	`<Stack><WS><IOrdSeq> |`
	`  <Stack><WS><Perm><WS><IOrdSeq>`
`<Perm>`	`<P(<NumSeq>)>`
`<DimSeq>`	`<Dim> | <Dim>,<DimenSeq>`
`<IOrdSeq>`	`<NULL> | <IOrd> | <IOrd><WS><IOrdSeq>`

Table 5.4: The Grammar for a Matrix Problem Description

Symbol	Production
`<Prob>`	`<Dimen><WS>`
	`<Names><WS><StackSize><WS>`
	`<ExtraSeq><WS><CompInfo><WS>`
	`<NumBlock><WS><BlockSeq>`
`<Names>`	`<CoordSeq><WS><HSeq>`
`<CoordSeq>`	`<Coord> | <Coord><WS><CoordSeq>`
`<HSeq>`	`<HName> | <HName><WS><HSeq>`
`<StackSize>`	`<NumMainVar><WS><NumExtra>`
`<ExtraSeq>`	`<ExtraInfo> | <ExtraInfo><WS><ExtraSeq>`
`<ExtraInfo>`	`<Dimen><WS><CoordSeq><WS><HSeq>`
`<DimSeq>`	`<Dim> | <Dim><WS><DimSeq>`
`<CompInfo>`	`<NumOfComp><WS><CompSeq>`
`<CompSeq>`	`<Comp> | <Comp><WS><CompSeq>`
`<Comp>`	`<CompSize><WS><ParamSeq>`
`<ParamSeq>`	`<Param> | <Param><WS><ParamSeq>`
`<Param>`	`<Domain><WS><Dim>`
`<BlockSeq>`	`<Block> | <Block><WS><BlockSeq>`

will range over some subset of the dimensions, and may even use different coordinate names and 'H'-parameter names.

The symbol <`ExtraInfo`> defines the dimensionality and variable names used for defining each extra stack variable and the block-column corresponding to this stack variable. The integer represented by <`Dimen`> specifies the number of dimensions which vary over the corresponding stack variable, and <`CoordSeq`> and <`HSeq`> specify the parameter names used in the symbolic description of the blocks.

The parameter components are specified by fields represented by the symbol <`Comp`>. The symbol <`NumOfComp`> represents a positive integer specifying the number of parameter components. The symbol <`CompSize`> represents a positive integer which specifies the number of parameters in the current parameter component. The symbols <`Domain`> and <`Dim`> specify the (zero based) number of the domain and dimension, respectively.

SolveLinKron

This procedure first writes the symbolic form of the matrix problem to a file. It then spawns a process which runs the C++ command returned by the procedure **ptolemy/solve_command**.

SolveLinKron(`Prob`: *LinKronType*, `N`: *list(posint)*, `H`: *list(numeric)*,
`BaseFileName`: *name*)

The argument `Prob` specifies a matrix problem in Kronecker product notation. The arguments `N` and `H` specify the simulation parameters; each parameter in each list corresponds to one parameter component. The order of the elements of `N` and `H` is determined by the order of the parameter components in `Prob`. Finally, the argument `BaseFileName` provides the base file name to be used for constructing the problem statement and the problem solution. The problem statement will be written to a file formed by concatenating '`.PROB`' to this base filename and, as a matter of convention, the solution should be written to the file with the name formed by concatenating '`.SOL`' to the base file name.

The Interface to solve_command

The **solve_command** procedure must have an interface of

solve_command(`BaseName`: *name*, `N`: *list(posint)*, `H`: *list(numeric)*)

The arguments are the same as the last three arguments to **SolveLinKron**. The result returned by **solve_command** is a string defining the command to be executed in the current environment to solve the problem.

If the global variable **ptolemy/solve_command** has not been assigned then it is assigned the procedure in Figure 5.5 during package initialization. The purpose of using this more complicated mechanism for constructing the command is to allow the user to redefine the command according to the needs of the particular computing environment or according to the needs of nontraditional applications.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy,LogRatioMap,CollocateRecProb, SToLinKron, SolveLinKron);
```

$$[\, CollocateRecProb, LogRatioMap, SToLinKron, SolveLinKron \,]$$

```
> Sys := D[1,1](V) + D[2,2](V) = K;
```

$$Sys := D_{1,1}(V) + D_{2,2}(V) = K$$

```
macro(TO_STRING = readlib(‘ptolemy/to_string‘));

‘ptolemy/solve_command‘ := proc(
  BaseName: name, N: list(posint), H: list(numeric))

local NumOfParam, DQ;

  DQ := substring(‘’"’‘, 2..2);
  NumOfParam := nops(N);

  cat( ‘ptolemy_solve ‘, BaseName, ‘.prob ‘, BaseName, ‘.sol ‘,
    DQ, TO_STRING(H)), DQ, ‘ ‘, DQ, TO_STRING(N)), DQ);
end;
```

Figure 5.5: The Default Value of **solve_command**

Maple Worksheet Continued from Previous Page

```
> Bound := {[1,LOW,V=0],[1,HIGH,V=0],[2,LOW,V=0],[2,HIGH,V=0]};
```

$$\mathit{Bound} := \{[1, \mathit{LOW}, V = 0], [1, \mathit{HIGH}, V = 0], [2, \mathit{LOW}, V = 0], [2, \mathit{HIGH}, V = 0]\}$$

```
> Map := LogRatioMap(0,1,x,z);
```

$$\mathit{Map} := \left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (x, 1-x)\right]$$

```
> RecProb := [Sys,Bound,[x1,x2], [Map,Map], {[V,[0,0]]}];
```

$$\mathit{RecProb} := \left[D_{1,1}(V) + D_{2,2}(V) = K, \{[1, \mathit{LOW}, V = 0], [1, \mathit{HIGH}, V = 0], [2, \mathit{LOW}, V = 0], [2, \mathit{HIGH}, V = 0]\}, [\mathit{x1}, \mathit{x2}], \left[\left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (x, 1-x)\right], \left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (x, 1-x)\right]\right], \{[V, [0, 0]]\}\right]$$

```
> type(RecProb, RecProbType);
```

$$\mathit{true}$$

```
> SProb := CollocateRecProb(RecProb, [z1,z2], 'ExtraBases');
```

$$\mathit{SProb} := \Big[\mathrm{D}(\mathit{V\_S1})(-2\,e^{(-\mathit{z1})} + 2\,e^{\mathit{z1}} + e^{(2\,\mathit{z1})} - e^{(-2\,\mathit{z1})})\,\mathit{V\_S2}$$
$$+ \mathit{V\_S1}\,\mathrm{D}(\mathit{V\_S2})(-2\,e^{(-\mathit{z2})} + 2\,e^{\mathit{z2}} + e^{(2\,\mathit{z2})} - e^{(-2\,\mathit{z2})})$$
$$+ (e^{(-2\,\mathit{z2})} + 4\,e^{(-\mathit{z2})} + 6 + 4\,e^{\mathit{z2}} + e^{(2\,\mathit{z2})})\,D^{(2)}(\mathit{V\_S2})\,\mathit{V\_S1}$$
$$+ \mathrm{V\_B1}\left(\frac{1}{1+e^{\mathit{z1}}}, \frac{e^{\mathit{z1}}}{1+e^{\mathit{z1}}}\right)\mathrm{D}(\mathit{V\_S2})(-2\,e^{(-\mathit{z2})} + 2\,e^{\mathit{z2}} + e^{(2\,\mathit{z2})} - e^{(-2\,\mathit{z2})})$$
$$+ \mathrm{V\_B2}\left(\frac{1}{1+e^{\mathit{z2}}}, \frac{e^{\mathit{z2}}}{1+e^{\mathit{z2}}}\right)\mathrm{D}(\mathit{V\_S1})(-2\,e^{(-\mathit{z1})} + 2\,e^{\mathit{z1}} + e^{(2\,\mathit{z1})} - e^{(-2\,\mathit{z1})})$$

$$+\left(\mathrm{e}^{(-2\,z2)}+4\,\mathrm{e}^{(-z2)}+6+4\,\mathrm{e}^{z2}+\mathrm{e}^{(2\,z2)}\right)D^{(2)}(\,V\_S2\,)$$
$$\mathrm{V\_B1}\left(\frac{1}{1+\mathrm{e}^{z1}},\frac{\mathrm{e}^{z1}}{1+\mathrm{e}^{z1}}\right)+\left(\mathrm{e}^{(-2\,z1)}+4\,\mathrm{e}^{(-z1)}+6+4\,\mathrm{e}^{z1}+\mathrm{e}^{(2\,z1)}\right)$$
$$D^{(2)}(\,V\_S1\,)\,\mathrm{V\_B2}\left(\frac{1}{1+\mathrm{e}^{z2}},\frac{\mathrm{e}^{z2}}{1+\mathrm{e}^{z2}}\right)$$
$$+\left(\mathrm{e}^{(-2\,z1)}+4\,\mathrm{e}^{(-z1)}+6+4\,\mathrm{e}^{z1}+\mathrm{e}^{(2\,z1)}\right)V\_S2\,D^{(2)}(\,V\_S1\,)=K,$$
$$[\,z1,z2\,],\{\,V\_S2\,\mathrm{V\_B1}(\,1,0\,)=0,\,V\_S2\,\mathrm{V\_B1}(\,0,1\,)=0,$$
$$V\_S1\,\mathrm{V\_B2}(\,1,0\,)=0,\,V\_S1\,\mathrm{V\_B2}(\,0,1\,)=0\},\Bigg[$$
$$\left[x\to\ln\left(\frac{x}{1-x}\right),z\to\frac{\mathrm{e}^{z}}{1+\mathrm{e}^{z}},x\to(\,x,1-x\,)\right],$$
$$\left[x\to\ln\left(\frac{x}{1-x}\right),z\to\frac{\mathrm{e}^{z}}{1+\mathrm{e}^{z}},x\to(\,x,1-x\,)\right]\Bigg],[[V,[\,0,0\,]]]\Bigg]$$

```
> type(SProb,SProbType);
```

$$true$$

```
> LinKron := SToLinKron(SProb);
```

$$LinKron:=\Bigg[\Bigg[\Bigg[\mathrm{Diag}\left(\frac{\%2\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2}{H1}\right)(\,I1\ \&\mathrm{K}\ I0\,)$$
$$+\,\mathrm{Diag}\left(\frac{\%4\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2}{H2}\right)(\,I0\ \&\mathrm{K}\ I1\,)$$
$$+\,\mathrm{Diag}\left(\frac{\%3\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2}{H2^2}\right)(\,I0\ \&\mathrm{K}\ I2\,)$$
$$+\,\mathrm{Diag}\left(\frac{\%1\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2}{H1^2}\right)(\,I2\ \&\mathrm{K}\ I0\,),$$
$$\mathrm{Diag}\left(\frac{\%4\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2}{1+\mathrm{e}^{z1}}\right)(\,I0\ \&\mathrm{K}\ I0\,)$$
$$+\,\mathrm{Diag}\left(\frac{\%3\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2}{1+\mathrm{e}^{z1}}\right)(\,I0\ \&\mathrm{K}\ I0\,),$$
$$\mathrm{Diag}\left(\frac{\%4\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2\,\mathrm{e}^{z1}}{1+\mathrm{e}^{z1}}\right)(\,I0\ \&\mathrm{K}\ I0\,)$$
$$+\,\mathrm{Diag}\left(\frac{\%3\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2\,\mathrm{e}^{z1}}{1+\mathrm{e}^{z1}}\right)(\,I0\ \&\mathrm{K}\ I0\,),$$
$$\mathrm{Diag}\left(\frac{\%2\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2}{1+\mathrm{e}^{z2}}\right)(\,I0\ \&\mathrm{K}\ I0\,)$$
$$+\,\mathrm{Diag}\left(\frac{\%1\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2}{1+\mathrm{e}^{z2}}\right)(\,I0\ \&\mathrm{K}\ I0\,),$$
$$\mathrm{Diag}\left(\frac{\%2\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2\,\mathrm{e}^{z2}}{1+\mathrm{e}^{z2}}\right)(\,I0\ \&\mathrm{K}\ I0\,)$$
$$+\,\mathrm{Diag}\left(\frac{\%1\,z1^2\,(\,1-z1\,)^2\,z2^2\,(\,1-z2\,)^2\,\mathrm{e}^{z2}}{1+\mathrm{e}^{z2}}\right)(\,I0\ \&\mathrm{K}\ I0\,)\Bigg],$$
$$\left[0,\mathrm{Diag}\left(z2^2\,(\,1-z2\,)^2\right)\,I0,0,0,0\right],$$
$$\left[0,0,\mathrm{Diag}\left(z2^2\,(\,1-z2\,)^2\right)\,I0,0,0\right],$$

$$[0,0,0,\mathrm{Diag}\left(z1^2\,(1-z1\,)^2\right)\,I0,0]\,,[0,0,0,0,\mathrm{Diag}\left(z1^2\,(1-z1\,)^2\right)\,I0]$$

$$\Big]\,,[V,\,V\_B1\_1,\,V\_B1\_2,\,V\_B2\_1,\,V\_B2\_2\,],[K,0,0,0,0\,],[\,z1\,,\,z2\,],$$

$$[\,H1,H2\,],[[[1,[2\,]]],[[1,[2\,]]],[[1,[1\,]]],[[1,[1\,]]]]\Big]$$

$$\%1 := \mathrm{e}^{(-2\,z1\,)}+4\,\mathrm{e}^{(-z1\,)}+6+4\,\mathrm{e}^{z1}+\mathrm{e}^{(2\,z1\,)}$$
$$\%2 := -2\,\mathrm{e}^{(-z1\,)}+2\,\mathrm{e}^{z1}+\mathrm{e}^{(2\,z1\,)}-\mathrm{e}^{(-2\,z1\,)}$$
$$\%3 := \mathrm{e}^{(-2\,z2\,)}+4\,\mathrm{e}^{(-z2\,)}+6+4\,\mathrm{e}^{z2}+\mathrm{e}^{(2\,z2\,)}$$
$$\%4 := -2\,\mathrm{e}^{(-z2\,)}+2\,\mathrm{e}^{z2}+\mathrm{e}^{(2\,z2\,)}-\mathrm{e}^{(-2\,z2\,)}$$

```
> type(LinKron, LinKronType);
```

true

```
> H := [ evalf(1/sqrt(2)), evalf(1/sqrt(3)) ];
```

$H := [\,.7071067810,\,.5773502693\,]$

```
> SolveLinKron(subs(K=1, LinKron), [[2,3]], [H], test);
SPAWNING:  ptolemy_solve test.prob test.sol "[[0.7071067810 0.5773502693 ] \
]" "[[2 3 ] ]"
```

End of Maple Worksheet

This example will create the file called **TEST.PROB** with constants shown in Figure 5.6. It will then spawn a process running **PTOLEMY_SOLVE** which will read this file and use it to build and solve a matrix problem specified in **TEST.PROG**; the solution will then be written to **TEST.SOL**.

Known Problems

If an expression is long, Maple's `write` command will split it into multiple lines. Long line breaks occur at any place which is acceptable for Maple input, includes places where white space is not allowed by the file format defined in Section 5.8. This problem is addressed by the improved I/O capabilities available in the next version of Maple (i.e., Version V; Release 4), but in the meantime the user will sometimes need to edit files created by **SolveLinKron** and to manually run **PTOLEMY_SOLVE**.

```
2
z1  z2
H1  H2
1  4

1  z2  H2  1  0  1
1  z2  H2  1  0  1
1  z1  H1  1  0  0
1  z1  H1  1  0  0

9
0  0
2  M  M
[  0  -1/H1  1  (2*exp(-z1)-2*exp(z1)-exp(2*z1)+exp(-2*z1))*z1^2*(-1+z1)^2 2  z2^2*(-1+z2)^2  ]  1  0
[  0  1/H2  1  z1^2*(-1+z1)^2  2  (-2*exp(-z2)+2*exp(z2)+exp(2*z2)-exp(-2*z2))*z2^2*(-1+z2)^2  ]  0  1
[  0  1/H2^2  1  z1^2*(-1+z1)^2  2  (exp(-2*z2)+4*exp(-z2)+6+4*exp(z2)+exp(2*z2))*z2^2*(-1+z2)^2  ]  0  2
[  0  1/H1^2  1  (exp(-2*z1)+4*exp(-z1)+6+4*exp(z1)+exp(2*z1))*z1^2*(-1+z1)^2  2  z2^2*(-1+z2)^2  ]  2  0

0  1
1  R  M
[  1  z1^2*(-1+z1)^2/(1+exp(z1))  2  (-2*exp(-z2)+2*exp(z2)+exp(2*z2)-exp(-2*z2))*z2^2*(-1+z2)^2  ]  0  0
[  1  z1^2*(-1+z1)^2/(1+exp(z1))  2  (exp(-2*z2)+4*exp(-z2)+6+4*exp(z2)+exp(2*z2))*z2^2*(-1+z2)^2  ]  0  0

0  2
1  R  M
[  1  z1^2*(-1+z1)^2*exp(z1)/(1+exp(z1))  2  (-2*exp(-z2)+2*exp(z2)+exp(2*z2)-exp(-2*z2))*z2^2*(-1+z2)^2  ]  0  0
[  1  z1^2*(-1+z1)^2*exp(z1)/(1+exp(z1))  2  (exp(-2*z2)+4*exp(-z2)+6+4*exp(z2)+exp(2*z2))*z2^2*(-1+z2)^2  ]  0  0

0  3
1  M  R
[  0  -1  1  (2*exp(-z1)-2*exp(z1)-exp(2*z1)+exp(-2*z1))*z1^2*(-1+z1)^2  2  z2^2*(-1+z2)^2/(1+exp(z2))  ]  0  0
[  1  (exp(-2*z1)+4*exp(-z1)+6+4*exp(z1)+exp(2*z1))*z1^2*(-1+z1)^2  2  z2^2*(-1+z2)^2/(1+exp(z2))  ]  0  0

0  4
1  M  R
[  0  -1  1  (2*exp(-z1)-2*exp(z1)-exp(2*z1)+exp(-2*z1))*z1^2*(-1+z1)^2  2  z2^2*(-1+z2)^2*exp(z2)/(1+exp(z2))  ]  0  0
[  1  (exp(-2*z1)+4*exp(-z1)+6+4*exp(z1)+exp(2*z1))*z1^2*(-1+z1)^2  2  z2^2*(-1+z2)^2*exp(z2)/(1+exp(z2))  ]  0  0

1  1
1  M
[  2  z2^2*(-1+z2)^2  ]  0

2  2
1  M
[  2  z2^2*(-1+z2)^2  ]  0

3  3
1  M
[  1  z1^2*(-1+z1)^2  ]  0

4  4
1  M
[  1  z1^2*(-1+z1)^2  ]  0

1
0
0
0
0
```

Figure 5.6: The Contents of the File **TEST.PROB**

5.10 The C++ Library

Initially most users may wish to implement their own solver in order to experiment with different schemes for exploiting the structure of the matrix. Such experimentation may be done easily with the aid of tools such as MATLAB.

Unfortunately most of these tools do not have the functionality necessary for converting the symbolic description of the matrix to a numerical approximation of the matrix (i.e., to build the numerical matrix). One simple solution to this problem is to build the numerical matrix in Maple and then write the result to disk. This solution is inefficient because Maple uses hash table to represent arrays. The author has build a fast interpreter for computing floating point approximations of a subset of Maple expressions and the file format described in Section 5.8. The result is about 10,000 times faster then a direct implementation in Maple.

Users interested in experimenting with this method may down load the code from

`http://daisy.uwaterloo.ca/~kparker/ptolemy`

However, experimentation suggests that for matrices that are small enough to fit in memory using Maple's representation, the slower Maple evaluation is adequate. This is because the total time required to build the matrix is small compared to the time required to solve the matrix; so even extreme inefficiency has little affect on the total process time. However, on many systems building sufficiently large matrices in Maple (large enough to cause thrashing) exploits bugs in either the operating system or Maple causing one or the other to crash. In such cases using a custom interpreter appears to be the only way to build the numerical matrix.

Chapter 6

Debugging

This chapter describes tools for identifying bugs in the use of **PTOLEMY**. The current version of Maple (i.e., Version 5, Release 3) is severely limited in its support of traditional debugging tools. The next version, release 4, will provide a traditional debugger that I have found invaluable for developing many types applications.

However, traditional debugging tools are not as powerful for finding bugs in computational software as they are for software dominated by logic. This is because in logic-oriented software most bugs cause the program to follow an unexpected execution path. As a result, providing the ability to pose questions about the execution path gives the user the ability to collect powerful hints about the cause of a bug. In contrast most bugs in numerically intensive programs do not alter the execution path. Bugs in computational software most often take the form of an implementation of incorrect formulas. Subtle examples of this kind of bug include the application of a valid formula to a situation where the formula is not applicable and the use of formulas that are algebraically correct but produce unacceptable amounts of round-off.

Empirical evidence suggests that when bugs in the formula being applied are not obvious from a visual inspection of the code, the most useful debugging information is a demonstration of the formula's effect on various data sets. Almost all problems of interest have too much data to examine the numerical values directly. As a result this kind of debugging information must primarily involve the graphical visualization of data sets.

The kind of visualization most useful for debugging typically requires more than just graphing raw data. What is most often needed is the graphical visualization of some user defined manipulation of the data. A common simple example would be to graph the difference between two data sets; a more complex example might be to graph the Discrete Fourier Transform (DFT) of a data set. However, in most cases the manipulation of the data to be performed as part of the visualization is understood by the user in terms of a symbolically defined mathematical operation. This means that Maple has unusual potential to provide such nontraditional debugging support.

This portion of the **PTOLEMY** package demonstrates some of the potential for this kind of debugging support. Like the numerical capabilities discussed in Chapter 5, **PTOLEMY** provides more of a demonstration of the potential than a full featured debugging environment.

PlotBound

This function graphs a boundary or a collection of boundaries.

PlotBound(`Bound`: *collection(BoundType)*, `PlotOptions`: *seq*)

The argument `Bound` specifies the boundary(s) to be graphed. The argument `PlotOptions` represents a possibly empty sequence of plotting options. These plotting options allow the user to directly to specify certain plotting options to whatever graphing procedure is used to produce the actual graphs, i.e., at this time either `plot` or `plot3d`.

When the embedded space is one-dimensional the procedure displays the boundaries as vertical lines that are intended to illustrate points on a "number line." In this case the global variable `ptolemy/OneDRange` specifies the vertical range used to depict the boundaries. It should be assigned an expression that evaluates to type ***range(numeric)*** at the time of procedure invocation. During package initialization the value of `ptolemy/OneDRange` is set to `-1/2..1/2`, if the variable is not already defined.

Limitation

The collection boundaries must all be embedded in the same space. That is all of the boundaries must be structurally of the same dimensionality. Boundaries need not actually vary in all, or even any, of the dimensions of the space, but the behavior of the boundary must be explicitly specified for each dimension in the space.

In addition warnings are generated for boundaries in spaces of dimensionality greater than three. This restriction is not intrinsic; for example four-dimensional spaces might be handled via animation or projection. However, the graphing capability of Maple is sufficiently limited so that it is unclear how useful this feature would be.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, PlotBound);
```

$$[PlotBound]$$

```
> D1 := { [0], seq([1/2^i / (1 + 1/2^i)], i=-3..3), [1]};
```

$$D1 := \left\{\left[\frac{2}{3}\right], \left[\frac{1}{2}\right], \left[\frac{1}{3}\right], \left[\frac{1}{5}\right], \left[\frac{1}{9}\right], [0], [1], \left[\frac{8}{9}\right], \left[\frac{4}{5}\right]\right\}$$

```
> type(D1,set(BoundType));
```

$$true$$

```
> 'ptolemy/OneDRange' := -0.1..0.1;
```

$$ptolemy/OneDRange := -.1...1$$

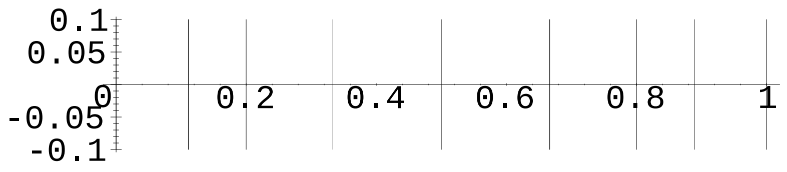

Figure 6.1: An Eight Subdomain Partitioning of the Interval $[0, 1]$

Maple Worksheet Continued from Previous Page

```
> PlotBound(D1);
```
See Figure 6.1.

```
> D2 :=
>   { [theta*cos(Pi*theta), theta*sin(Pi*theta), theta=0..2],
>     [theta*cos(Pi*(theta + 1/2)), theta*sin(Pi*(theta + 1/2)), theta=0..2],
>     [2*cos(Pi*theta), 2*sin(Pi*theta), theta=0..1/2] };
```

$$D2 := \Big\{[\theta\cos(\pi\,\theta), \theta\sin(\pi\,\theta), \theta = 0..2],$$
$$\left[\theta\cos\left(\pi\,\left(\theta+\frac{1}{2}\right)\right), \theta\sin\left(\pi\,\left(\theta+\frac{1}{2}\right)\right), \theta = 0..2\right],$$
$$\left[2\cos(\pi\,\theta), 2\sin(\pi\,\theta), \theta = 0..\frac{1}{2}\right]\Big\}$$

```
> PlotBound(D2, tickmarks=[4,4]);
```
See Figure 6.2.

```
> D3 := { [t1,1.1,t2,t1=0..1,t2=0..1], [1.1,t1,t2,t1=0..1,t2=0..1],
>   [1 - t1*(1+t2),1 - t1*(1-t2),1.1,t1=0..1/2,t2=-1..1] };
```

$$D3 := \Big\{[1.1, t1, t2, t1 = 0..1, t2 = 0..1],$$

Maple Worksheet Continued on Next Page

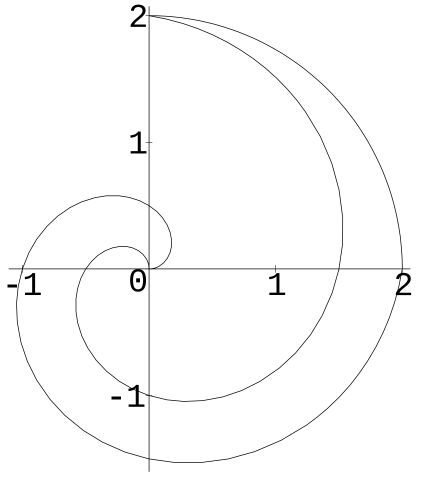

Figure 6.2: A Two-Dimensional Spiral Domain

$$\left[1 - t1\,(1 + t2\,), 1 - t1\,(1 - t2\,), 1.1, t1 = 0..\frac{1}{2}, t2 = -1..1\right],$$
$$[\,t1, 1.1, t2, t1 = 0..1, t2 = 0..1]\Big\}$$

```
> PlotBound(D3, orientation=[30,70], axes=boxed, tickmarks=[3,3,3]);
```

See Figure 6.3

End of Maple Worksheet

Method of Implementation

If the argument `Bond` represents a single boundary then the dimensionality of the space it determined and a graph of the boundary is produced. However, if `Bound` represents a collection of boundaries then **PlotBound** is called recursively on each boundary and then all of the graphs are combined using the **display** procedure form Maple's ***plots*** package.

This approach is slower than first normalizing the ranges and then plotting all of the lines or planes at once, and should probably be changed in the next version of this procedure.

Dependencies

This LLF does not depend on any other part of **PTOLEMY**.

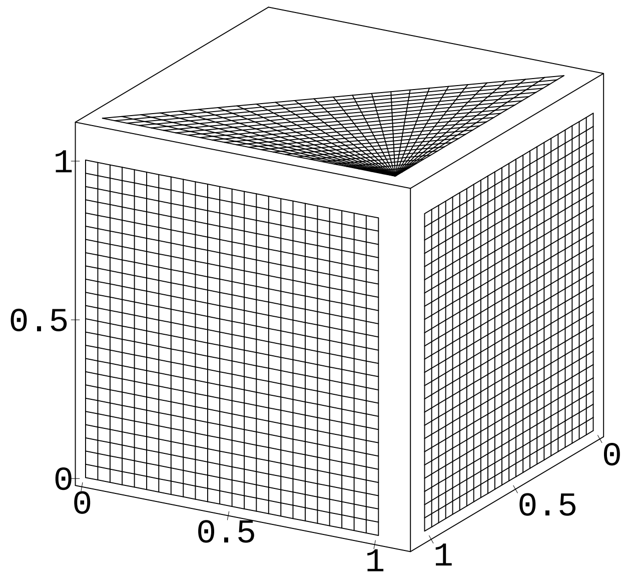

Figure 6.3: Three Boundaries in Three Space

PlotDomBound

This function graphs the boundaries of a collection of domains or subdomains.

PlotDomBound(`Domain:` *collection*({*DomainType, MultiDomainType*}),
`PlotOptions:` *seq*)

The argument `Domain` specifies the domain(s) to be graphed. The argument `Plot-Options` represents a possibly empty sequence of plotting options.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, PlotDomBound):
> D1 := [[x1,x2,x3], 0..1, 0..1, 0..1];
```

$$D1 := [[\,x1, x2, x3\,], 0..1, 0..1, 0..1\,]$$

```
> type(D1,RecDomainType);
```

true

```
> PlotDomBound(D1,
>   orientation=[60,75], axes=framed, tickmarks=[3,3,3], grid=[15,15]);
```

See Figure 6.4.

```
> D2 := [[x,y], [y=0..1, x=0..2-y], [x=1..2, y=2-x..2]];
```

$$D2 := [[x, y], [y = 0..1, x = 0..2 - y], [x = 1..2, y = 2 - x..2]]$$

Maple Worksheet Continued on Next Page

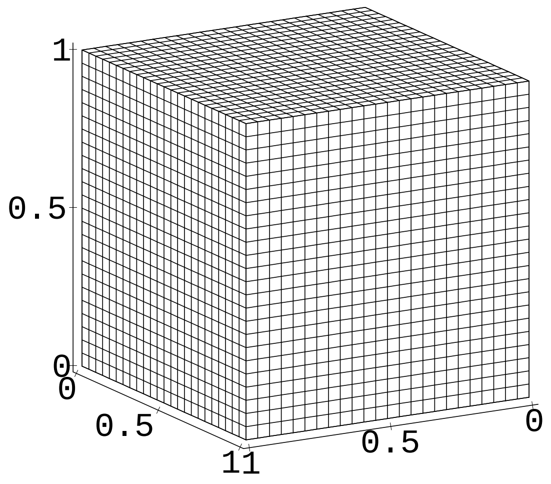

Figure 6.4: The Boundaries of a Unit Cube

```
> type(D2,MultiTradDomainType);
```

$$true$$

```
> PlotDomBound(D2, axes=framed);
```

See Figure 6.5.

```
> D3 := [[x1,x2,x3], x2=0..1, x1=0..2-x2, x3=x1/2..(2-x2)];
```

$$D3 := \left[[\, x1, x2, x3\,], x2 = 0..1, x1 = 0..2 - x2, x3 = \frac{1}{2}\, x1..2 - x2 \right]$$

```
> type(D3,TradDomainType);
```

$$true$$

```
> Options :=
>   axes=framed, orientation=[40,80], tickmarks=[3,2,5], labels=[x1,x2,x3];
```

$$Options := axes = framed, orientation = [\,40, 80\,], tickmarks = [\,3, 2, 5\,], labels = [\, x1, x2, x3\,]$$

```
> PlotDomBound(D3, Options);
```

See Figure 6.6.

```
> PlotDomBound(D3, Options, grid=[5,5], style=wireframe);
```

See Figure 6.7.

```
> Map := (x0,x1) ->
>   (1/2*ln(x0^2 + x1^2) / ln(1/2),
>    2*arctan(x1/x0) - ln(x0^2 + x1^2) / ln(1/2));
```

Maple Worksheet Continued on Next Page

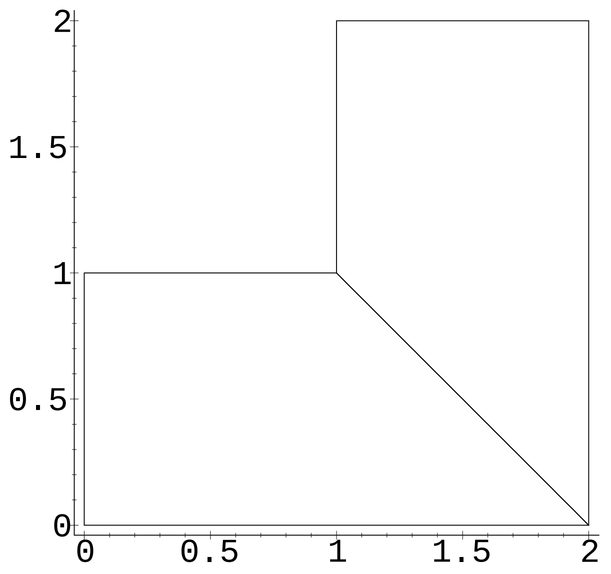

Figure 6.5: The Two Subdomains Commonly Used for the ‘L’-Problem

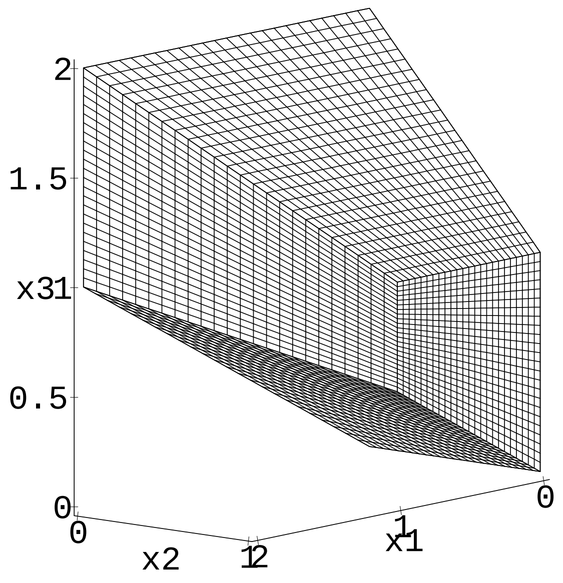

Figure 6.6: An Example Three-Dimensional Traditional Domain

The confusing overlap of the x_3 label and the tick mark label, '1', is unavoidable in the current version of Maple.

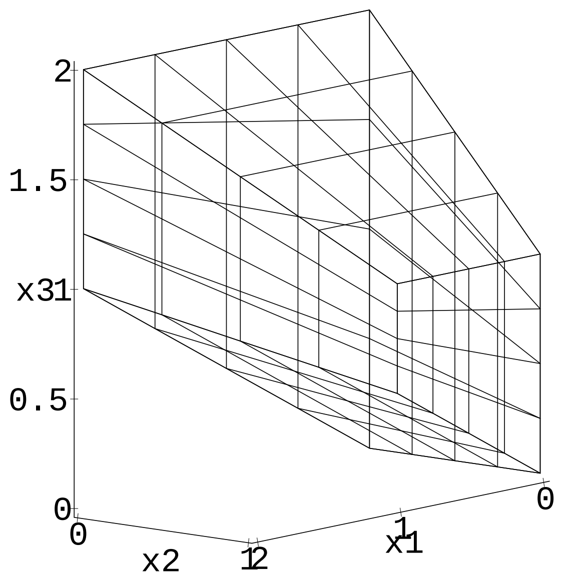

Figure 6.7: Another Representation of Figure 6.6

$$Map := (\,x0, x1\,) \to \left(\frac{1}{2}\,\frac{\ln(\,x0^2 + x1^2\,)}{\ln\left(\frac{1}{2}\right)}, 2\arctan\left(\frac{x1}{x0}\right) - \frac{\ln(\,x0^2 + x1^2\,)}{\ln\left(\frac{1}{2}\right)}\right)$$

```
> InvMap := (y0,y1) -> ((1/2)^y0*cos(y0 + y1/2), (1/2)^y0*sin(y0 + y1/2));
```

$$InvMap := (\,y0, y1\,) \to \left(\left(\frac{1}{2}\right)^{y0}\cos\left(y0 + \frac{1}{2}\,y1\right), \left(\frac{1}{2}\right)^{y0}\sin\left(y0 + \frac{1}{2}\,y1\right)\right)$$

```
> simplify([Map(InvMap(y0,y1))]);
```

$$\left[y0, 2\arctan\left(\frac{\sin\left(y0 + \frac{1}{2}\,y1\right)}{\cos\left(y0 + \frac{1}{2}\,y1\right)}\right) - 2\,y0\right]$$

```
> D4 := [[y0,y1], [-1..1, 0..1], Map, InvMap];
```

$$D4 := [\,[\,y0, y1\,], [\,-1..1, 0..1\,], Map, InvMap\,]$$

```
> type(D4, MappedDomainType);
```

true

```
> PlotDomBound(D4, tickmarks=[3,5]);
```

See Figure 6.8.

End of Maple Worksheet

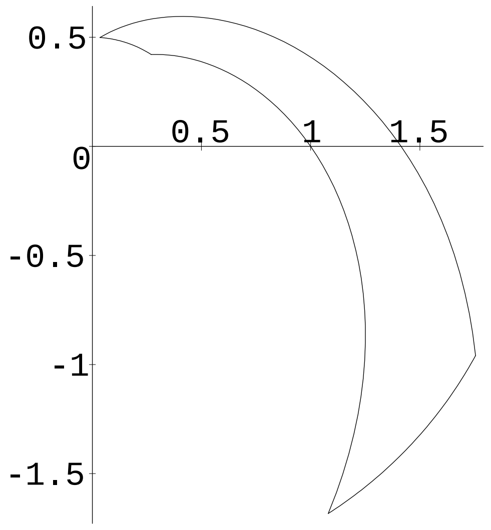

Figure 6.8: The Boundary of a Section of a Spiral

Notice that this domain cannot be represented as a traditional domain type.

Method of Implementation

The procedure calls a helper function **domain_to_bound** on each of the domain specified by the argument `Domain`. This helper procedure returns a set of boundaries for the domain passed to it. The main procedure calls **PlotBound** to construct the actual graph.

The procedure **domain_to_bound** has the interface

domain_to_bound(`Domain:` {*DomainType, MultiDomainType*})

This helper procedure checks the type of the argument `Domain`. If `domain` is a *MultiDomainType* it calls its constructs domain types for each subdomain and calls itself recursively. If `Domain` specifies a single domain then the procedure decides what kind of domain was specified, e.g., *RecDomainType* versus *TradDomainType* and so on, and calls one of the helper procedures **rec_to_bound**, **trad_to_bound**, or **mapped_to_bound** which produces a set of boundaries for this particular kind of domain.

The interface **rec_to_bound**, **trad_to_bound**, and **mapped_to_bound** is:

rec_to_bound(`Domain:` *RecDomainType*)
trad_to_bound(`Domain:` *TradDomainType*)
mapped_to_bound(`Domain:` *MappedDomainType*)

Each of these procedures constructs a table of boundaries where the absolute value of the index is the dimension that is perpendicular to the boundary and the sign of the index indicates rather the boundary is on the low end or the high end.

Dependencies

This procedure calls **PlotBound** in order to plot all of the boundaries.

GridInvMap

This procedure displays the inverse map of a regular grid in the mapped-to domain.

GridInvMap(`InvMap`: *procedure*, `GridSize`: *list(posint)*, `Box`: *list(range)*, `PlotOptions`: *seq*)

The argument `InvMap` defines the inverse map to be visualized. The argument `GridSize` indicates the number of grid point in each dimension of the mapped-to domain. This includes grids on the edges of the domain, so a `GridSize` component of three would indicate one grid on each end and one in the middle. The argument `Domain` indicates the rectangular region in the mapped to domain over which the grid is constructed. The grid is uniform over this region, such that the total number of grid lines or planes in each dimension equals that specified by `GridSize`. Any other arguments are treated as Maple plotting options.

All of the plot options except color specifications are passed to the final graphing routine without change. Color specifications are slightly different than for either `plot` or `plot3d`; the symbol `color` should be equated to either a single color specification (just as is done for `plot` or `plot3d`) or a list of color specifications, one per dimension of the inverse map. If the color specification is a list of colors then the grid lines which are constant in the first dimension will be graphed in the first color and so on.

For one-dimensional spaces the global variable `ptolemy/OneDRange` indicates the vertical extent of the lines used to represent the grid points. During package initialization this global variable is set to `-1/2..1/2`, if it is not already defined. See Section 1.5 for more details.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, GridInvMap);
```

$$[\,GridInvMap\,]$$

```
> InvMap1 := z -> exp(z) / (1 + exp(z));
```

$$InvMap1 := z \to \frac{e^z}{1 + e^z}$$

```
> `ptolemy/OneDRange` := -0.1..0.1;
```

$$ptolemy/\,OneDRange := -.1...1$$

```
> GridInvMap(InvMap1, [16], [-4..4], tickmarks=[6,3]);
```

See Figure 6.9

```
> InvMap2 := (r,theta) -> ((1/2)^r*cos(r + theta/2), (1/2)^r*sin(r + theta/2));
```

$$InvMap2 := (\,r, \theta\,) \to \left(\left(\frac{1}{2}\right)^r \cos\left(r + \frac{1}{2}\,\theta\right), \left(\frac{1}{2}\right)^r \sin\left(r + \frac{1}{2}\,\theta\right)\right)$$

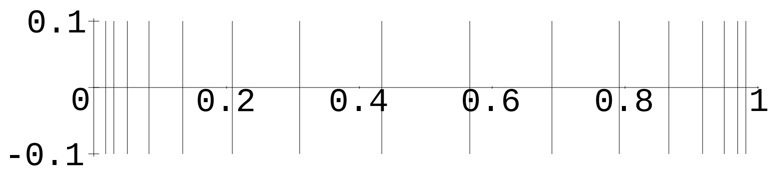

Figure 6.9: The Sinc Points for the Log-Ratio Map from the Interval $[0, 1]$

Maple Worksheet Continued from Previous Page

```
> GridInvMap(InvMap2, [15,5], [0..Pi, 0..Pi/2], tickmarks=[4,7]);
```
See Figure 6.10

```
> InvMap3 := (y1,y2,y3) -> (y1, y2+1/4*(y1-2)*(y1+1)^2, (y1^2 + 1/2)*y3 - y1^2);
```

InvMap3 :=

$$(y1, y2, y3) \to \left(y1, y2 + \frac{1}{4}(y1 - 2)(y1 + 1)^2, \left(y1^2 + \frac{1}{2}\right) y3 - y1^2\right)$$

```
> GridInvMap(InvMap3, [4,3,3], [-1..1,0..1,0..1],
>   axes=framed, tickmarks=[5,5,4]);
```
See Figure 6.11

```
> GridInvMap(InvMap3, [6,3,4], [-1..1,0..1,0..1], orientation=[85,88]);
```
See Figure 6.12

```
> GridInvMap(InvMap3, [6,4,3], [-1..1,0..1,0..1],
>   orientation=[90,15], projection=0.3, axes=boxed, tickmarks=[5,5,2]);
```
See Figure 6.13

End of Maple Worksheet

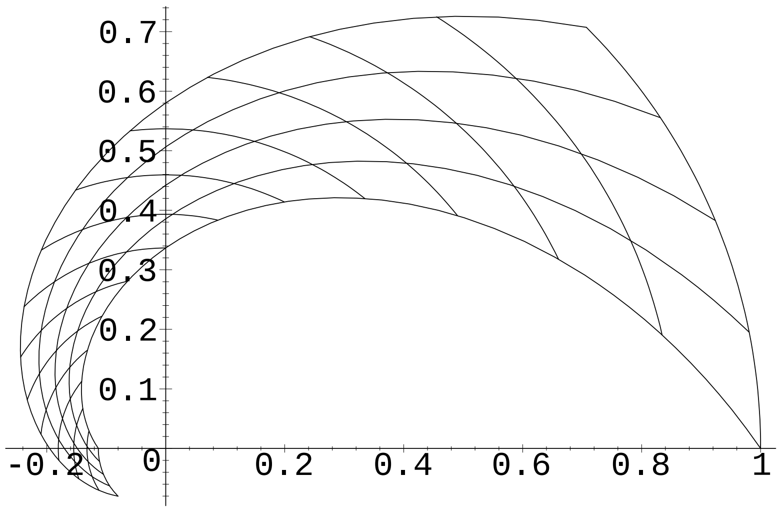

Figure 6.10: The Gridding of the Inverse of a Map from a Sector of a Logarithmic Spiral to a Unit Square

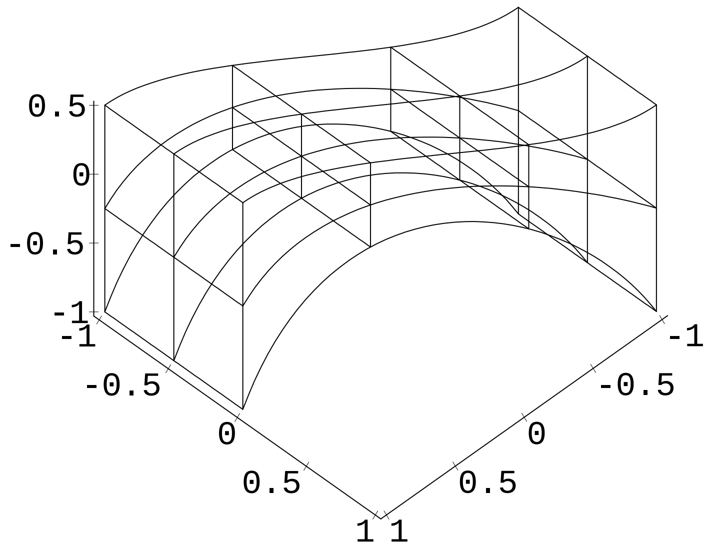

Figure 6.11: The Gridding of the Inverse of a Map from a Stylized Bridge Shaped Region to a Unit Cube

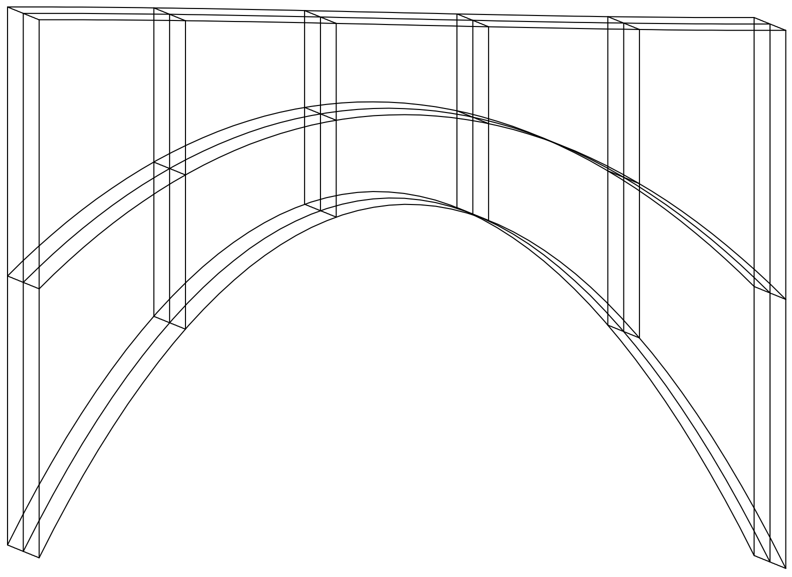

Figure 6.12: A Side View of Figure 6.11

The grid size has increased slightly as compared to Figure 6.11

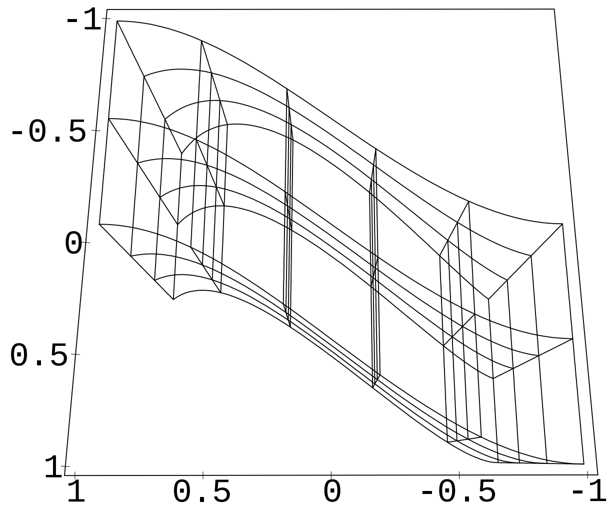

Figure 6.13: A Top View of Figure 6.11

The grid size is slightly different than in either Figure 6.11 or Figure 6.12.

More than Three Dimensions

This module defines graphing routines only for one-, two-, and three-dimensional inverse maps. At this time Maple does not provide much power for visualizing in more than three dimensions. Fortunately, this covers most of the important cases. However, four-dimensional problems do occur often enough, usually as problems with three special dimensions plus time; as a result, it would be useful to be able to handle these higher-dimensional maps.

The procedure **GridInvMap** first determines the dimension of the inverse map, denoted by n. Then it checks for the existence of a globally visible procedure named `grid_inv_Nd`, where the letter `N` is replaced by the number representing the dimension of the inverse map. For example for a two-dimensional map **GridInvMap** looks for a global procedure named **grid_inv_2d**. If this procedure exists **GridInvMap** calls this procedure to create the graphical representation. The result returned by this procedure is in turn returned by **GridInvMap**.

The prescribed interface for these support functions is

grid_inv_Nd(`InvMap`: *procedure*, `GridSize`: *list(posint)*, `Box`: *list(range)*,
`Color`: *list(name))*, `PlotOptions`)

where `InvMap`, `GridSize`, and `Box` are the inputs to **GridInvMap**, `Color` indicates the color to be used for each dimension of the output, and the optional plots options are the subset of any plot options provided to **GridInvMap** which did not specify color information.

The following example illustrates how the user can provide customized support procedures.

Start of Maple Worksheet

```
> with(ptolemy, GridInvMap);
```

$$[\,GridInvMap\,]$$

```
> r1 := t -> cosh(3*(t-1/2));
```

$$r1 := t \to \cosh\left(3\,t - \frac{3}{2}\right)$$

```
> r2 := t -> cosh(2*(t-1/2)) - 3/4;
```

$$r2 := t \to \cosh(\,2\,t - 1\,) - \frac{3}{4}$$

```
> InvMap4 := (r,theta,phi,t) ->
>   ((r*r2(t) + (1-r)*r1(t)) * cos(theta)*sin(phi),
>    (r*r2(t) + (1-r)*r1(t)) * sin(theta)*sin(phi),
>    (r*r2(t) + (1-r)*r1(t)) * cos(phi), t);
```

$$InvMap4 := (\,r, \theta, \phi, t\,) \to ((\,r\,\mathrm{r2}(\,t\,) + (\,1 - r\,)\,\mathrm{r1}(\,t\,)\,)\cos(\,\theta\,)\sin(\,\phi\,),$$
$$(\,r\,\mathrm{r2}(\,t\,) + (\,1 - r\,)\,\mathrm{r1}(\,t\,)\,)\sin(\,\theta\,)\sin(\,\phi\,),$$
$$(\,r\,\mathrm{r2}(\,t\,) + (\,1 - r\,)\,\mathrm{r1}(\,t\,)\,)\cos(\,\phi\,), t)$$

```
> GridInvMap(InvMap4, [3,3,3,10], [0..1, -Pi..Pi, 0..Pi, 0..1]);
Error, (in GridInvMap) GridInvMap can't do 4-dimenional graphs.
```

```
> `ptolemy/grid_inv_4d` := proc
>   (InvMap: procedure, GridSize: [posint,posint,posint,posint],
>    Domain: [range,range,range,range], Color: [name,name,name,name])
>
> local i, LowT, HighT, SubMap, Graph, t;
>   LowT := op(1,Domain[4]);
>   HighT := op(2,Domain[4]);
>
>   for i to GridSize[4] do
>     t := (i - 1) / (GridSize[4] - 1) * (HighT - LowT) + LowT;
>     SubMap := readlib(procmake)(`&proc`([x,y,z],[],[],
>       `&expseq`(op(1..3,[InvMap(x,y,z,t)]))));
>     Graph[i] := `ptolemy/grid_inv_3d`(SubMap, [op(1..3,GridSize)],
>       [op(1..3,Domain)], [op(1..3,Color)], op(5..nargs, [args]))
>   od;
>
>   plots[display]([seq(Graph[i], i=1..GridSize[4])], insequence=true,
>     op(5..nargs,[args]));
> end:
```

```
> GridInvMap(InvMap4, [2,7,6,15], [0..1, -Pi..Pi, 0..Pi, 0..1],
>   orientation=[15,85], axes=boxed);
```

The results can not be effectively typeset. The interested reader can try this example on his own machine.

End of Maple Worksheet

Method of Implementation

The base procedure, **GridInvMap**, first checks the dimension of the inverse map, the grid-size specification, and the domain specification to make sure that they are the same. It then separates any of the color directives from all of the other plot options. If the color option is an array it is saved for passing to the **grid_inv_Nd** procedure otherwise an array with the desired color specifications is constructed. Only the last color directive in the list of plot options is used.

From this it is clear that most of the code for gridding the inverse map is not in the procedure **GridInvMap** but is rather in the support procedures **grid_inv_Nd**. This module provides three standard support procedures for one, two, and three dimensions. These procedures are structurally similar.

Each curve is created by calling the inverse map with all but one of the arguments at fixed numerical values and the one varying argument as a linear expression in a local variable. This linear expression ranges over the range in the corresponding element of `Box`, as the variables of the parameterization range from zero to one. The result is a parameterization, in terms of parameterization variable, of the inverse map of one of the grid point, lines, or plane.

These parameterized curves are stored in a table according to the color assigned to them. Once all of the parameterized lines have been stored in the table, a graph for each color is created and stored in a second table. Finally, the graphs are combined into one graph using the **display** function from the ***plots*** package. Care must be taken to pass the plot options to both procedure for creating the graphs and to the **display** procedure.

Dependencies

This procedure does not depend on any other procedures in the **PTOLEMY** package.

GridMap

This procedure graphs the result of mapping a grid in the original domain to the mapped to domain.

GridMap(`Map`: *procedure*, `N`: *list(posint)*, `Box`: *list(range)*, `PlotOptions`: *seq*)

The argument `Map` describes the map to be visualized. The arguments `N` and `Box` together describe the grid in the original domain. This map will be applied to the grid and the result will be graphed using any plotting options provided by the argument `PlotOptions`. The argument `PlotOptions` is a possibly empty sequence of plotting options.

Just as for **GridInvMap** all of the plotting options except for the color specification are passed directly to the final plotting command (i.e., `plot` or `plot3d`) and color specification can either specify the color for the entire grid or list the colors for each dimension of the grid.

The global variable `ptolemy/OneDRange` specifies the vertical range of the lines used to indicate points when `Map` is defined from a one-dimensional to another one dimensional space. During procedure initialization `ptolemy/OneDRange` is assigned the value `-1/2..1/2` if it is not already assigned a value.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy,GridMap,LogSinhMap);
```

$$[\, GridMap,\, LogSinhMap \,]$$

```
> MapInfo := LogSinhMap(0,LOW,x,z,3/2);
```

$$MapInfo := \left[x \to \ln\left(\sinh\left(\frac{2}{3}\operatorname{arcsinh}(1)\,x\right)\right),\, z \to \frac{3}{2}\,\frac{\operatorname{arcsinh}(e^z)}{\operatorname{arcsinh}(1)},\right.$$
$$\left. x \to \left(\frac{1}{2}\,\frac{\sqrt{6}\sinh\left(\frac{2}{3}\operatorname{arcsinh}(1)\,x\right)}{\sqrt{\operatorname{arcsinh}(1)}},\, \frac{1}{2}\,\frac{\sqrt{6}\operatorname{sech}\left(\frac{2}{3}\operatorname{arcsinh}(1)\,x\right)}{\sqrt{\operatorname{arcsinh}(1)}}\right)\right]$$

```
> Map1 := MapInfo[1];
```

$$Map1 := x \to \ln\left(\sinh\left(\frac{2}{3}\operatorname{arcsinh}(1)\,x\right)\right)$$

```
> `ptolemy/OneDRange` := -0.3..0.3;
```

$$ptolemy/\,OneDRange := -.3...3$$

```
> GridMap(Map1, [16], [0..3], tickmarks=[4,3]);
```

See Figure 6.14.

```
> with(ptolemy,MakeTradMap);
```

$$[\, MakeTradMap \,]$$

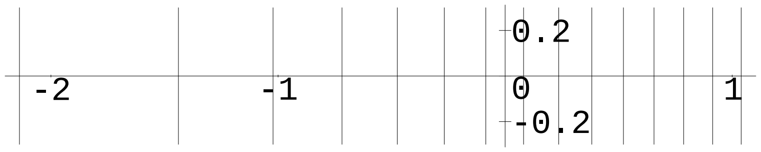

Figure 6.14: A Gridding of the Log-Sinh Map

Notice how the grid becomes more uniform on the right end of the graph. The overlap of tick mark labels between the two dimensions is unavoidable in the current version of Maple.

Maple Worksheet Continued from Previous Page

```
> DomainA := [[x1,x2], x2=0..1, x1=0..2-x2];
```

$$DomainA := [[x1, x2], x2 = 0..1, x1 = 0..2 - x2]$$

```
> MakeTradMap(DomainA, 'Map2A', 'Trash');
> eval(Map2A);
```

$$(x1, x2) \to \left(-\frac{x1}{-2 + x2}, x2\right)$$

```
> GridMap(Map2A, [9,5], [0..2,0..1], 0..1,0..1, axes=none);
```

See Figure 6.15.

Maple Worksheet Continued on Next Page

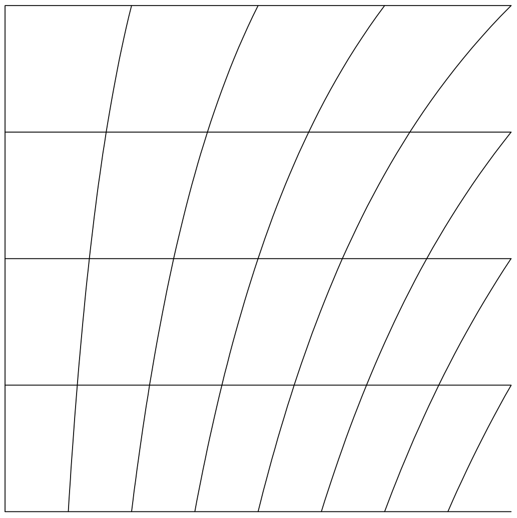

Figure 6.15: The Map of the Gridding of the Lower-Left Subdomain in the 'L'-Problem

Assuming that the coordinates in the original domain are denoted by X_1 and X_2, the horizontal lines are the iso-X_2 lines and the iso-X_1 lines are quadratic, curving towards the right when traced from top to bottom.

```
> DomainB := [[x1,x2], x1=1..2, x2=2-x1..2];
```

$$DomainB := [\,[\,x1\,, x2\,], x1 = 1..2, x2 = 2 - x1..2\,]$$

```
> MakeTradMap(DomainB, 'Map2B', 'Trash');
> eval(Map2B);
```

$$(\,x1\,, x2\,) \rightarrow \left(x1 - 1, \frac{x2 - 2 + x1}{x1}\right)$$

```
> GridMap(Map2B, [5,9], [1..2,0..2], 0..1,0..1, axes=none);
```

See Figure 6.16.

End of Maple Worksheet

More then Three Dimensions

This module only defines graphing procedures for one-, two-, and three-dimensional maps. Just as with **GridInvMap** the user can add graphing procedures for higher dimensional spaces or redefine the standard graphing procedures. The graphing prescribed interface is

ptolemy/grid_Nd(`Map:` *procedure*, `GridSize:` *list(posint)*, `Box:` *list(range)*, `Color:` *list(name))*, `PlotOptions`)

where `Map`, `GridSize`, and `Box` are the inputs to **GridInvMap**, `Color` indicates the

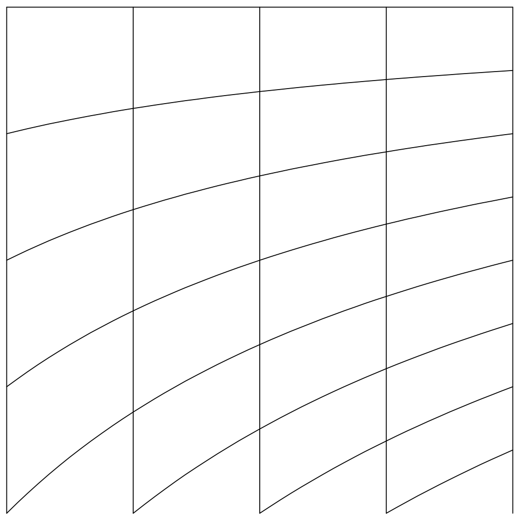

Figure 6.16: The Map of the Gridding of the Upper-Right Subdomain in the 'L'-Problem

Assuming that the coordinates in the original domain are denoted X_1 and X_2, the vertical lines are the iso-X_1 lines and the iso-X_2 lines are quadratic, curving down when traced from right to left.

color to be used for each dimension of the output, and `PlotOptions` is a possibly empty sequence of plotting options. The argument `PlotOptions` should not contain any color plot options.

Method of Implementation

The core procedure, **GridMap**, first checks the dimension of the map, the grid-size specification, and the domain specification to make sure that they are all the same. It then separates any of the color directives from all of the other plot options. If the color option is an array it is saved to be used as an argument for the **grid_Nd** procedure; otherwise an array with the desired color specifications is constructed. Only the last color directive in the list of plot options is used. Then the name of the plotting procedure is constructed, i.e., with the appropriate value of n; if a global variable with this name has been assigned a procedure then this procedure is called, otherwise an error message is issued.

From this description it is clear that most of the code in this LLF is in the support procedures, **grid_Nd**. This module provides three standard support procedures for one, two, & three dimensions. These procedures are structurally similar; they produce a description of the mapping of each grid line. Each description of a curve in the final graph stored in a table according the the specified graphing color. Once all of the curves have been stored in the table a graph is created with all of the like colored curves and stored in a second table. Finally, the different graphs are combined into one graph using the **display** function from the ***plots*** package. Care must be taken to pass necessary plot options to both the procedure for creating the graphs and to the **display** command.

The description of the map of each grid line is created by evaluating the map with all but one of the arguments specified to be a specific numerical values and the one nonnumeric argument as a linear expression which varies over the range in the corresponding element of `Box`, as the parameterization variable ranges from zero to one. The result is a parameterization of the map of one of the grid lines.

Dependencies

This procedure does not depend on any other part of the **PTOLEMY** system.

ClipPlot

This procedure will clip any two-dimensional Maple plot to some oriented box.

ClipPlot(`Plot:` *PLOT*, `Box:` *[range(numeric)*, `range(numeric)])`

The argument `Plot` specifies the plot to be clipped and the argument `Box` specifies the clipping box. The result returned is a new plot object in which the data have been modified so that all of the curves lie within the clipping box.

Maple's *PLOT* record has an optional `VIEW` field which can be used to specify clipping to be applied during the display of a plot. This field is set when the plotting range is limited by the appropriate arguments to either the `plot` command or the `display` command from the ***plots*** package. This mechanism is actually preferable to using the **ClipPlot** procedure for most interactive uses, but it does not work when joining clipped forms of two different plots to form a new plot and the clipping box is different for each plot. This need arises often when using **GridMap** to simultaneously visualize the maps from several different subdomains. It is primarily for this application that **ClipPlot** was created.

Finally, notice that Maple uses a different type to represent three-dimensional plots, called *PLOT3D*. This procedure will not work on objects of the three-dimensional type.

Example Usage

The first example illustrates the problem that motivates the need for this procedure as well as illustrates a simple application of the procedure.

Start of Maple Worksheet

```
> with(plots,display);
```

$$[\,display\,]$$

```
> with(ptolemy, MakeTradMap,GridMap,ClipPlot);
```

$$[\,ClipPlot, GridMap, MakeTradMap\,]$$

```
> Domain1 := [[x1,x2], x2=0..1, x1=0..2-x2];
```

$$Domain1 := [[\,x1, x2\,], x2 = 0..1, x1 = 0..2 - x2\,]$$

```
> MakeTradMap(Domain1, 'Map1', 'Trash');
> eval(Map1);
```

$$(\,x1, x2\,) \to \left(-\frac{x1}{-2 + x2}, x2\right)$$

```
> Domain2 := [[x1,x2], x1=1..2, x2=2-x1..2];
```

$$Domain2 := [[\,x1, x2\,], x1 = 1..2, x2 = 2 - x1..2\,]$$

```
> MakeTradMap(Domain2, 'Map2', 'Trash');
> eval(Map2);
```

$$(\,x1, x2\,) \to \left(x1 - 1, \frac{x2 - 2 + x1}{x1}\right)$$

```
> A := GridMap(Map1, [11,5], [0..2,0..1], 0..1,0..1):
> Move := (y1,y2) -> (y2 + 1, 1 - y1);
```

$$Move := (\,y1, y2\,) \to (\,y2 + 1, 1 - y1\,)$$

```
> Map2A := subs(RESULT = Move(Map2(x1,x2)), (x1,x2) -> RESULT);
```

$$Map2A := (\,x1, x2\,) \to \left(\frac{x2 - 2 + x1}{x1} + 1, 2 - x1\right)$$

```
> B := GridMap(Map2A, [6,9], [1..2,0..2], 1..2, 0..1):
> Bound := plot([1,t,t=0..1], thickness=3, linestyle=4):
> plots[display]({A,B,Bound}, axes=none);
```

See Figure 6.17.

```
> AA := ClipPlot(A, [0..1,0..1]):
> BB := ClipPlot(B, [1..2,0..1]):
> plots[display]({AA,BB,Bound}, axes=none, scaling=constrained);
```

See Figure 6.18.

End of Maple Worksheet

The complexity of the previous example is limited by the fact that all of the points to be removed are on the same end of each curve. When the same curve crosses the boundary of the clipping box more than once then that curve must be divided into multiple curves. The following example illustrated that this procedure can handle these cases as well.

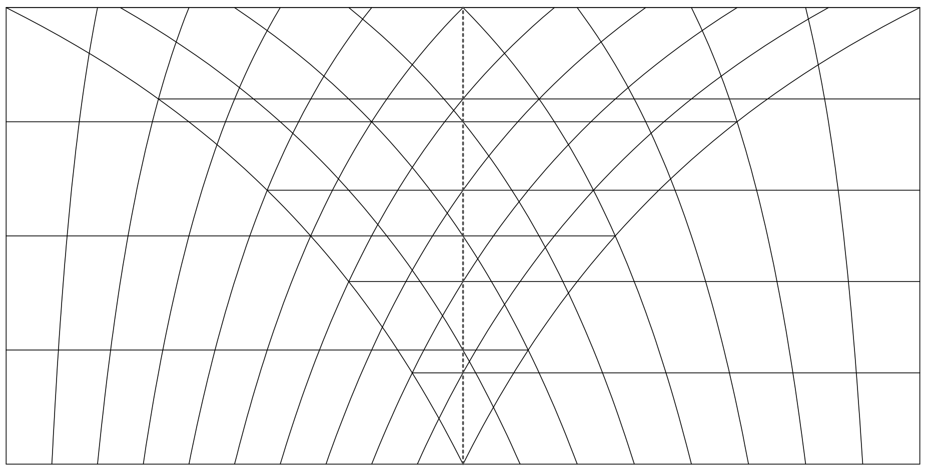

Figure 6.17: An Example of the Problem of Joining Two Graphs with Different `View` Fields

The thick vertical dashed line indicates the line along which both parts of the final plot should have been clipped.

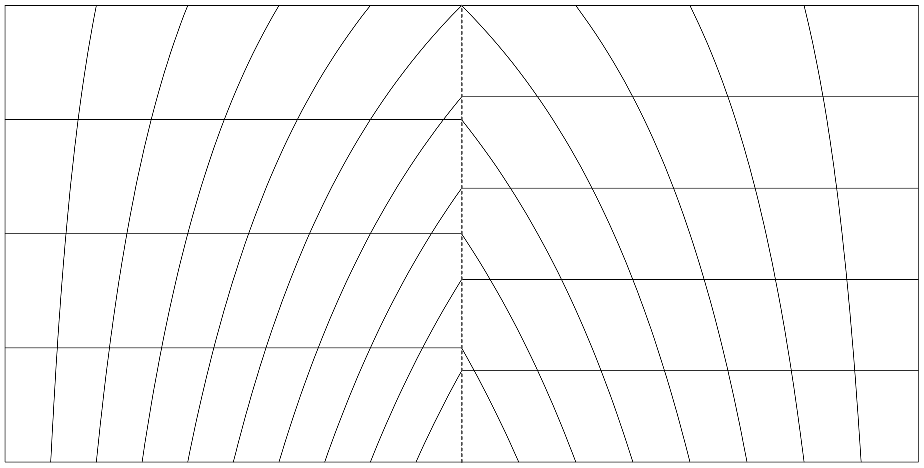

Figure 6.18: The Result of Figure 6.17 Using **ClipPlot** in Place of the `VIEW` Field

Maple Worksheet Continued from Previous Page

Start of Maple Worksheet

```
> with(plots,display);
```

$$[\,display\,]$$

```
> with(ptolemy, PlotDomBound,MakeTradMap,GridMap,ClipPlot);
```

$$[\,ClipPlot, GridMap, MakeTradMap, PlotDomBound\,]$$

```
> Domain := [[x1,x2], x1=0..1, x2=0..1 + 2*x1*(1-x1)*sin(Pi*5*x1)];
```

$$Domain := [[\,x1, x2\,], x1 = 0..1, x2 = 0..1 + 2\,x1\,(\,1 - x1\,)\sin(\,5\,\pi\,x1\,)\,]$$

```
> DomPlot := PlotDomBound(Domain, thickness=3):
> GridPlot := GridMap((x1,x2) -> (x1,x2), [11,16], [0..1,0..1.5]):
> display({GridPlot,DomPlot}, axes=framed, tickmarks=[3,4]);
```

See Figure 6.19.

```
> MakeTradMap(Domain,'Map','Trash');
> eval(Map);
```

$$(\,x1, x2\,) \to \left(x1, -\frac{x2}{-1 - 2\,x1\,\sin(\,5\,\pi\,x1\,) + 2\,x1^2\sin(\,5\,\pi\,x1\,)}\right)$$

```
> MapOfGrid := GridMap(Map, [11,16], [0..1,0..1.5]):
> Top := plot([t,1,t=0..1], linestyle=4, thickness=3):
> display({MapOfGrid, Top}, axes=framed, tickmarks=[3,6]);
```

See Figure 6.20.

```
> MapOfGridInDom := ClipPlot(MapOfGrid, [0..1,0..1]):
> display({MapOfGridInDom, Top}, axes=framed, tickmarks=[3,3]);
```

See Figure 6.21.

End of Maple Worksheet

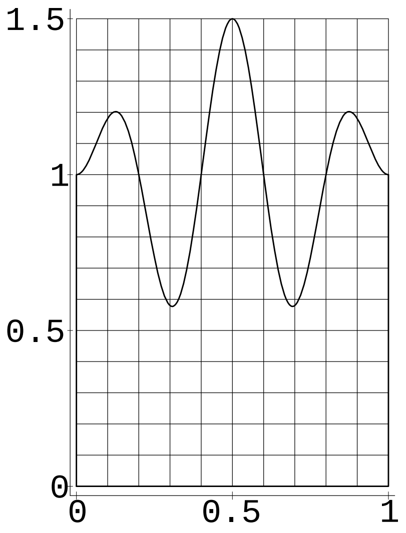

Figure 6.19: A Traditional Domain with a Nonmonotonic Boundary

Method of Implementation

Roughly half of the code in this LLF is devoted to extracting the curves from the *PLOT* record and inserting the modified curves back into the *PLOT* record. The actual clipping is done by adding a new point that lies on the boundary between any pair of points that straddle a boundary and then removing all of the points that lie outside the boundary.

The *PLOT* record modified by recursively applying two local procedures, **PlotOp** and **CurveOp**, with the `map` command. The procedure **PlotOp** checks to see if the field passed to it as an argument is a `CURVES` field; if it is, the procedure will apply **CurveOp** to each of the the curve's subfield using the `map` command. The **CurveOp** procedure then checks to see if the subfield it received is a list of points; if this is the case the procedure traverses the list looking for transitions from inside the clipping box to outside the clipping box or vice versa. At every transition it calls the support procedure **interp_to_bound** to estimate the point on the boundary that should be added to the curve. Each portion of the curve that is inside the clipping box is added to a table. When the curve has been traversed **CurveOp** returns the list of curves that will replace the old curve in the *PLOT* record.

The interface for the procedure **interp_to_bound** is

interp_to_bound(`In`: *[numeric*, `numeric]`, `Out`: *[numeric*, `numeric]`,
`Box`: *[range(numeric)*, `range(numeric)])`

The argument `In` specifies the point that is inside the boundary, the argument `Out` specifies the point that is outside the boundary, and the argument `Box` specifies the clipping box. The procedure will return the point on the line segment connecting `In` and `Out` which is on the boundary. This is equivalent to filling in the plot with linear interpolation near the boundary.

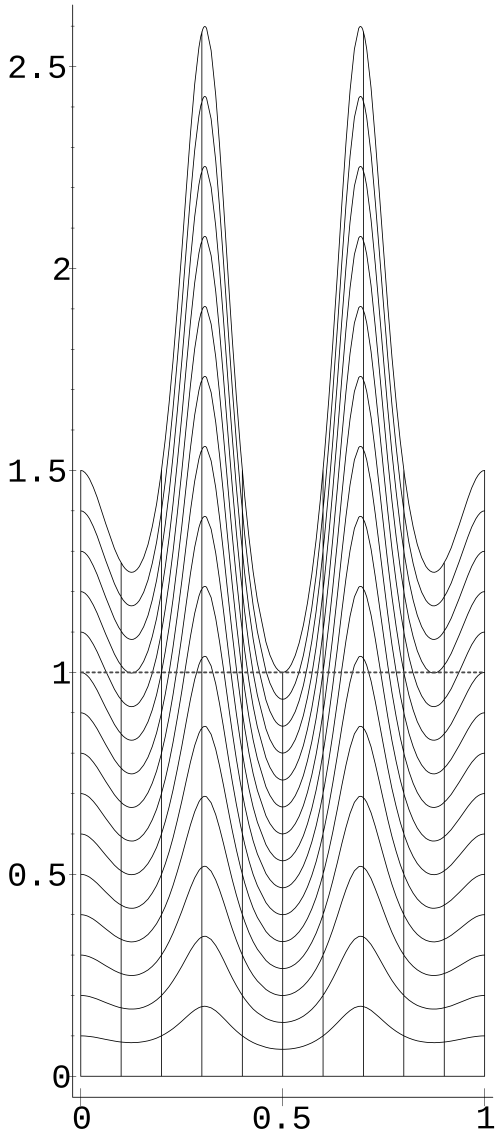

Figure 6.20: The Mapping of the Grid Shown in Figure 6.19

The thick horizontal dashed line is the boundary of the domain. Denoting the coordinates of the original domain as X_1 and X_2 it is clear from the symbolic form of the map that the iso-X_1 lines are vertical in this graph, but a powerful optical illusion makes it appear otherwise. Using visually orthoginal colors for the iso-X_1 and the iso-X_2 line dramatically reduces the effect of this illusion.

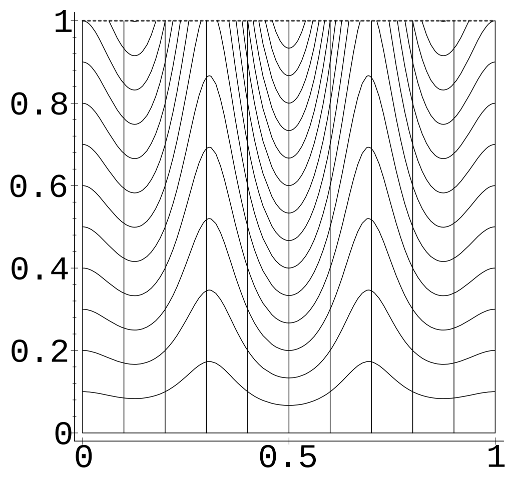

Figure 6.21: The Mapping Onto the Unit Square of a Gridding of the Domain in Figure 6.19

It is not strictly necessary for `In` to be inside the box or for `Out` to be outside the box; either may lie on the boundary of the box. However, the procedure will fail if `In` is outside the box or if `Out` is inside the box.

Dependencies

This procedure does not depend on any other part of the **PTOLEMY** package.

sinc

This LLF makes `sinc` one of the functions recognized by Maple. It accomplishes this by defining a function named **sinc** that handles automatic simplifications and defining the functions needed to make **evalf**, **diff**, and **int** correctly operate on expressions containing the `sinc` function.

Integration and numerical evaluation of the sinc function are both defined. However, at this time, the result of differentiated a `sinc(x)` contains a removable singularity at $x = 0$. That is, the symbolic result is incorrectly defined at $x = 0$. This problem should be fixed in the next version.

Definition

The sinc function is defined as

$$\operatorname{sinc}(x) := \sin(\pi x)/(\pi x)$$

In some signal processing literature the function $\sin(x)/x$ is referred to as the sinc function. This less common definition of "the sinc function" is equivalent to $\operatorname{sinc}(x/\pi)$ using the definition used here.

Example

Start of Maple Worksheet

```
> with(ptolemy, sinc);
```

$$[\mathit{sinc}]$$

```
> sinc(1/2);
```

$$2\,\frac{1}{\pi}$$

```
> sinc(2/3);
```

$$\frac{3}{4}\,\frac{\sqrt{3}}{\pi}$$

```
> sinc(2/7);
```

$$\operatorname{sinc}\left(\frac{2}{7}\right)$$

```
> sinc(2./7.);
```

$$.8710264157$$

```
> evalf(sinc(1E-9), 20);
```

$$.99999999999999999836$$

```
> temp := int(sinc(x), x=0..1);
```

$$\mathit{temp} := \frac{\operatorname{Si}(\pi)}{\pi}$$

```
> evalf(temp);
```

$$.5894898722$$

```
> int(sinc(y), y=-infinity..x);
```

$$\frac{\mathrm{Si}(\pi x)}{\pi}+\frac{1}{2}$$

```
> temp := diff(sinc(x), x);
```

$$temp := \frac{\cos(\pi x)}{x}-\frac{\sin(\pi x)}{\pi x^2}$$

```
> unapply(temp, x)(0);
Error, (in unknown) division by zero
```

```
> limit(temp, x=0);
```

$$0$$

End of Maple Worksheet

Method of Implementation

The LLF defines four global functions, i.e., **sinc**, **int/sinc**, **diff/sinc**, and **evalf/sinc**. The **sinc** procedure performs all automatic simplifications on expression involving the sinc functions. The **int/sinc** and **diff/sinc** procedures instruct Maple's `diff` and `int` commands on how to differentiate and integrate expressions involving the sinc function. The **evalf/sinc** computes performs numerical evaluation of the sinc function for numerical arguments.

The automatic simplifications pose a philosophical dilemma, specifically which simplifications should be performed automatically. Many Maple developers, including the author, believe that Maple performs more automatic simplifications then it should, but when adding a new function to the system it is also important that the function behave in expressions similar to other already defined functions. In this end **sinc** will evaluate exactly to one or zero for all integer arguments and it will invoke **evalf/sinc** for all floating point arguments. If the argument denoted by `Arg` is rational and `sin(Pi*Arg)` automatically simplifies to some other expression, then **sinc** will return this simplified expression divided by `Pi*Arg`. In all other cases the **sinc** returns an inert expression of the form which caused its invocation. That is, the **sinc** function performs no other automatic simplifications.

The **int/sinc** and **diff/sinc** functions simply return the result of integration or differentiation using `sin(Pi*Arg) / (Pi*Arg)` in place of the sinc function. As mentioned in the introduction to this subsection, this is always correct for integration, but not quite correct for differentiation. In particular, differentiation results in a removable singularity at the origin.

The **evalf/sinc** procedure first checks to see that only one argument was specified. Denote this argument by `Arg`. Maple's `evalf` command will call **evalf/sinc** without first applying `evalf` to the argument, so **evalf/sinc** must first apply `evalf` to the argument. The result may or may not be numerical; if the result is not numerical then the result returned by the **evalf/sinc** procedure should be an inert sinc function applied to the partially evaluated argument. Otherwise the procedure should compute a numerical result.

In this latter case the procedure next determines how close the argument is to the origin. If the argument is close enough to the origin then the procedure returns the integer 1. If it is not quite this close but still very close it uses the 1 term Taylor series expansion about the origin to compute the results. Similarly, for arguments a little further from the origin it uses the two term Taylor series expansion about the origin. If the argument is further than this from the origin then the function uses the $\mathrm{sinc}(x) = \sin(\pi\, x)/(\pi\, x)$ formula to compute the result.

The intent is for the result to have a relative error of less then plus or minimum `10^(-Digits)` for any argument and any value of `Digits`. The error is defined as the error with respect to the exact answer when the argument equals the floating point value `evalf(Arg,Digits)`.

Dependencies

This LLF does not depend on any other part of the **PTOLEMY** package.

Chapter 7

Low Level

This chapter describes the LLFs that exist primarily to provide support for other parts of **PTOLEMY**. These LLFs may be divided into five groups, 1) general utilities, 2) utilities for order manipulation, 3) utilities to support mapping, 4) utilities to support collocation, and 5) utilities to support writing to files.

The general utilities include: The LLF ***proc_dimen***, which contains two procedures for determining the dimensionality of the domain and range of a procedure; the LLF ***map_info_ops***, which includes three procedures for constructing parts of a *MapInfoType* given a definition of the map; the procedure **cross_prod** for constructing cross products of arbitrary sets; and the LLF ***comb_ops***, which constraints two procedures for constructing combinations of elements of a single set.

The utilities for order manipulation include: The LLF ***order_ops*** which contains 12 procedures for expanding, simplifying, constructing the difference of, constructing the minimum or maximum of, and converting between multidomain and single domain representations of structures involving *OrderType*s; the procedure **pde_order** which computes the order of a PDE; and the procedure **required_order** which determines the order of approximation required to accurately evaluate the boundary constraints.

The utilities to support mapping include: The procedure **free_var** which determines a superset of the state-variables a PDE; the procedure **get_D_forms** which extracts terms involving D-operators from an expression; the procedure **isDop** which decides if a term is a D-operator and if it is to what variable it is applied; the procedure **pde_apply** which converts PDEs from an unapplied form to an applied form, and the procedure **pde_unapply** which performs the reverse conversion; the procedure **pde_collect** which converts PDEs to a normal form where the differential expression is expanded and grouped with respect to differential operators but the coefficients are simplified; the procedure **fast_map** which can quickly map a PDE; and the procedure **unit_normal** which computes an expression for the vectors which are of length one and is normal to the boundary of a subdomain.

The utilities to support collocation include: The procedure **spline** which will construct a spline with prescribed boundary values (including derivative values) in the original domain and with exponential decay once the spline is been mapped; the procedure **make_bases** which constructs bases suitable for a prescribed order of approximation; the procedure **get_SB_var** which extracts all differential terms involving `S`-variables or `B`-variables from a PDE; the procedure **collocate_main** which performs collocation on a governing equation, and the procedure **collocate_bound** which performs the collocation of boundary constraints. The procedure **assign_bound** which assigns boundary constraints to specific collocation points; and the LLF ***b_ops*** which contains four procedures for manipulating expressions involving `B`-variables.

The utilities to support writing to files include: The LLF **kron_ops** which contains six procedure for converting expressions from `SB`-notation to Kronecker product notation; and the procedure **to_string** which converts a Maple expression to a string in the format used by the `lprint` command.

proc_dimen

This LLF provides two functions for computing the dimensionality of a procedure.

range_dimen(`F`: *procedure*)
domain_dimen(`F`: *procedure*)

The procedure **domain_dimen** checks the procedure specified by the argument `F` to see how many formal parameters it expects. The function **range_dimen** checks attempts to invoke the procedure specified by `F` with its formal parameters used as actual parameters and then report the length of the sequence returned. If this invocation of **F** results in an error then **range_dimen** returns the symbol `UNKNOWN`.

Example

Start of Maple Worksheet

```
> readlib(`ptolemy/proc_dimen`):
> Map1 := (theta,phi) -> [cos(theta)*cos(phi), sin(theta)*cos(phi),sin(phi)];
```

$$Map1 := (\theta,\phi) \to [\cos(\theta)\cos(\phi), \sin(\theta)\cos(\phi), \sin(\phi)]$$

```
> `ptolemy/range_dimen`(Map1);
```

$$2$$

```
> `ptolemy/domain_dimen`(Map1);
```

$$3$$

```
> Map2 := proc(X)
> local i,Result;
>   Result := X;
>   for i to 4 do Result := Result - (Result^2 - X) / (2*Result) od;
> end;

Map2 := proc(X)
        local i,Result;
            Result := X;
            for i to 4 do  Result := Result-1/2*(Result^2-X)/Result od
        end
```

```
> Map2(3);
```

$$\frac{18817}{10864}$$

```
> evalf("); sqrt(3.);
```

$$1.732050810$$

$$1.732050808$$

```
> `ptolemy/range_dimen`(Map2);
```

$$1$$

```
> `ptolemy/domain_dimen`(Map2);
                                        1
```

```
> Map3 := x -> if (x < 0) then -x else x fi;

Map3 := proc(x) options operator,arrow; if x < 0 then -x else x fi end

> `ptolemy/range_dimen`(Map3);
                                        1

> `ptolemy/domain_dimen`(Map3);
                                   UNKNOWN
```

End of Maple Worksheet

The Unknowable

If calling the procedure **F** with its formal arguments causes **F** to enter an "infinite loop" the procedure **domain_dimen** will never return. No amount of additional logic can solve this problem, since it is a form of the halting problem.

The more common problem is that for many procedures type checking (or other restrictions) does not allow the procedure to be invoked with its formal arguments, even though the dimensionality of the range would be obvious from a casual inspection of the procedure. Since the internals of the procedure are visible in Maple, additional logic might resolve many of these problems.

However, the current implementation is sufficient for procedures that are "arrow operators." This is the common case for procedures that define maps and inverse maps. In the future if more complex mapping functions become common, **range_dimen** may become inadequate. However, if this happened significant effort to improve this functions would be misguided. It would be better to redesign the procedures which currently depend on **range_dimen** to require that the user explicitly specific the dimensionality of some procedures.

Dependencies

This LLF does not depend on any other part of the **PTOLEMY** system. In addition the **check_mapped_domain** in the LLF ***type*** is the only part of the **PTOLEMY** system that uses this LLF.

map_info_ops

This LLF provides procedures for manipulating *MapInfoType*s.

weight(`MapInfo`: *MapInfoType*, `Arg`: *algebraic*, `SubOrder`: *SubOrderType*)
weight(`MapInfo`: *MapInfoType*, `Arg`: *algebraic*, `SubOrder`: *SubOrderType*,
 `OrderUsed`: *name*)

mapped_weight(`MapInfo`: *MapInfoType*, `Coord`: *name*, `SubOrder`: *SubOrderType*)
mapped_weight(`MapInfo`: *MapInfoType*, `Coord`: *name*, `SubOrder`: *SubOrderType*,
 `OrderUsed`: *name*)

make_map_info(`Coord`: *name*, `NewCoord`: *name*, `Map`: *algebraic*)
make_map_info(`Coord`: *name*, `NewCoord`: *name*, `Map`: *algebraic*, `Options`)

The procedure **weight** returns the weight function evaluated at `Arg` corresponding to the map and raised to the power specified by `SubOrder`. If the `SubOrder` specifies a different order for the low end than for the high end, but the factorization of the weight is not known, then the weight will be raised to the maximum of the two requested requested order. If present the optional argument `OrderUsed` specifies the name of the variable to be assigned a *SubOrderType* indicating the power to which the weight was actually raised.

The procedure **mapped_weight** returns a result equivalent to **weight** except mapped to the mapped-to domain. The simplification procedure specified by the global variable **ptolemy/SimpProc** is applied to this result.

The procedure **make_map_info** constructs a *MapInfoType* from partial information about the map. Two options are currently supported:

inv_map: If one of the optional argument is of the form `inv_map=expression` then the expression will be used as an explicit statement of the map inverse.

weight: If one of the optional arguments is of the form `weight=expression` then the expression will be used as the weight. If the weight has been factored into a two parts one corresponding to the low end and one corresponding to the high end this information can be specified with an optional argument of the form `weight=(low,high)` where `low` and `high` are the factors corresponding to the low and high end of the interval, respectively.

If no options are given the procedure will attempt to use **solve** and **diff** in order to construct the information not provided. In this case the procedure specified by the global variable `ptolemy/SimpProc` will be applied to both of the results. If **make_map_info** cannot construct the inverse map it will return an error.

If the inverse map or the weight are explicitly specified no effort is made to double check that the specified expressions are correct. Checking might prevent some errors but when Maple is unable to find the inverse or to adequately simplify the weight it may also be unable to confirm that the user provided expressions are correct. Thus the error checking mechanism could become a new source of errors.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, LogRatioMap);
```

$$[\mathit{LogRatioMap}]$$

```
> readlib(`ptolemy/map_info_ops`):
> Map1 := LogRatioMap(-1,1,x,z);
```

$$\mathit{Map1} := \left[x \to \ln\left(\frac{x+1}{1-x}\right), z \to \frac{e^z - 1}{1 + e^z}, x \to \left(\frac{1}{2}(x+1)\sqrt{2}, \frac{1}{2}(1-x)\sqrt{2}\right)\right]$$

```
> T1 := `ptolemy/weight`(Map1,exp(z)/(1+exp(z)), 1);
```

$$\mathit{T1} := \frac{1}{2}\left(\frac{e^z}{1+e^z} + 1\right)\left(1 - \frac{e^z}{1+e^z}\right)$$

```
> factor(T1);
```

$$\frac{1}{2}\,\frac{2\,e^z + 1}{(1+e^z)^2}$$

```
> T2 := `ptolemy/weight`(Map1, exp(z)/(1+exp(z)), [2,0], 'OrderUsed');
```

$$\mathit{T2} := \frac{1}{2}\left(\frac{e^z}{1+e^z} + 1\right)^2$$

```
> OrderUsed;
```

$$[2, 0]$$

```
> factor(T2);
```

$$\frac{1}{2}\,\frac{(2\,e^z + 1)^2}{(1+e^z)^2}$$

```
> `ptolemy/weight`(Map1, exp(z)/(1+exp(z)), [0,0], 'OrderUsed');
```

$$1$$

```
> phi := x -> ln(tan(Pi/2*x));
```

$$\phi := x \to \ln\left(\tan\left(\frac{1}{2}\pi x\right)\right)$$

```
> Map2 := `ptolemy/make_map_info`(x,z,phi(x));
```

$$\mathit{Map2} := \left[x \to \ln\left(\tan\left(\frac{1}{2}\pi x\right)\right), z \to 2\,\frac{\arctan(e^z)}{\pi}, x \to 2\,\frac{\tan\left(\frac{1}{2}\pi x\right)}{\left(1 + \tan\left(\frac{1}{2}\pi x\right)^2\right)\pi}\right]$$

```
> T1 := `ptolemy/weight`(Map2,exp(z)/(1+exp(z)), 1);
```

$$\mathit{T1} := 2\,\frac{\tan\left(\frac{1}{2}\,\frac{\pi e^z}{1+e^z}\right)}{\left(1 + \tan\left(\frac{1}{2}\,\frac{\pi e^z}{1+e^z}\right)^2\right)\pi}$$

```
> `ptolemy/weight`(Map2,exp(z)/(1+exp(z)), [2,0], 'OrderUsed');
```

$$4\,\frac{\tan\left(\frac{1}{2}\,\frac{\pi\,e^z}{1+e^z}\right)^2}{\left(1+\tan\left(\frac{1}{2}\,\frac{\pi\,e^z}{1+e^z}\right)^2\right)^2\,\pi^2}$$

```
> OrderUsed;
```

$$2$$

```
> `ptolemy/make_map_info`(x1,z1, ln(x1/(1-x1)));
```

$$\left[x1 \to \ln\left(\frac{x1}{1-x1}\right),\, z1 \to \frac{e^{z1}}{1+e^{z1}},\, x1 \to -x1\,(\,-1+x1\,)\right]$$

```
> `ptolemy/make_map_info`(x,z, ln((x - a) / (b - x)));
```

$$\left[x \to \ln\left(\frac{x-a}{b-x}\right),\, z \to -\frac{e^z\,b+a}{-e^z-1},\, x \to \frac{(\,-x+a\,)\,(\,b-x\,)}{-b+a}\right]$$

End of Maple Worksheet

Dependencies

This LLF does not depend on any other part of the **Ptolemy** system. However, the procedures **SToLinKron** and **make_bases** both depend on procedures in this LLF.

cross_prod

This procedure forms a "set" that is the cross-product of a "set" of discrete "sets." Let each of these discrete "sets" be called an *input set.* The words sets is quoted because these sets are mathematical sets, but are not necessarily Maple sets. Each of the input sets and the "set" of input sets may be either ordered or unordered. An ordered mathematical set is represented using a Maple list.

cross_prod(`InputSets`: *Collection(collection(anything)*)
cross_prod(`InputSets`: *Collection(collection(anything)*, `NameProc`: *procedure*,
`ExtraArgs`: *seq*)

If the optional argument `NameProc` is not specified the result will be of type *Collection(collections)*, where each inner collection contains one element from each of the input sets. Collectively the elements of each inner element specify one element in the output set. Often these elements of the result are used to form some more natural name, via concatenation, multiplication, or some similar process. If the optional argument `NameProc` is specified it will be invoked on each element in the output set. If present the argument `ExtraArgs` will be passed to **NameProc**.

Order of the Results

The result may be an ordered set, i.e., a Maple list, if some part of the argument `InputSets` is ordered. If the "set" of input sets is ordered then this indicates that the order in which the dimensions are traversed is fixed by `InputSets`; if any one of the input sets is ordered then this indicates that the order in which this particular dimension is traversed is to be fixed by `InputSets`. In either case the result will be a Maple list, not a Maple set, so that the order of the points may be fixed.

In addition, if the order of the dimensions is fixed, this same order will be used for the construction of the elements of the result. This means that either the result will be of type `list(list)` or that the result passed to **NameProc** will be of type list, depending on whether or not any optional arguments are specified.

Examples

Start of Maple Worksheet

```
> with(ptolemy,cross_prod);
```

$$[\,cross\_prod\,]$$

```
> cross_prod({});
```

$$\{\,\}$$

```
> cross_prod([]);
```

$$\{\,\}$$

```
> cross_prod([a,b,c]);
```

$$[[a, b, c]]$$

```
> cross_prod([a,[b,B],c]);
```

$$[[a, b, c], [a, B, c]]$$

```
> cross_prod({{1,2,3}, {a,b,c}});
```

$$\{\{1, a\}, \{1, b\}, \{1, c\}, \{2, a\}, \{2, b\}, \{2, c\}, \{3, a\}, \{3, b\}, \{3, c\}\}$$

```
> cross_prod({{c,b,a}, {2,3,1}});
```

$$\{\{1, a\}, \{1, b\}, \{1, c\}, \{2, a\}, \{2, b\}, \{2, c\}, \{3, a\}, \{3, b\}, \{3, c\}\}$$

```
> cross_prod([[c,b,a], [2,3,1]]);
```

$$[[c, 2], [c, 3], [c, 1], [b, 2], [b, 3], [b, 1], [a, 2], [a, 3], [a, 1]]$$

```
> cross_prod({[c,b,a], [2,3,1]});
```

$$[\{2, c\}, \{3, c\}, \{1, c\}, \{2, b\}, \{3, b\}, \{1, b\}, \{2, a\}, \{3, a\}, \{1, a\}]$$

```
> cross_prod({[c,b,a], {2,3,1}});
```

$$[\{1, c\}, \{1, b\}, \{1, a\}, \{2, c\}, \{2, b\}, \{2, a\}, \{3, c\}, \{3, b\}, \{3, a\}]$$

```
> cross_prod({{c,b,a}, [2,3,1]});
```

$$[\{2, a\}, \{3, a\}, \{1, a\}, \{2, b\}, \{3, b\}, \{1, b\}, \{2, c\}, \{3, c\}, \{1, c\}]$$

```
> cross_prod([{c,b,a}, {2,3,1}]);
```

$$[[a, 1], [a, 2], [a, 3], [b, 1], [b, 2], [b, 3], [c, 1], [c, 2], [c, 3]]$$

```
> cross_prod([[2,3,1], {c,b,a}]);
```

$$[[2, a], [2, b], [2, c], [3, a], [3, b], [3, c], [1, a], [1, b], [1, c]]$$

```
> cross_prod([{2,3,1}, [c,b,a]]);
```

$$[[1, c], [1, b], [1, a], [2, c], [2, b], [2, a], [3, c], [3, b], [3, a]]$$

```
> cross_prod([[A,B,C], [1,2,3]], x -> cat(op(x)));
```

$$[A1, A2, A3, B1, B2, B3, C1, C2, C3]$$

```
> cross_prod([[1,2,3], [1,2,3]], (x,y) -> cat(y,op(x)), V);
```

$$[V11, V12, V13, V21, V22, V23, V31, V32, V33]$$

End of Maple Worksheet

Method of Implementation

The procedure first checks for optional arguments. Then it checks to see if the input mathematical set of mathematical sets is the empty set. If it is not the empty set then it places the sets into a table indexed by the set number. This process is necessary because any of the input sets that contain only a single element may be represented by the element in isolation; such instances are converted into a list before adding them to the table. At the same time the size of each input set and the total number of results is computed. Also the input is checked to determine if dimensions need to be ordered. The dimensions need to be ordered if `InputSets` is of type ***list***.

Next an index into the cross-product set is constructed to point to the first element. Then a "for loop" constructs the result for the current position into a table and increments the position. The position is incremented so that the last dimension varies most rapidly.

Finally, the table of results is either converted to a set or to a list, i.e., in order, depending on whether or not the result needs to be ordered. The result needs to be ordered if either the dimensions are ordered of if the set for any specific dimension is ordered.

Dependences

The procedures in this LLF do not depend on any other part of the **PTOLEMY** library. However, the procedures **assign_bound**, **collocate_bound**, and **collocate_main** all depend on procedures from this LLF.

comb_ops

This LLF contains procedures for computing the collection of all combinations of elements from a collection of elements.

comb_fixed_size(`Data`: *Collection(anything)*, `Size`: *posint*)
comb_fixed_size(`Data`: *Collection(anything)*, `Size`: *posint*, `NameProc`: *procedure*, `ExtraArgs`)

comb_all(`Data`: *collection(anything)*)
comb_all(`Data`: *collection(anything)*, `NameProc`: *procedure*, `ExtraArgs`)

The procedure **comb_fixed_size** will return all of the combinations with the number of elements specified by the argument `Size`. The procedure **comb_all** will return all of the combinations of any number of elements.

In both cases, if the optional arguments are not specified, the result will be of type *Collection(Collection)*. Each inner collection is one of the combinations. Sometimes it is desirable to think of the result as a set of some other type, then a set of combinations. In such cases it is often necessary to convert each combination into some more natural name. The optional argument `NameProc` provides a simple way of doing this. When present the optional argument `NameProc` is applied to each combination. If any extra arguments are present they are also passed to the **NameProc**.

The Order of the Combinations

If the augment `Data` is a list both procedures will represent the order of the elements within the set. This means that the elements within each combination will be reported in the same order as they appear in `Data` and that the combinations will ordered in an order determined by the order of the elements within `Data`.

Defining the ordering of the combination (of fixed size) requires first introducing an ordering on the permutations (of fixed size). This procedure orders the parameters using a cartesian ordering. The permutations of a fixed size can be thought of as the indices of the array with dimensionality equal to size of the permutation where each dimension varies over the elements of `Data`. For example the permutations of size two of the set $\{A, B, C\}$ are illustrated in Figure 7.1. This procedure adopts the convention that the right most index varies most rapidly.

The set of fixed size ordered combinations is a subset of the fixed size permutations, specifically the subset of elements $(x_1, x_2, \ldots, x_n)$ where $x_1 < x_2$, $x_2 < x_3$, and $x_i < x_{i+1}$ for i up to $n-1$. In two dimensions this subset is rather easy to visualize; it is merely the upper triangle of the two-dimensional array (excluding the diagonal) as illustrated in Figure 7.2. The ordering of elements of a multidimensional array has a well-established convention which is also illustrated in Figure 7.1. This procedure orders this subset of the permutations in the same order as the permutations. This is also illustrated in Figure 7.2.

For combinations of size greater than two the ordering is still defined as equivalent to the ordering of the permutations of the same size but visualizing the relevant subsets becomes much harder. Each constraint of the form $x_i < x_{i+1}$ defines a half-space and the ordered combinations occurs at the intersections of all these half spaces. Figure 7.3

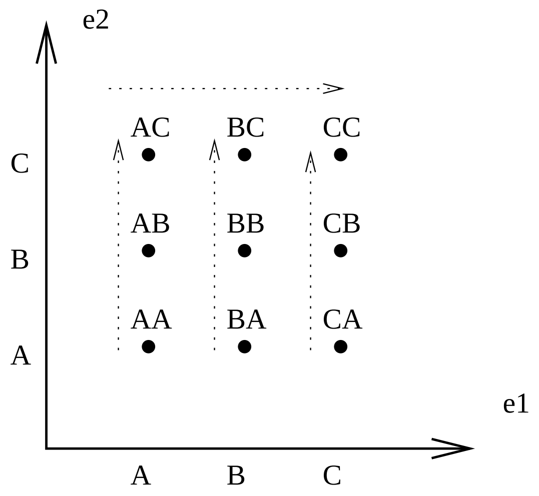

Figure 7.1: Permutations of Size Two Viewed as a Two-Dimensional Array

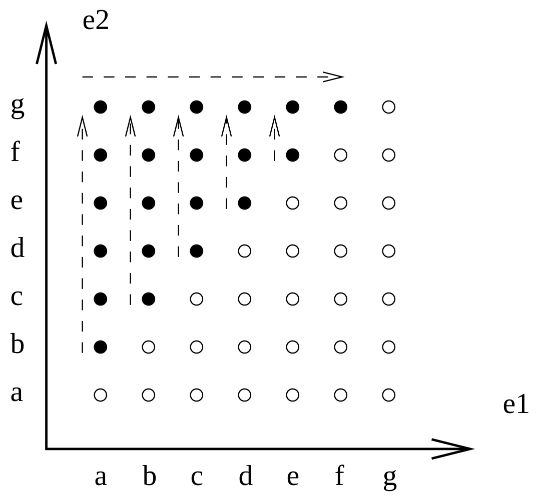

Figure 7.2: Combinations of Size Two Viewed as a Two-Dimensional Array

illustrates this for combinations of size three.

Examples

Start of Maple Worksheet

```
> ptolemy[init]();
> readlib(`ptolemy/comb_ops`):
```

```
> Set := {Red, Green, Blue};
```

$$Set := \{Red, Green, Blue\}$$

```
> List := [Red,Green,Blue];
```

$$List := [Red, Green, Blue]$$

```
> `ptolemy/comb_fixed_size`(Set, 1);
```

$$\{\{Red\}, \{Green\}, \{Blue\}\}$$

```
> `ptolemy/comb_fixed_size`(Set, 2);
```

$$\{\{Red, Green\}, \{Red, Blue\}, \{Green, Blue\}\}$$

```
> `ptolemy/comb_fixed_size`(Set, 3);
```

$$\{\{Red, Green, Blue\}\}$$

```
> `ptolemy/comb_all`(Set);
```

$$\{\{Red, Green, Blue\}, \{Red\}, \{Green\}, \{Blue\}, \{Red, Green\},$$
$$\{Red, Blue\}, \{Green, Blue\}\}$$

```
> `ptolemy/comb_fixed_size`(List, 2);
```

$$[[Red, Green], [Red, Blue], [Green, Blue]]$$

```
> `ptolemy/comb_all`(List);
```

$$[[Red], [Green], [Blue], [Red, Green], [Red, Blue], [Green, Blue],$$
$$[Red, Green, Blue]]$$

```
> `ptolemy/comb_all`(Set, x -> cat(op(x)));
```

$$\{RedGreenBlue, RedGreen, Red, Green, Blue, RedBlue, GreenBlue\}$$

```
> `ptolemy/comb_all`(List, x -> cat(op(x)));
```

$$[Red, Green, Blue, RedGreen, RedBlue, GreenBlue, RedGreenBlue]$$

End of Maple Worksheet

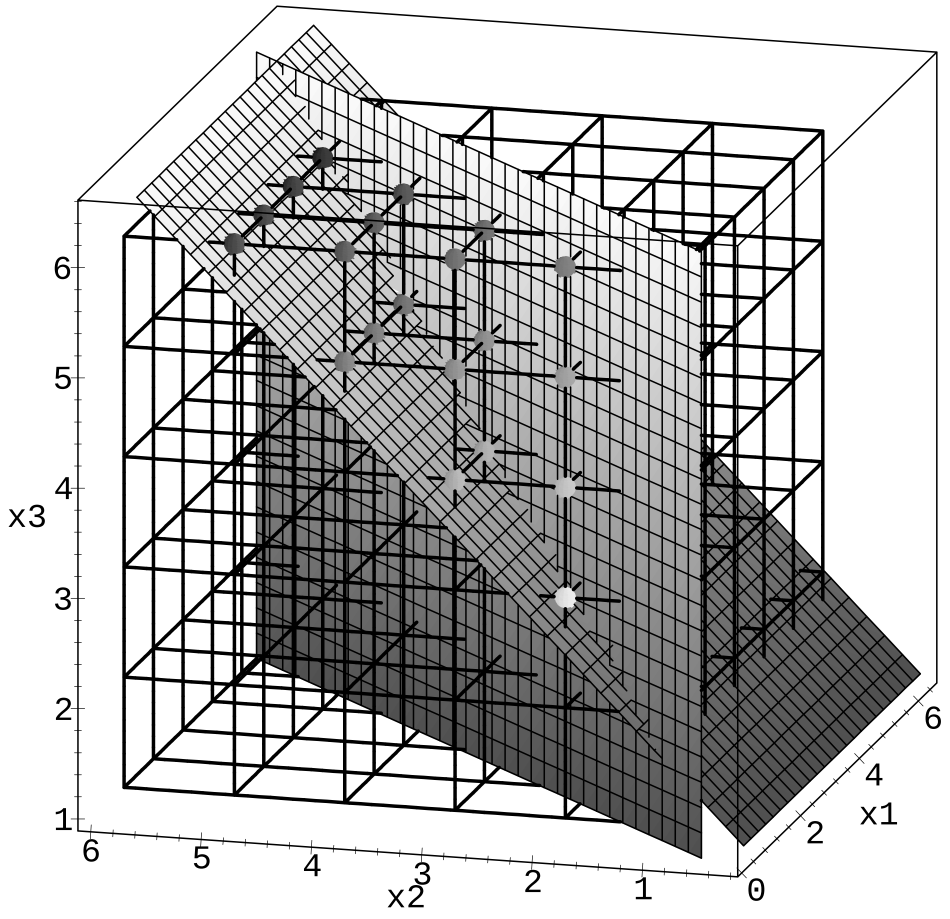

Figure 7.3: Combinations of Size Three Viewed as a Three-Dimensional Array

The elements of this three-dimensional array are illustrated by the edges of the cubic grid (i.e. the thicker lines). The permeations occur at the intersections of the lines in this grid. The plane dividing the top and bottom faces of the region represents the constraint $x1 < x2$, and the plane dividing the front and back faces of the region represent the constraint $x2 < x3$. Only the region intersected by the two half spaces contains the subset of permutations which are the ordered combinations. Each of these combinations is highlighted by gray spheres surrounding the node point.

Method of Implementation

The procedure **comb_all** simply calls the procedure **comb_fixed_size** for sizes ranging from one to the number of elements in `Data`.

The procedure **comb_fixed_size** first computes the number of combinations of the specified size using the standard formula

$$\frac{m!}{n!(m-n)!}$$

where m is the total number of elements in `Data` and n is size of each combination. Then it "counts" these combinations out in the order described in the subsection titled "The Order of the Combinations" on on page 211. If the final result is to be ordered, lists are used to preserve this order; otherwise sets are used that generally do not preserve this order.

The method of counting out the combinations is to use an index which is initialized to $(1, 2, \ldots, n)$. This index acts as a collection of cursor[1] into `Data` (i.e., referencing the elements of `Data`) specifying which elements appear in the current combination. On each iteration of the loop the index is incremented such that all of the ordering constraints are satisfied. This is done by:

1. Searching from the last element of the index that is not already at the upper limit of its range. For all but the last element of the index the upper limit of its range is defined by the value of the element following it.
2. Incrementing this element of the index by one.
3. Reseting all of the elements in the index that follow this element to the lower limits of their ranges. For all but the first element of the index the lower limit is defined by the value of the element preceding it.

To make checking of the upper range of the last element of the index easier the index is actually extended by one element. This extra element is assigned the value $n+1$ element. This element of the index is only used to check the range on the n^{th} component. This creates problems if an attempt is made to increment the index past the last value in the set of combinations to be used. To avoid this problem the loop is rearranged so that the index is incremented before it is used. This also reduces the number of times the index is incremented by one, but this optimization is performed to simplify the logic for incrementing the index not to reduce the number of loop iterations.

Dependences

This module does not depend on any other part of the **Ptolemy** system.

[1]Here the word cursor is used in the data structures sense to indicate the pointer like use of an index into an array. One of the more obvious uses of cursors, keeping track of the current location on an array oriented display, has gradually replaces this usage of cursor as the most common usage in computer science.

order_ops

This module provides several functions for manipulating *SubOrderTypes*, *OrderTypes* and *OrderSpectTypes.*

sub_order_expand(`Sub`: *SubOrderType*)
sub_order_simplify(`Sub`: *SubOrderType*)

sub_order_subtract(`Plus`: *SubOrderType*, `Minus`: *SubOrderType*)
sub_order_min(`Sub1`: *SubOrderType*, `Sub2`: *SubOrderType*)
sub_order_max(`Sub1`: *SubOrderType*, `Sub2`: *SubOrderType*)

order_expand(`Ord`: {*OrderType, OrderSpecType*})
order_simplify(`Ord`: {*OrderType, OrderSpecType*})

order_subtract(`Plus`: *OrderType*, `Minus`: *OrderType*)
order_max(`Ord1`: *OrderType*, `Ord2`: *OrderType*)
order_min(`Ord1`: *OrderType*, `Ord2`: *OrderType*)

spec_list_to_multi(`List`: *collection(list(OrderSpecType))*)
multi_spec_to_list(`Multi`: *collection(MultiOrderSpecType)*)

The procedures **sub_order_expand** and **sub_order_simplify** check their arguments to determine whether it is an integer or a list of two integers and return the alternate form when needed.

The procedures **sub_order_subtract**, **sub_order_min** and **sub_order_max** first expand both of their arguments, then perform subtraction, max, or min operation between corresponding locations in an expression tree, and finally simplify the result.

The procedure **order_expand** and **order_simplify** either apply the procedure **sub_order_expand** or **sub_order_simplify** to each of the *SubOrderType* components of an argument of type *OrderType* or call themselves with the *OrderType* component of an argument of type *OrderSpecType.*

The procedures **order_subtract**, **order_min**, and **order_max** each call **sub_order_subtract**, **sub_order_min**, or **sub_order_max** to pair of corresponding *SubOrderType*'s in its arguments.

Finally, the functions **spec_list_to_multi** and **multi_spec_to_list** convert between a collection of *MultiOrderSpecType*'s and a list of *OrderSpecType*'s, i.e., one element of the list per domain.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy,init); readlib(`ptolemy/order_ops`):
                                        [init]

> `ptolemy/sub_order_expand`([1,2]);
                                        [1, 2]

> `ptolemy/sub_order_expand`(0);
                                        [0, 0]
```

```
> `ptolemy/sub_order_simplify`([1,2]);
                                        [1, 2]

> `ptolemy/sub_order_simplify`([1,1]);
                                          1

> `ptolemy/sub_order_simplify`(2);
                                          2
```

```
> `ptolemy/sub_order_subtract`([1,2], 1);
                                        [0, 1]

> `ptolemy/sub_order_subtract`(2, [0,2]);
                                        [2, 0]

> `ptolemy/sub_order_subtract`([2,1], [1,0]);
                                          1
```

```
> `ptolemy/order_expand`([[1,2], 1]);
                                    [[1, 2], [1, 1]]

> `ptolemy/order_simplify`([[1,2], [1,1], 2]);
                                     [[1, 2], 1, 2]
```

```
> `ptolemy/order_subtract`([[1,2], [2,3], 1, 2], [1,[1,2], 1,[0,1]]);
                                  [[0, 1], 1, 0, [2, 1]]

> `ptolemy/order_max`([[1,2], [1,3], 1], [[2,0], [2,1], 0]);
                                     [2, [2, 3], 1]

> `ptolemy/order_max`([[1,2], [0,2], [2,1]], [2,1,1]);
                                   [2, [1, 2], [2, 1]]
```

```
> `ptolemy/order_min`([[0,2], [1,3], 1], [[1,0], [2,0], 2]);
```
$$[0,[1,0],1]$$

```
> `ptolemy/order_min`([[1,2], [0,2], [2,1]], [2,1,1]);
```
$$[[1,2],[0,1],1]$$

```
> Multi1 := [V, [[1,[0,1]], [[1,2], 1]]];
```
$$\mathit{Multi1} := [V,[[1,[0,1]],[[1,2],1]]]$$

```
> List1 := `ptolemy/multi_spec_to_list`(Multi1);
```
$$\mathit{List1} := [[V,[1,[0,1]]],[V,[[1,2],1]]]$$

```
> `ptolemy/spec_list_to_multi`(List1);
```
$$[V,[[1,[0,1]],[[1,2],1]]]$$

```
> Multi2 := {[U, [[1,0], [1,2], [1,1]]], [V, [[2,1], [0,1], [0,2]]]};
```
$$\mathit{Multi2} := \{[U,[[1,0],[1,2],[1,1]]],[V,[[2,1],[0,1],[0,2]]]\}$$

```
> List2 := `ptolemy/multi_spec_to_list`(Multi2);
```
$$\mathit{List2} := [\{[V,[1,0]],[U,[2,1]]\},\{[V,[1,2]],[U,[0,1]]\},$$
$$\{[V,[1,1]],[U,[0,2]]\}]$$

```
> `ptolemy/spec_list_to_multi`(List2);
```
$$\{[V,[[1,0],[1,2],[1,1]]],[U,[[2,1],[0,1],[0,2]]]\}$$

```
> List3 := `ptolemy/multi_spec_to_list`(convert(Multi2,list));
```
$$\mathit{List3} := [[[U,[1,0]],[V,[2,1]]],[[U,[1,2]],[V,[0,1]]],$$
$$[[U,[1,1]],[V,[0,2]]]]$$

```
> `ptolemy/spec_list_to_multi`(List3);
```
$$[[U,[[1,0],[1,2],[1,1]]],[V,[[2,1],[0,1],[0,2]]]]$$

End of Maple Worksheet

pde_order

This module contains several functions to assist in determining the order of differentiation in various differential forms. The primary function is:

pde_order!(`Eq`: {*algebraic, equation, list(equation), set(equation)*}, `Var`: *list(name)*,
`Dimen`: *posint*)

This function expects a differential form, a list of state-variables, and the dimensionality of the problem. The function returns a list of ordered pairs, one per state-variable; the first element in each ordered pair is the state-variable name and the second is the maximum order of differentiation per dimension. `Eq` may be either an applied or an unapplied form.

In addition this module provides three other functions for computing the differential order of various other differential forms.

'ptolemy/diff_list'(`Term`: *algebraic*)
'ptolemy/diff_order'(`Term`: *algebraic*, `Dimen`: *posint*)
'ptolemy/de_order'(`Eq`: {*algebraic, equation, list(equation), set(equation)*},
`Var`: *name*)

The function **diff_list** accepts a single partial (or regular) differential term, and returns a list indicating with respect to which dimensions differentiation is performed. The input to **diff_list** may be either applied or unapplied form, but must use the `D`-notation. Note the result is exactly the same list as used by Maple's indexed `D`-notation.

The function **diff_order** will accept the same inputs as **diff_list** but returns an ordered list indicating the order of differentiation applied at each dimension. In order to know the size of this list, the caller must specify the dimension of the problem.

The function **de_order** will accept a (nonpartial) differential form and a single state-variable name. It will return the maximum order of differentiation with respect to this variable. The differential form may be an expression, an equation, or a collection of equations.

Example Usage

```
> with(ptolemy, pde_order);
```

$$[\mathit{pde\_order}]$$

```
> 'ptolemy/diff_list'(V);
```

$$[\,]$$

```
> 'ptolemy/diff_list'(D(V));
```

$$[1]$$

```
> 'ptolemy/diff_list'(D[2,2](V)(x,y));
```

$$[2,2]$$

```
> `ptolemy/diff_list`((D@@3)(V)(x));
```

$$[1,1,1]$$

```
> `ptolemy/diff_list`((D[1,2]@@2)(V)(x,y));
```

$$[1,2,1,2]$$

```
> `ptolemy/diff_order`(D(V), 1);
```

$$[1]$$

```
> `ptolemy/diff_order`(D[2,2](V)(x,y,z), 3);
```

$$[0,2,0]$$

```
> Sys := D(V) + (D@@2)(U) = K;
```

$$Sys := \mathrm{D}(V) + D^{(2)}(U) = K$$

```
> `ptolemy/de_order`(Sys, V);
```

$$1$$

```
> `ptolemy/de_order`(Sys, U);
```

$$2$$

```
> `ptolemy/de_order`(Sys, K);
```

$$0$$

```
> `ptolemy/de_order`(Sys, P);
```

$$-1$$

These are the two-dimentional incompressable flow, time varying, Navior Stokes Equations. The coordinates are x,y,t, i.e., two spatial dimensions and time. So including time the problem is actually three-dimentional.

```
> Sys := {
>   D[3](V) + U*D[1](U) + V*D[2](V) = -D[1](P)/rho + mu*(D[1,1](U) + D[2,2](U)),
>   D[3](U) + V*D[1](V) + V*D[2](V) = -D[2](P)/rho + mu*(D[1,1](V) + D[2,2](V)),
>   D[1](U) + D[2](V) = 0};
```

$$Sys := \Big\{ D_1(U) + D_2(V) = 0,$$

$$D_3(V) + U\,D_1(U) + V\,D_2(V) = -\frac{D_1(P)}{\rho} + \mu\,(D_{1,1}(U) + D_{2,2}(U)),$$

$$D_3(U) + V\,D_1(V) + V\,D_2(V) = -\frac{D_2(P)}{\rho} + \mu\,(D_{1,1}(V) + D_{2,2}(V))\Big\}$$

```
> pde_order(Sys, [U,V,W,P], 3);
```

$$[[U,[2,2,1]],[V,[2,2,1]],[\mathrm{W},[-1,-1,-1]],[P,[1,1,0]]]$$

Method of Implementation

The procedure **diff_list** largely checks the type of `Term` and parses for the indexing information. If the `D`-operator is an unindexed form, indicating that the differential term is with respect to a single variable, the result `[1]` is returned. If `Term` is an `@@`-operator, **diff_list** is applied to `D`-operator portion of the `@@`-operator and the result is repeated the appropriate number of times. If `Term` is any other kind of function the result is obtained by recursively applying **diff_list** to the function name, i.e., the zeroth operand. In all other cases `Term` is assumed to not be a differential form and `[]` is returned.

The procedure **diff_order** calls **diff_list**, sums the results into a zero-initialized table, and finally constructs a list from this table. Since the state-variables are not known by **diff_order** it is always assumed that a state-variable is contained in `Term`, so the result is always nonnegative.

The procedure **de_order** calls **get_D_forms** to extract each of the differential forms and then applies **diff_list** to each of theses terms. It uses the list size returned by **diff_list** as the order of differentiation, and returns the maximum order for any term returned by **get_D_forms**. If the specified state-variable does not appear in the specified differential form, the value `-1` is returned. Returning `-1` allows the caller to distinguishes the case when the state-variable does not appear in the expresion from the case when the state-varable does appear but is never differentiated (in which case zero is returned). This same convention is used for **pde_order**.

The procedure **pde_order** preforms the following steps: 1) calls **get_D_forms** to extract each of the differential forms, 2) uses **isDop** to extract the state-variable corresponding to each term returned by **get_D_forms**, 3) calls **diff_order** to figure out the order of differentiation occurring in this term, 4) uses a local table to keep track of the maximum order of differentiation, and 5) finally returns the information in this table in a list of lists format.

Design Deficiencies

No check is made to see if the specified dimension makes sense for the particular differential form.

Specifically **diff_order** should report an error if the specified dimension is greater than 1, but the `Term` is an unindexed `D`-operator (which is valid syntax only for single variable problems). Similarly, **diff_order** should report an error if differentiation is performed with respect to a dimension greater than the specified dimensionality of the problem.

Perhaps **de_order** should allow either a single state-variable or a list of state-variables. And perhaps it is unnecessary for **pde_order** to have state-variable names in its results. An ordered list of lists where each entry is the order information corresponding to the state-variable of the corresponding position is the specified list of state-variables.

External Dependencies

de_order uses **get_D_forms** to extract all of the differential forms and evaluate them as single terms.

pde_order uses **get_D_forms** for the same purpose as **de_order**, but also uses **isDop** to extract the state-variable name. By handling only one state-variable at a time **de_order** avoids the need to use **isDop**.

required_order

This procedure figures the minimum order of approximation required in order to both satisfy the minimum requested order and to be able to accurately approximate specified expressions or equations at specified boundaries.

ptolemy/required_order(`OrderSpec`: *collection(OrderSpecType)*,
`Bound`: *collection(BoundFormType)*)

The result will be a list of *OrderSpecType*'s.

Example

Start of Maple Worksheet

```
> with(ptolemy,required_order);
```

$$[\mathit{required\_order}]$$

```
> required_order([V, [1,0]], [1,LOW, D[2](V), TagA]);
```

$$[[V,[1,1]]]$$

```
> required_order([V, [1,0]], [2,LOW, D[2](V) = 0]);
```

$$[[V,[1,[1,0]]]]$$

```
> required_order([V, [1,0]], {[1,LOW, D[1,1](V) = 0], [2,LOW, D[2,2](V) = 0]});
```

$$[[V,[[2,1],[2,0]]]]$$

```
> required_order({[U, [0,0]], [V, [1,0]]}, [2,LOW, D[1](V), TagB]);
```

$$[[V,[1,0]],[U,[0,0]]]$$

```
> required_order({[U, [0,0]], [V, [1,0]]},
>   [[2,LOW, D[1](V) = D[2](U)], [1,HIGH, D[1,2](U), TagC]]);
```

$$[[V,[1,0]],[U,[[0,1],1]]]$$

```
> required_order({[U, [0,0]], [V, [1,0]]},
>   {[1,LOW, D[1,1](V) + D[1](U) = 0], [1,HIGH, D[2](U) = V],
>    [2,LOW, D[2](V) = U], [2,HIGH, U, TagD]});
```

$$[[V,[[2,1],[1,0]]],[U,[[1,0],1]]]$$

End of Maple Worksheet

Method of Implementation

The procedure first converts the argument `OrderSpec` into a table; each entry in the table has an index of the form `[Var,Dimen,End]` and a positive integer value indicating the required order for this state-variable along this boundary. Then it looks at each boundary form, computes its order using **pde_order**, and determines if approximation of this boundary form will require the order to be increased. Any such changes are recorded in the table, and after consideration of each boundary form the table is used to construct the final result.

The logic for determining if the order need to be increased merits a few comments. The high level criterion is that if a term cannot be accurately extracted from an approximation of the current order along the specified boundary then the order of the current order must be increased. Consider a term with a derivative in dimension d_1 of order n, applied on a boundary at one end of dimension d_2. If $d_1 = d_2$ then the derivative is in a direction perpendicular to the boundary and the order of approximation needs to be at least n in the $d_1 = d_2$ direction along this boundary, only. However, if $d_1 \neq d_2$ then the direction of the derivative is tangent to the boundary and in order to approximate the term over the entire boundary the order of approximation must be at least n in the d_1 direction at both ends.

External Dependencies

This procedure uses **order_expand** and **order_simplify** from the module ***order_ops***. The procedure **order_expand** is used to construct the table from the argument `OrderSpec` and **pde_simplify** is used to construct a simplified final result from the table.

The procedure **pde_order** is applied to each boundary form.

free_var

This procedure returns a set of potential state-variables, extracted from an expression.

free_var(Exp: *collection({algebraic, equation})*)
free_var(Exp: *collection({algebraic, equation})*, Coord: *list(name)*)

The argument Exp is expected to be either an applied or an unapplied differential form.

If Exp is in applied form, then the optional argument Coord should be used indicating the coordinate names of the problem. This use of **free_var** is the most robust since the applied form unambiguously indicates whether or not a function depends on the coordinates of the problem.

When Exp is in unapplied form, the optional argument Coord should *not* be specified. In this case, **free_var** often cannot determine by usage alone whether a symbol is a state-variable or a symbolic constant. The procedure assumes that a symbol is a state variable unless the name has been explicitly flagged as a constant with Maple's **assume** procedure. When using **free_var** on unapplied equations, the user often needs to trim the result by informing Maple about constants.

Example

Start of Maple Worksheet

```
> with(ptolemy, free_var);
```

$$[free\_var]$$

```
> free_var(D(D(V))(x) = P(x), [x]);
```

$$\{V, P\}$$

```
> free_var(D(D(V))(x) = P(x) + Q(y), [x]);
```

$$\{V, P\}$$

```
> free_var(D(D(V)) = K);
```

$$\{K, V\}$$

```
> assume(K,constant);
> free_var(D(D(V)) = K);
```

$$\{V\}$$

```
> Sys := {
>   U*D[1](U)(x,y) + V*D[2](V)(x,y) =
>    -D[1](P)(x,y)/rho + mu*(D[1,1](U)(x,y) + D[2,2](U)(x,y)),
>   U*D[1](V)(x,y) + V*D[2](V)(x,y) =
>    -D[2](P)(x,y)/rho + mu*(D[1,1](V)(x,y) + D[2,2](V)(x,y)),
>   D[1](U)(x,y) + D[2](V)(x,y) = 0};
```

$$Sys := \Big\{U\, D_1(U)(x, y) + V\, D_2(V)(x, y) =$$

$$-\frac{D_1(P)(x,y)}{\rho} + \mu\,(D_{1,1}(U)(x,y) + D_{2,2}(U)(x,y)),$$
$$U\,D_1(V)(x,y) + V\,D_2(V)(x,y) =$$
$$-\frac{D_2(P)(x,y)}{\rho} + \mu\,(D_{1,1}(V)(x,y) + D_{2,2}(V)(x,y)),$$
$$D_1(U)(x,y) + D_2(V)(x,y) = 0\Big\}$$

```
> free_var(Sys, [x,y]);
```

$$\{V, P, U\}$$

```
> Sys1 := ptolemy[pde_unapply](Sys, {V,P,U}, [x,y]);
```

$$Sys1 := \Big\{U\,D_1(U) + V\,D_2(V) = -\frac{D_1(P)}{\rho} + \mu\,(D_{1,1}(U) + D_{2,2}(U)),$$
$$U\,D_1(V) + V\,D_2(V) = -\frac{D_2(P)}{\rho} + \mu\,(D_{1,1}(V) + D_{2,2}(V)),$$
$$D_1(U) + D_2(V) = 0\Big\}$$

```
> free_var(Sys1);
```

$$\{V, P, U, \rho, \mu\}$$

```
> assume(rho,constant); assume(mu,constant);
> free_var(Sys1);
```

$$\{V, P, U\}$$

End of Maple Worksheet

Method of Implementation

A local mapping procedure is applied to all the intermediates of `Exp`. A different mapping procedure is used depending on whether `Exp` is an applied or an unapplied form (determined by the presence of the optional argument `Coord`). If an intermediate does not contain a state-variable the mapping procedure returns `NULL` which is later removed from the result.

The selection procedure for an applied form returns the variable name of any `D`-operators that are applied at the coordinates specified by `Coord`, or the function names of any functions applied to the coordinates specified by `Coord`. The selection procedure for an unapplied form returns the variable name of all `D`-operators and any intermediates which are names. In every case, only those results that are not explicitly specified as constants are returned (via the **assume** mechanism).

It would be more logical to use **is** to check if a symbol is a constant, but in some cases this causes enormous delays while Maple tries to figure out if the name can be reduced to a constant. As a result, as of Maple V R3, **isgiven** must be used instead.

Dependencies

The procedure **free_var** depends on the procedure **isDop** to parse intermediates involving `D`-operators.

get_D_forms

This function searches an expression for all of the differential forms of the state-variables.

get_D_forms(`Expression:` *anything*, `VarList:` {*set(name), list(name)*})

The return result is a sequence of the "leaves" of the expression tree, except that all leaves which are replaced by `NULL` and that subtrees that are D-forms involving one of the state-variables are returned as a unit, i.e., not traversed.

The automatic simplification rules for sequences will remove all of the `NULL`'s from the sequence, but to remove other duplicate entries the users will typically enclose the call in a set. This may be done with the command

```
DForms := { get_D_forms(D[1,2](V) = D[2,1](U), [U,V]) }
```

Example

Start of Maple Worksheet

```
> with(ptolemy, get_D_forms);
```

$$[\,get\_D\_forms\,]$$

```
> Sys0 := D(V) + (D@@2)(U) = K;
```

$$Sys0 := \mathrm{D}(\,V\,) + D^{(2)}(\,U\,) = K$$

```
> get_D_forms(Sys0, {V});
```

$$\mathrm{D}(\,V\,)$$

```
> get_D_forms(Sys0, {K});
```

$$K$$

```
> get_D_forms(Sys0, {P});
```

```
> Sys1 := D[1,1](V) + D[2,2](V) = V + U;
```

$$Sys1 := D_{1,1}(\,V\,) + D_{2,2}(\,V\,) = V + U$$

```
> get_D_forms(Sys1, {V});
```

$$D_{1,1}(\,V\,), D_{2,2}(\,V\,), V$$

```
> get_D_forms(Sys1, {U,V});
```

$$D_{1,1}(\,V\,), D_{2,2}(\,V\,), V, U$$

```
> get_D_forms(Sys1,{P});
```

```
> get_D_forms(D[1,1](V)(x1,x2) + D[2,2](V)(x1,x2) = V(x1,x2), [V]);
```

$$D_{1,1}(\,V\,)(\,x1, x2\,), D_{2,2}(\,V\,)(\,x1, x2\,), \mathrm{V}(\,x1, x2\,)$$

End of Maple Worksheet

Method of Implementation

This function first checks to see if `Exp` is:

1. A `D`-operator where the operated upon variable is a state-variables (e.g., `D[1,1](V)` where V is a state-variable).
2. A function evaluation where the function is one of the state-variable (e.g., `V(phi)` where V is a state-variable).
3. A state-variable (e.g., simply `V` where V is a state-variable).

In any of these cases the result is simply the `Args` applied to `Exp`.

If this is not the case then the program checks to see if `Exp` is a ***name***. In this case the result is simply `Exp`.

In all other cases the result is obtained by recursively applying **pde_apply** to each of the components of `Exp`.

Design Flaw

It is not possible to tell if the differential form is actually in an unapplied form. For example the expression `V(phi)` will be taken to mean $V \circ \phi$ where $\phi \in \mathbb{R}^n \rightarrow \mathbb{R}^m$ and $V \in \mathbb{R}^m \rightarrow \mathbb{R}^n$. However, this might mean that ϕ is actually the coordinate name and the user's call is erroneous. Since there is no way to detect this kind of user error the function proceeds and generates what is probably a very confusing result.

It is not clear how useful it is to handle constructs of the form $V \circ \phi$. Disallowing them would allow for detection of the common user trying to apply a PDE that is already in an applied form.

Dependencies

This function depends on **isDop** to determine if an expression of the form `D[1,1](V)` is a reference to one of the state-variables.

isDop

This function determines if the argument is an **D**-operator, possibly applied to a function.

boolean isDop(`Term`: *algebraic*)
boolean isDop(`Term`: *algebraic*, `Fun`: *name*)

If `Fun` is specified, it is the name of the variable in which the function name is to be returned.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, isDop);
                              [isDop]

> isDop(D(V));
                               true

> isDop(D(V), 'Var');
                               true

> Var;
                                V
```

```
> isDop(D[1,1](U)(x,y,z), 'Var');
                               true

> Var;
                                U
```

```
> isDop(D[1, 1](V)*V);
                               false

> isDop(D[1, 1](V) + D[1](V));
                               false
```

End of Maple Worksheet

Method of Implementation

The **D** symbol is treated as a function by Maple. So `D(V)` is actually an instance of applying the function **D** to the argument `V`. So if `Term` is a function and the zeroth operand is **D** then the first operand is the value returned in `Fun`.

However, `D[1,1](V)` is actually represented internally as (D[1,1])(V), so it is also necessary to check that the function name is of type ***indexed*** and that the zeroth operand of the function name is **D**.

In addition, `(D@@2)(V)` is actually represented as `(@@(D,2))(V)`. So it is also necessary to check to see if the function name is of type ***function*** and the function name of the function name is @@ and its first argument is **D**.

Finally, an expression of the form `D[1,1](V)(x,y)` should be recognized as a **D**-op, which is simply applied at `x,y`. To handle these cases, **isDop** recursively calls itself on the function name, terminating only when `Term` is not an applied function.

Design Flaw

There is an automatic simplification rule which converts expressions of the form `D(D(D(V)))` to `(D@@3)(V)`. As a result, **isDop** does not deal with the first case. I did not succeed in generating a meaningful example where this was a problem.

However, more effort should be invested to determine if this problem can actually arise in real applications. If it can, the design will have to significantly altered, perhaps doubling the complexity of the source code.[2]

[2]Since this is a relatively simple procedure, doubling the complexity is not a serious problem.

pde_apply

This function converts an unapplied PDE to one evaluated at a specific set of coordinates.

pde_apply(Exp: *collection({algebraic, equation})*, StateVar: *collection(name)*,
Args: *list(name)*)

Example

Start of Maple Worksheet

```
> with(ptolemy, pde_apply);
```

$$[\mathit{pde\_apply}]$$

```
> pde_apply(Q + K = V, {V}, [x,y]);
```

$$Q + K = \mathrm{V}(x, y)$$

```
> pde_apply(D[1,1](T) + D[2,2](T) = K, T, [x,y]);
```

$$D_{1,1}(T)(x,y) + D_{2,2}(T)(x,y) = K$$

```
> Sys := [
>   U*D[1](U) + V*D[2](V) = -P/rho + mu*(D[1,1](U) + D[2,2](U)),
>   U*D[1](V) + V*D[2](V) = -P/rho + mu*(D[1,1](V) + D[2,2](V)),
>   D[1](U) + D[2](V) ];
```

$$\mathit{Sys} := \Big[U\,D_1(U) + V\,D_2(V) = -\frac{P}{\rho} + \mu\,(D_{1,1}(U) + D_{2,2}(U)),$$
$$U\,D_1(V) + V\,D_2(V) = -\frac{P}{\rho} + \mu\,(D_{1,1}(V) + D_{2,2}(V)), D_1(U) + D_2(V)$$
$$\Big]$$

```
> pde_apply(Sys, {U,V,P}, [x,y]);
```

$$\Big[\mathrm{U}(x,y)\,D_1(U)(x,y) + \mathrm{V}(x,y)\,D_2(V)(x,y) =$$
$$-\frac{\mathrm{P}(x,y)}{\rho} + \mu\,(D_{1,1}(U)(x,y) + D_{2,2}(U)(x,y)),$$
$$\mathrm{U}(x,y)\,D_1(V)(x,y) + \mathrm{V}(x,y)\,D_2(V)(x,y) =$$
$$-\frac{\mathrm{P}(x,y)}{\rho} + \mu\,(D_{1,1}(V)(x,y) + D_{2,2}(V)(x,y)),$$
$$D_1(U)(x,y) + D_2(V)(x,y)\Big]$$

End of Maple Worksheet

Method of Implementation

This procedure first checks to see if `Exp` is:

1. A `D`-operator applied to a variable that is a state-variable (e.g., `D[1,1](V)` where V is a state-variable).
2. A function evaluation where the function is one of the state-variable (e.g., `V(phi)` where V is a state-variable).
3. A state-variable (e.g., simply `V` where V is a state-variable).

In each of these cases, the result is `Exp(Args)`. If `Exp` is not any of these forms and is of type ***numeric*** or type ***name***, then the result is `Exp`. This is a terminating case. In all other cases, the result is obtained by recursively applying **pde_apply** to each of the operands of `Exp`.

Design Weakness

This procedure suffers from the same design weaknesses as **get_D_forms**. Refer to section titled “isDop” on page 230for an explanation of these problems.

Dependencies

This function depends on **isDop** to determine if the expression is a `D`-operator and to extract the variable name to which the `D`-operator is applied when it is in fact a `D`-operator.

pde_unapply

This function converts a PDE applied at a specific set of coordinates into an unapplied form.

pde_apply(Exp: {*algebraic, equation, list(equation), set(equation)*},
StateVar: *collection(name)*, Coords: {*algebraic, list(algebraic)*},
ExtraCoords)

This procedure searches for state-variables (i.e., specified by `StateVar`) appearing in `Exp` which are applied at `Coords` and rewrites them in the unapplied form.

If extra coordinates are specified the result is equivalent to performing each unapply in sequence. Because functions are converted into the unapplied format only if they are applied at exactly the coordinates specified, the order in which these convections are performed does *not* affect the result.

Example

Start of Maple Worksheet

```
> with(ptolemy, pde_unapply);
```

$$[pde\_unapply]$$

```
> pde_unapply(Q(t) + K = V(x,y), V, [x,y]);
```

$$\mathrm{Q}(t) + K = V$$

```
> pde_unapply(ln(x) + (D@@2)(V)(x) = V(x), V, x);
```

$$\ln(x) + D^{(2)}(V) = V$$

```
> pde_unapply(sin(D[1](V)(x,y)) + cos(D[2](V)(x,y)) = 0, [U,V], [x,y]);
```

$$\sin(D_1(V)) + \cos(D_2(V)) = 0$$

```
> pde_unapply(S1(phi1(x1)) + S2(phi2(x2)) = k, [S1,S2], phi1(x1), phi2(x2));
```

$$S1 + S2 = k$$

```
> Exp := (D@@2)(S1)(z1)*D(z1)^2*S2(z2);
```

$$Exp := D^{(2)}(S1)(z1)\,\mathrm{D}(z1)^2\,\mathrm{S2}(z2)$$

```
> pde_unapply(Exp, {S1,S2}, z1,z2);
```

$$D^{(2)}(S1)\,\mathrm{D}(z1)^2\,S2$$

```
> Exp := sqrt(2)/2 * (D[1](V)(y1,y2) * D[2](y1)(x1,x2) + D[2](V)(y1,y2) * D(y2)(x2));
```

$$Exp := \frac{1}{2}\sqrt{2}\,(D_1(V)(y1, y2)\,D_2(y1)(x1, x2) + D_2(V)(y1, y2)\,\mathrm{D}(y2)(x2))$$

```
> pde_unapply(Exp, [y1,y2], [x1,x2], x2);
```

$$\frac{1}{2}\sqrt{2}\,(D_1(\,V\,)(\,y1\,,y2\,)\,D_2(\,y1\,)+D_2(\,V\,)(\,y1\,,y2\,)\,\mathrm{D}(\,y2\,))$$

End of Maple Worksheet

Method of Implementation

If extra coordinates have been specified, conversions are first performed with respect to these coordinates (this is done via recursive calls) and the result is then unapplied with respect to the coordinates specified by `Coords`.

The procedure first checks to see if the expression being rewritten is a function and if it is one of the state-variables applied to the current coordinate(s). In this case it simply returns the state-variable. Otherwise it checks to see if the expression is a `D`-operator applied to a state-variable which collectively is applied to the current coordinate(s). Again if this is the case the state-variable is returned. For all other functions the result is constructed by unapplying each of the arguments.

Otherwise the result is constructed by recursively unapplying to each of the functions operands. The exceptions are atomic expressions (i.e., expressions that do not divide into subexpressions), floats, and fractions.

Dependencies

This procedure depends on **isDop** to determine if the expression is a `D`-operator and to extract the variable name to which the `D`-operator is applied when it is in fact a ‘`D`-operator.

pde_collect

This function collects like terms of an unapplied PDE.

pde_collect(Eq: *collection({algebraic, equation})*, StateVar: *collection(name)*)
pde_collect(Eq: *collection({algebraic, equation})*, StateVar: *collection(name)*, SimpProc: *procedure*)
pde_collect(Eq: *collection({algebraic, equation})*, StateVar: *collection(name)*, SimpProc: *procedure*, ExtraArgs)

This function will collect like forms of the state-variables in each subequation within Exp. The coefficients will then be combined into a single coefficient of a single term per state-variable form. If a simplification procedure is provided it will be applied to each coefficient before the final term is produced; if no simplification procedure is specified the output coefficient will be the simple sum of the coresponding input coefficients. If extra arguments are provided they will be passed to the simplification procedure, following the term to be simplified.

Example

Start of Maple Worksheet

```
> with(ptolemy, pde_collect, pde_unapply):
> t1 := diff(S(ln(x/(1-x))), x$2);
```

$$t1 := \frac{D^{(2)}(S)\left(\ln\left(\frac{x}{1-x}\right)\right)\left(\frac{1}{1-x}+\frac{x}{(1-x)^2}\right)^2(1-x)^2}{x^2} + \frac{\mathrm{D}(S)\left(\ln\left(\frac{x}{1-x}\right)\right)\left(2\,\frac{1}{(1-x)^2}+2\,\frac{x}{(1-x)^3}\right)(1-x)}{x} - \frac{\mathrm{D}(S)\left(\ln\left(\frac{x}{1-x}\right)\right)\left(\frac{1}{1-x}+\frac{x}{(1-x)^2}\right)(1-x)}{x^2} - \frac{\mathrm{D}(S)\left(\ln\left(\frac{x}{1-x}\right)\right)\left(\frac{1}{1-x}+\frac{x}{(1-x)^2}\right)}{x}$$

```
> t2 := pde_unapply(t1, [ln(x/(1-x))]);
```

$$t2 := \frac{D^{(2)}(S)\left(\frac{1}{1-x}+\frac{x}{(1-x)^2}\right)^2(1-x)^2}{x^2} + \frac{\mathrm{D}(S)\left(2\,\frac{1}{(1-x)^2}+2\,\frac{x}{(1-x)^3}\right)(1-x)}{x} - \frac{\mathrm{D}(S)\left(\frac{1}{1-x}+\frac{x}{(1-x)^2}\right)(1-x)}{x^2} - \frac{\mathrm{D}(S)\left(\frac{1}{1-x}+\frac{x}{(1-x)^2}\right)}{x}$$

```
> t3 := unapply(t2,x)(exp(z) / (1+exp(z)));
```

$$t3 := \frac{D^{(2)}(S)\,\%1^2\,(1+e^z)^2\left(1-\frac{e^z}{1+e^z}\right)^2}{(e^z)^2} + D(S)$$
$$\left(2\,\frac{1}{\left(1-\frac{e^z}{1+e^z}\right)^2} + 2\,\frac{e^z}{(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)^3}\right)(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)/($$
$$e^z) - \frac{D(S)\,\%1\,(1+e^z)^2\left(1-\frac{e^z}{1+e^z}\right)}{(e^z)^2} - \frac{D(S)\,\%1\,(1+e^z)}{e^z}$$
$$\%1 := \frac{1}{1-\frac{e^z}{1+e^z}} + \frac{e^z}{(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)^2}$$

```
> pde_collect(t3,[S]);
```

$$\frac{D^{(2)}(S)\,\%1^2\,(1+e^z)^2\left(1-\frac{e^z}{1+e^z}\right)^2}{(e^z)^2} + \Bigg($$
$$\left(2\,\frac{1}{\left(1-\frac{e^z}{1+e^z}\right)^2} + 2\,\frac{e^z}{(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)^3}\right)(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)/($$
$$e^z) - \frac{\%1\,(1+e^z)^2\left(1-\frac{e^z}{1+e^z}\right)}{(e^z)^2} - \frac{\%1\,(1+e^z)}{e^z}\Bigg)\,D(S)$$
$$\%1 := \frac{1}{1-\frac{e^z}{1+e^z}} + \frac{e^z}{(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)^2}$$

```
> pde_collect(t3,[S],simplify);
```

$$(-e^{(-z)} + 1 + e^z - e^{(-2\,z)})\,(1+e^z)\,D(S) + (1+e^z)^4\,e^{(-2\,z)}\,D^{(2)}(S)$$

```
> pde_collect(t3,[S],factor);
```

$$\frac{(e^z-1)\,(1+e^z)^3\,D(S)}{(e^z)^2} + \frac{(1+e^z)^4\,D^{(2)}(S)}{(e^z)^2}$$

End of Maple Worksheet

Method of Implementation

The function **pde_collect** recursively calls itself until `Eq` is an algebraic form (i.e., not an equation or a collection of equations). Then it

1. Extracts all intermediates from the subequation. Maple's definition of an intermediate is broader than the concept of a free variable (as commonly defined in lambda calculus). As a result the list of intermediates will include all of the differential forms of the state-variables.

   As of the time of this writing, Maple R3 is the current version of Maple. The R3 documentation does not explicitly state the definition of an intermediate. Experimentation suggests that an intermediate of an expression is: 1) any free variable of the expression, 2) any applied function in the expression, and 3) any intermediates of arguments of any applied functions. Since the Maple **D**-operator is implemented as a function, subexpressions of the form `D[1,1](V)` are intermediates.

2. Selects all intermediates containing state-variables to form the list of differential forms to be collected against.

3. Performs a **collect/distributed** on the subexpression in order do the actual collection of like terms. See the Maple on-line help on **collect**.

4. If a simplification procedure is specified then

   (a) The coefficients and terms are separated into two sequences using the **coeffs** procedure.

   (b) The simplification procedure is applied to each coefficients, using any extra arguments.

   (c) The two sequences are recombined into a single expression using the **sum** function.

Design Flaws

For consistency with other parts of the **PTOLEMY** package, the value of the global variable `ptolemy/SimpProc` should be used to determine the simplification procedure.

It should be possible for the user to specify the differential forms against which collection occurs, but then the functionality of figuring out all of the differential forms in a subexpression would need to be packaged into a separate routine. This is not done simply because no application has yet arisen that requires the user to specify only some subset of the differential forms.

An expansion of expressions involving only `D`-operators would probably cause some unusual constructs to be better handled. At the moment `(D@(1 + D))(V)` is not expanded to be `D(V) + (D^2)(V)` but is rather treated as a distinct differential form and is not collected against the forms `D(V)` or `(D^2)(V)`.

It is not clear how some kinds of functions of differential forms should be treated. For example a human might choose to rewrite

```
sin(D[1,1](V) + D[2,2](V)) + sin(D[1,1](V) - D[2,2](V))
```

as

```
2*sin(D[1,1](V))*cos(D[2,2](V))
```

pde_collect attempts nothing of the sort.

Granularity of Collection

Though **pde_collect** will operate on an entire collection of PDEs, like terms are found only on one side of each equation. Sometimes the users would like to collect like terms from both sides of the equation. Because there are other times when this would be inappropriate (for example when each side of the equation is applied in a different domain), the user who wants to do this will have to provide additional functionality.

This performance granularity is the same as that provided by Maple's standard **collect** procedure. For example:

```
collect(x^2 + x = -x, x)
```

produces

```
x^2 + x = -x, x
```

fast_map

This procedure performs a highly optimized mapping of an unapplied equation (and lists or sets of equations and expressions).

fast_map(`Exp`: *collection({algebraic, equation})*, `Map`: *procedure*, `InvMap`: *procedure*, `Coord`: *list(name)*, `NewCoordDepend`: *list(function)*, `Subs`: *set(equation)*, `OutVarDepend`: *collection(function)*)

There are easier ways of mapping equations than the one used by this procedure, but simpler methods are significantly slower and more vulnerable to the limitations of simplification procedures. Both problems can be especially troubling for higher order maps and higher order equations. Pre-prototype versions of this procedure took two hours on a high end work-station to map equation sets used in testing the module `CollocateRec`; this version takes only tens of seconds for the same problems. Because mapping is central to **PTOLEMY** the extra complexity of this procedure is necessary. This procedure is intended as an internal routine for code developers; most interactive users are encouraged to use higher level routines, such as **Warp**, instead.

The procedure **fast_map** uses the global variable `ptolemy/SimpProc` to determine which simplification procedure is to be applied to the various parts of its result.

Synopsis

Conceptually the argument `Exp` defines the "equation" to be mapped. However, as indicated by the type definition in the prototype, `Exp` need not be an equation.

The arguments `Map` and `InvMap` define the map to be applied to the "equation." Although the type restriction in the prototype allows for arbitrary procedures, **fast_map** places two additional restrictions on these arguments:

1. Both procedures must be functions in the mathematical sense (e.g., idempotent and with well-defined results over the entire range).

2. The results of both procedures must be defined for purely symbolic arguments. (In addition the result when the arguments are purely symbolic must be consistent with the results for numerical arguments).

It is imagined that `Map` and `InvMap` will be arrow operators without any control structures, such as "if statements," but this is not necessary as long as the above two conditions are meet.

The arguments `Coord` and `NewCoordDepend` define names that may be safely used to represent the coordinates in both the original domain and the mapped to domain. These names must not conflict with each other, with any global definitions, or with any symbols used in the "equation."

The argument `NewCoordDepend` also indicates, for each new coordinate, the dependencies on the original coordinates. That is each new coordinate is a function of some or all of the original coordinates. If a new coordinate does not depend on all of the old coordinates this information is exploited by **fast_map** to improve the efficiency of the mapping. Syntactically, this information is encoded in the argument `NewCoordDepend` by applying the new coordinate name to the subset of `Coord` on which it depends. So the new coordinate names are only defined indirectly as the function names in the list of functions.

For example, if the original coordinates are `x1` and `x2` and the new coordinate names are `z1` and `z2`, in the most general map both `z1` and `z2` would depend on both `x1` and `x2`. This would be indicated by setting `NewCoordDepend` to

```
[z1(x1,x2),z2(x1,x2)].
```

If for a particular map the value of `z1` depends only on the value of `x1` (and not on the value `x2`) but `z2` depends on both `x1` and `x2`, this could be indicated by setting `NewCoordDepend` to

```
[z1(x1),z2(x1,x2)].
```

The procedure `fast_map` also allows use of different state variable names for the mapped "equation" than those used in the original "equation." The name of the new state-variables is given in the argument `OutVarDepend` and their relationship to the original state-variables is specified by the argument `Subs`.

The argument `Subs` is a set of equations which if converted to assignments, would define replacement for the original state-variable in terms of the new state-variables. The functional relationship between the original state-variables and the new state-variables may be nontrivial. The associated substitution takes place before the mapping is performed, so any representation to the problem coordinates in `Subs` should be to the original problem coordinates. If an input state-variables names is used to represent the same state variable after mapping then no substitution equation is required for this state-variable.

For example, if the only state-variable is `Speed`, then the typical value of `Subs` would be {} (which is equivalent to `Speed = Speed`). It is possible to replace the state variable `Speed` with two new state-variables representing vector velocities while performing this mapping. Assuming cartesian coordinates, this might be done by letting `Subs` equal

```
Speed = sqrt(Left^2 + Up^2).
```

The argument `OutVarDepend` defines all of the output state variable names and their dependencies on the output coordinates. The syntax is strictly analogous to the argument `NewCoordDepend`. So, continuing the previous example, if `OutVarDepend` were equal to

```
{ Left(z1), Up(z1,z2) }
```

this would indicate that the the new state-variable `Left` only depends on `z1`. This is equivalent to asserting that, with mapping, the `z2` lines are the iso-`Left` lines. This information can save a great deal of unnecessary work.

Example Usage

```
> with(ptolemy, fast_map);
```

$$[\mathit{fast\_map}]$$

```
> MapInfo := ptolemy[LogRatioMap](0,1,x1,z1);
```

$$\mathit{MapInfo} := \left[\mathit{x1} \rightarrow \ln\left(\frac{\mathit{x1}}{1-\mathit{x1}}\right), \mathit{z1} \rightarrow \frac{e^{\mathit{z1}}}{1+e^{\mathit{z1}}}, \mathit{x1} \rightarrow (\,\mathit{x1}, 1-\mathit{x1}\,)\right]$$

```
> fast_map([1-x,x], MapInfo[1],MapInfo[2], [x],[z(x)], {}, {});
```

$$\left[\frac{1}{1+e^z}, \frac{e^z}{1+e^z}\right]$$

```
> g := sum('a.i'*x^i, i=0..2);
```

$$g := a0 + a1\,x + a2\,x^2$$

```
> temp := fast_map(
>   [x,1-x, f = g + r], MapInfo[1],MapInfo[2], [x],[z(x)], {F=f,R=r}, {F(z),R(z)});
```

$$temp := \left[\frac{e^z}{1+e^z}, \frac{1}{1+e^z}, f = \left(a0 + 2\,a0\,e^z + a0\,(e^z)^2 + a1\,e^z + a1\,(e^z)^2 + a2\,(e^z)^2 + r + 2\,r\,e^z + r\,(e^z)^2\right) \Big/ (1+e^z)^2\right]$$

```
> Sys := D[1,1](V) + D[2,2](V) = K;
```

$$Sys := D_{1,1}(V) + D_{2,2}(V) = K$$

```
> Coord := [x1,x2]; NewCoord := [z1(x1),z2(x2)];
```

$$Coord := [x1, x2]$$

$$NewCoord := [\mathrm{z1}(x1), \mathrm{z2}(x2)]$$

```
> MapInfo := [ ptolemy[LogRatioMap](0,w,x1,z1),
>   ptolemy[LogRatioMap](0,h,x2,z2) ];
```

$$MapInfo := \left[\left[x1 \to \ln\left(\frac{x1}{w-x1}\right), z1 \to \frac{w\,e^{z1}}{1+e^{z1}}, x1 \to \left(\frac{x1}{\sqrt{w}}, \frac{w-x1}{\sqrt{w}}\right)\right], \left[x2 \to \ln\left(\frac{x2}{h-x2}\right), z2 \to \frac{h\,e^{z1}}{1+e^{z1}}, x2 \to \left(\frac{x2}{\sqrt{h}}, \frac{h-x2}{\sqrt{h}}\right)\right]\right]$$

```
> Map := subs(RESULT = seq(MapInfo[i][1](Coord[i]), i=1..2), (x1,x2)-> RESULT);
```

$$Map := (x1, x2) \to \left(\ln\left(\frac{x1}{w-x1}\right), \ln\left(\frac{x2}{h-x2}\right)\right)$$

```
> InvMap := subs(RESULT = (MapInfo[1][2](z1), MapInfo[2][2](z2)),
>   (x1,x2)-> RESULT);
```

$$InvMap := (x1, x2) \to \left(\frac{w\,e^{z1}}{1+e^{z1}}, \frac{h\,e^{z2}}{1+e^{z2}}\right)$$

```
> Subs := { V = S1*S2 };
```

$$Subs := \{V = S1\ S2\}$$

```
> ZForms := [z1(x1), z2(x2)];
```

$$ZForms := [\mathrm{z1}(x1), \mathrm{z2}(x2)]$$

```
> OutVar := {S1(z1),S2(z2)};
```

$$OutVar := \{\mathrm{S1}(z1), \mathrm{S2}(z2)\}$$

```
> fast_map(Sys, Map,InvMap, Coord,NewCoord, Subs,OutVar);
```

$$\frac{D^{(2)}(\mathit{S1})\,(1+e^{\mathit{z1}})^4\,\mathit{S2}}{w^2\,(e^{\mathit{z1}})^2}+\frac{\mathrm{D}(\mathit{S1})\,(e^{\mathit{z1}}-1)\,(1+e^{\mathit{z1}})^3\,\mathit{S2}}{w^2\,(e^{\mathit{z1}})^2}+\frac{\mathit{S1}\,D^{(2)}(\mathit{S2})\,(1+e^{\mathit{z2}})^4}{h^2\,(e^{\mathit{z2}})^2}+\frac{\mathit{S1}\,\mathrm{D}(\mathit{S2})\,(e^{\mathit{z2}}-1)\,(1+e^{\mathit{z2}})^3}{h^2\,(e^{\mathit{z2}})^2}=K$$

```
> Subs := { V = S1*B2 };
```

$$\mathit{Subs} := \{V = \mathit{S1}\ \mathit{B2}\}$$

```
> OutVar := {S1(z1),B2(x2)};
```

$$\mathit{OutVar} := \{\mathrm{S1}(\mathit{z1}), \mathrm{B2}(\mathit{x2})\}$$

```
> fast_map(Sys, Map,InvMap, Coord,NewCoord, Subs,OutVar);
```

$$\frac{D^{(2)}(\mathit{S1})\,(1+e^{\mathit{z1}})^4\,\mathit{B2}}{w^2\,(e^{\mathit{z1}})^2}+\frac{\mathrm{D}(\mathit{S1})\,(e^{\mathit{z1}}-1)\,(1+e^{\mathit{z1}})^3\,\mathit{B2}}{w^2\,(e^{\mathit{z1}})^2}+\mathit{S1}\,D^{(2)}(\mathit{B2})=K$$

```
> Subs := { V = x1*S1 * (h-x2)*S2 + B1*B2 };
```

$$\mathit{Subs} := \{V = \mathit{x1}\ \mathit{S1}\ (h-\mathit{x2})\ \mathit{S2} + \mathit{B1}\ \mathit{B2}\}$$

```
> OutVar := {S1(z1),S2(z2), B1(x1),B2(x2)};
```

$$\mathit{OutVar} := \{\mathrm{S1}(\mathit{z1}), \mathrm{B2}(\mathit{x2}), \mathrm{B1}(\mathit{x1}), \mathrm{S2}(\mathit{z2})\}$$

```
> Start := time():
> fast_map(Sys, Map,InvMap, Coord,NewCoord, Subs,OutVar);
```

$$D^{(2)}(\mathit{B1})\,\mathit{B2}+\mathit{B1}\,D^{(2)}(\mathit{B2})+\frac{(1+e^{\mathit{z2}})^3\,w\,e^{\mathit{z1}}\,D^{(2)}(\mathit{S2})\,\mathit{S1}}{h\,(e^{\mathit{z2}})^2\,(1+e^{\mathit{z1}})}+\frac{h\,(1+e^{\mathit{z1}})^3\,\mathit{S2}\,\mathrm{D}(\mathit{S1})}{(1+e^{\mathit{z2}})\,w\,e^{\mathit{z1}}}+\frac{h\,(1+e^{\mathit{z1}})^3\,\mathit{S2}\,D^{(2)}(\mathit{S1})}{(1+e^{\mathit{z2}})\,w\,e^{\mathit{z1}}}-\frac{(1+e^{\mathit{z2}})^3\,w\,e^{\mathit{z1}}\,\mathrm{D}(\mathit{S2})\,\mathit{S1}}{h\,(e^{\mathit{z2}})^2\,(1+e^{\mathit{z1}})}=K$$

```
> time() - Start;
```

$$2.734$$

```
> L := x -> 8*x^2*(1-x)^2;
```

$$L := x \to 8\,x^2\,(1-x)^2$$

```
> TradDomain := [[x1,x2], x2= 0..1, x1=-L(x2)..1+L(x2)];
```

$$\mathit{TradDomain} := \left[[\mathit{x1},\mathit{x2}], \mathit{x2}=0..1, \mathit{x1}=-8\,\mathit{x2}^2\,(1-\mathit{x2})^2..1+8\,\mathit{x2}^2\,(1-\mathit{x2})^2\right]$$

```
> type(TradDomain,TradDomainType);
```

$$\mathit{true}$$

```
> ptolemy[PlotDomBound](TradDomain, scaling=constrained, axes=framed);
```
See Figure 7.4
```
> ptolemy[MakeTradMap](TradDomain, 'Map', 'InvMap');
> eval(Map); eval(InvMap);
```
$$(\,x1, x2\,) \to \left(\frac{x1 + 8\,x2^2 - 16\,x2^3 + 8\,x2^4}{1 + 16\,x2^2 - 32\,x2^3 + 16\,x2^4}, x2\right)$$

$$(\,x1, x2\,) \to (\,x1 + 16\,x1\,x2^2 - 32\,x1\,x2^3 + 16\,x1\,x2^4 - 8\,x2^2 + 16\,x2^3 - 8\,x2^4, x2\,)$$

```
> Subs := { V = V };
```
$$Subs := \{\,V = V\,\}$$

```
> NewCoord := [z1(x1,x2),z2(x1,x2)];
```
$$NewCoord := [\mathrm{z1}(\,x1, x2\,), \mathrm{z2}(\,x1, x2\,)\,]$$

```
> OutVar := { V(z1,z2) };
```
$$OutVar := \{\,\mathrm{V}(\,z1, z2\,)\,\}$$

```
> Start := time():
> fast_map(Sys, Map,InvMap, Coord,NewCoord, Subs,OutVar);
```

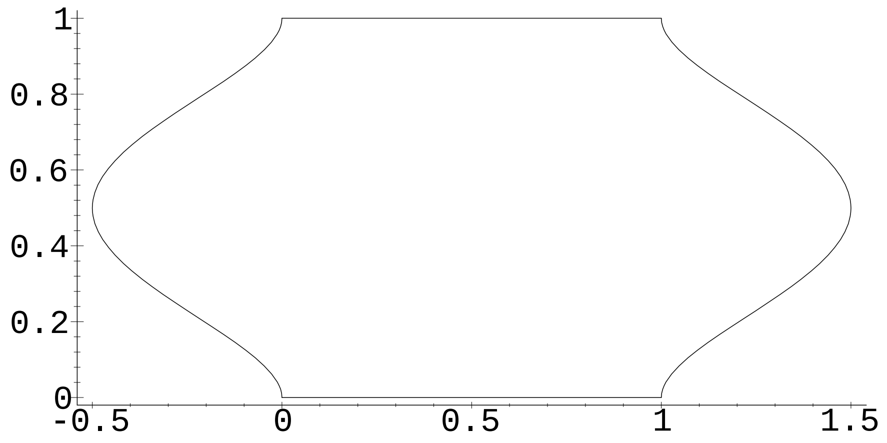

Figure 7.4: A Vase Shaped Subdomain

$$-32\,\frac{z2\,(2\,z2-1)\,(z2-1)\,(-1+2\,z1)\,D_{1,2}(V)}{1+16\,z2^2-32\,z2^3+16\,z2^4}+D_{2,2}(V)+(1+256\,z2^2$$
$$-1024\,z1\,z2^2+1024\,z2^2\,z1^2-1536\,z2^3+6144\,z1\,z2^3-6144\,z2^3\,z1^2$$
$$+3328\,z2^4-13312\,z1\,z2^4+13312\,z2^4\,z1^2-3072\,z2^5+12288\,z1\,z2^5$$
$$-12288\,z2^5\,z1^2+1024\,z2^6-4096\,z1\,z2^6+4096\,z2^6\,z1^2)D_{1,1}(V)\Big/$$
$$(1+16\,z2^2-32\,z2^3+16\,z2^4)^2+16$$
$$(42\,z2^2-256\,z2^3+528\,z2^4-480\,z2^5+160\,z2^6-1+6\,z2)\,(-1+2\,z1)$$
$$D_1(V)\Big/(1+16\,z2^2-32\,z2^3+16\,z2^4)^2=K$$

```
> time() - Start;
                                        4.666
```

Coordinate Names

The coordinate names instruct **fast_map** about which symbols in the "equation" correspond to the coordinates of the domain. These symbols get converted to the new coordinate names, whereas other undefined symbols are assumed to be constants. The new coordinate names instruct **fast_map** of which symbols to use in its result to represent the coordinates of the domain.

These symbols cannot evaluate to anything other than a symbol. Even when either the input or the output make no explicit reference to the coordinates of the domain, **fast_map** must make internal use of these symbols. So, it is important that these names never evaluate to anything other than a symbol.

One could imagine an "equation" defined implicitly by procedure whose arguments are the domain's coordinate names. Many general purpose procedures return results in this form to ensure that no naming conflicts occur. In this case it may be necessary to search the global symbol table in order to be sure that the selected coordinate names are not previously used. This can be done in Maple, but such an approach is undesirably slow. A more efficient, but more complicated, approach in such cases would be to require `Exp` to be a procedure whose arguments are the coordinate names and to return the results as a procedure whose arguments are coordinate names. Internally the local variables, which would naturally not conflict with previous global definitions, could be used for the argument names, but representing all expressions as procedures greatly complicates the use of **PTOLEMY**.

The justification for this particular design is that most of the time the user may employ simple conventions to ensure the uniqueness of the coordinate names, resulting in a faster system. However, as is generally the case, this improvement in efficiency is obtained by forcing the user to provide more information.

Method of Implementation

Perhaps the simplest way of mapping an equation is to 1) apply the state-variables at the map of the original coordinates, 2) expand any derivatives, and 3) substitute the inverse map of the new coordinates for the old coordinates. This method can be performed in between two and six lines of code, but as mentioned in the introduction to this section the performance is often unacceptable. This is caused by three facts. First, the intermediate result is often orders of magnitude bigger than the final simplified result. Second, simplifying this intermediate result is often excruciatingly slow and sometimes beyond the capabilities of current technology. Finally, the same operations are often repeated frequently.

To illustrate these problems, consider the following extremely simple example.

Start of Maple Worksheet

```
> Map := x -> ln(x/(1-x));
```

$$Map := x \to \ln\left(\frac{x}{1-x}\right)$$

```
> InvMap := z -> exp(z) / (1 + exp(z));
```

$$InvMap := z \to \frac{e^z}{1+e^z}$$

```
> Eq := (D@@2)(V);
```

$$Eq := D^{(2)}(V)$$

```
> Step1 := subs(V(x) = V(Map(x)), convert(Eq(x), diff));
```

$$Step1 := \frac{\partial^2}{\partial x^2}\,\mathrm{V}\left(\ln\left(\frac{x}{1-x}\right)\right)$$

```
> Step2 := eval(Step1);
```

$$Step2 := \frac{D^{(2)}(V)\left(\ln\left(\frac{x}{1-x}\right)\right)\left(\frac{1}{1-x}+\frac{x}{(1-x)^2}\right)^2(1-x)^2}{x^2}$$
$$+\frac{\mathrm{D}(V)\left(\ln\left(\frac{x}{1-x}\right)\right)\left(2\,\frac{1}{(1-x)^2}+2\,\frac{x}{(1-x)^3}\right)(1-x)}{x}$$
$$-\frac{\mathrm{D}(V)\left(\ln\left(\frac{x}{1-x}\right)\right)\left(\frac{1}{1-x}+\frac{x}{(1-x)^2}\right)(1-x)}{x^2}$$
$$-\frac{\mathrm{D}(V)\left(\ln\left(\frac{x}{1-x}\right)\right)\left(\frac{1}{1-x}+\frac{x}{(1-x)^2}\right)}{x}$$

```
> Result := subs(x = InvMap(z), Step2);
```

$$Result := \frac{D^{(2)}(V)(\%2)\,\%1^2\,(1+e^z)^2\left(1-\frac{e^z}{1+e^z}\right)^2}{(e^z)^2}+\mathrm{D}(V)(\%2)$$
$$\left(2\,\frac{1}{\left(1-\frac{e^z}{1+e^z}\right)^2}+2\,\frac{e^z}{(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)^3}\right)(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)/($$

$$e^z) - \frac{D(V)(\%2)\,\%1\,(1+e^z)^2\left(1-\frac{e^z}{1+e^z}\right)}{(e^z)^2} - \frac{D(V)(\%2)\,\%1\,(1+e^z)}{e^z}$$

$$\%1 := \frac{1}{1-\frac{e^z}{1+e^z}} + \frac{e^z}{(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)^2}$$

$$\%2 := \ln\left(\frac{e^z}{(1+e^z)\left(1-\frac{e^z}{1+e^z}\right)}\right)$$

```
> ptolemy[fast_map](Eq, Map,InvMap, [x], [z(x)], {V=V}, {V(z)});
```

$$\frac{D^{(2)}(V)(1+e^z)^4}{(e^z)^2} + \frac{D(V)(e^z-1)(1+e^z)^3}{(e^z)^2}$$

End of Maple Worksheet

Obviously the result of the of the direct approach needs to be simplified to the result produced by **fast_map**. In this case the simplification can be done, but its quite a bit harder than just invoking the **simplify** procedure. More importantly if this simple equation yields such a complex intermediate result which must then be simplified, it is reasonable to expect that the complexity of intermediate results would exceed the simplification capabilities of current technology for even moderately complicated equations.

The strategy for dealing with this problem, used throughout **PTOLEMY**, is to avoid dramatically increasing the complexity of the result before trying to simplify the result. Instead, this procedure performs intermediate simplifications whenever possible.

This example also illustrates the problem of recomputing the same result, although it is less obvious. In Step 2 the derivative of `ln(x/(1-x))` is computed four times, once for each term.

Conceptual Description

The method used by **fast_map** is to:

1. Represent `Map(x)` in step 1 as an unevaluated function.
2. Expand all the derivatives, yielding only generic forms of the derivatives of `Map(x)` (i.e., representations like `D(Map)(x)`, which contains no information about the actual map).
3. Extract all of the differential forms of the map from this result, figure out what each differential form is for the specific map, and substitute these results back into the equation.

This ensures that a given order derivative of the map needs to be computed only once.

Detailed Description

The procedure actually performs the following lengthy list of ordered steps:

1. Apply the "equation" at the coordinates specified by `Coord`.
2. Use the values of `Coord` and `OutVarDepend` to convert the unapplied form of `Subs` into an applied form.

3. Convert the applied "equation" to `diff` form, perform the specified substitution, expand the derivatives, and convert the result back to `D` form.
4. Convert the applied form of the new coordinate names to their unapplied form when they are not in a differential form by invoking the procedure **subs**. It is more common to have the coordinate names not in the differential form, and doing a **subs** is much faster than the next step, invoking **pde_unapply**.
5. Convert any remaining applied forms of the coordinate names to their unapplied forms by invoking **pde_unapply**.
6. Convert all applied forms of the output variables (extracted from the argument `OutVarDepend`) to their unapplied form.
7. Invoke **pde_collect** to merge similar terms and to invoke the simplification procedure specified by `ptolemy/SimpProc` on all of the coefficients. That is, perform an intermediate simplification of the result.
8. Use `get_D_forms` to construct a list of all of the differential forms of the new coordinates names which appear in the result. Care must be taken to exclude nondifferential forms of the new coordinates.
9. Use `Map`, `InvMap`, and `NewCoordDepend` to figure out what each of these differential forms maps to in the new coordinate system.
10. Invoke `ptolemy/SimpProc` on each of these replacements for the differential forms of the new coordinates.
11. Substitute the simplified replacements back into the "equation."
12. Replace any remaining occurrences of the original coordinate names with their equivalence in terms of the new coordinate names.
13. Finally, invoke **pde_collect** to simplify the final result.

unit_normal

This procedure will return a vector valued function that is normal to a specified side of an arbitrary domain, provided the domain is representable as a *DomainType*.

unit_normal(`Domain:` *DomainType*, `DimNum:` *posint*, `End:` *EndType*)

The result will be a list of vector coordinates, each of which is a function of the domain's coordinates.

If the argument `Domain` is of type *MappedDomainType*, then the result is a function of the coordinates of the mapped-to domain. That is, **unit_normal** returns a vector that is normal to the boundary in the original domain, but that has been mapped to the mapped-to domain. This is the natural way to construct the unit normal in this case. Of course, the user can map this result back to the mapped-from domain. This is not done automatically because the user will typically want the normal to the boundary of the original domain, mapped to the mapped-to domain. Mapping from the final domain to the original domain and having the user map this result back again is obviously inefficient, but a much more serious problem with doing this is that the result may not be as simplified after all of this mapping as it was when the process started.

Example

Start of Maple Worksheet

```
> with(ptolemy,unit_normal, PlotDomBound);
```

$$[\,PlotDomBound, unit\_normal\,]$$

```
> Domain := [[x,y], 0..1, 0..1];
```

$$Domain := [\,[\,x, y\,], 0..1, 0..1\,]$$

```
> seq(seq(unit_normal(Domain,i,End), End=[LOW,HIGH]), i=1..2);
```

$$[\,1, 0\,], [\,-1, 0\,], [\,0, 1\,], [\,0, -1\,]$$

```
> DomainA := [[x1,x2], x2=0..1, x1=0..2-x2];
```

$$DomainA := [\,[\,x1, x2\,], x2 = 0..1, x1 = 0..2 - x2\,]$$

```
> DomainB := [[x1,x2], x1=1..2, x2=2-x1..2];
```

$$DomainB := [\,[\,x1, x2\,], x1 = 1..2, x2 = 2 - x1..2\,]$$

```
> PlotDomBound({DomainA,DomainB},scaling=constrained);
```

See Figure 7.5

```
> unit_normal(DomainA,1,LOW);
```

$$[\,1, 0\,]$$

```
> seq(seq(unit_normal(DomainA,i,End), End=[LOW,HIGH]), i=1..2);
```

$$[\,1, 0\,], \left[-\frac{1}{2}\sqrt{2}, -\frac{1}{2}\sqrt{2}\right], [\,0, 1\,], [\,0, -1\,]$$

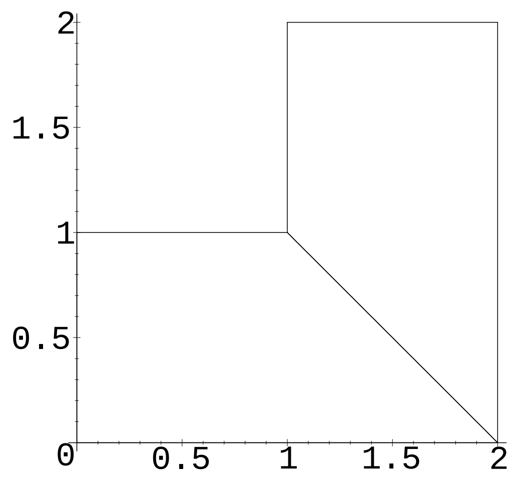

Figure 7.5: The Domains of The "L"-Problem

Maple Worksheet Continued from Previous Page

```
> seq(seq(unit_normal(DomainB,i,End), End=[LOW,HIGH]), i=1..2);
```

$$[1,0],[-1,0],\left[\frac{1}{2}\sqrt{2},\frac{1}{2}\sqrt{2}\right],[0,-1]$$

```
> Domain := [[x,y], y=0..1, x=y*(1-y)..1+sin(Pi*y)];
```

$$Domain := [[x,y],y=0..1,x=y\,(1-y)..1+\sin(\pi\,y)]$$

```
> PlotDomBound(Domain,scaling=constrained);
```

See Figure 7.6

```
> seq(seq(unit_normal(Domain,i,End), End=[LOW,HIGH]), i=1..2);
```

Maple Worksheet Continued on Next Page

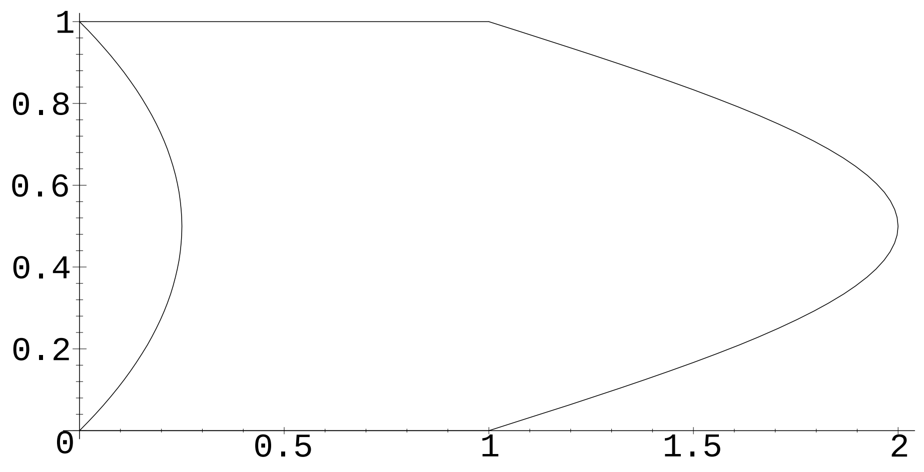

Figure 7.6: A Bullet Shaped Domain

$$\left[\frac{1}{\sqrt{2-4\,y+4\,y^2}}, \frac{-1+2\,y}{\sqrt{2-4\,y+4\,y^2}}\right],$$

$$\left[-\frac{1}{\sqrt{1+\cos(\,\pi\,y\,)^2\,\pi^2}}, \frac{\cos(\,\pi\,y\,)\,\pi}{\sqrt{1+\cos(\,\pi\,y\,)^2\,\pi^2}}\right], [0,1], [0,-1]$$

```
> Map := (x,y) -> (sqrt(x,y), tan(x,y));
```

$$Map := (\,x, y\,) \rightarrow (\,\mathrm{sqrt}(\,x, y\,), \tan(\,x, y\,)\,)$$

```
> InvMap := (r,theta) -> (r*cos(theta),r*sin(theta));
```

$$InvMap := (\,r, \theta\,) \rightarrow (\,r\cos(\,\theta\,), r\sin(\,\theta\,)\,)$$

```
> Domain := [[[r,theta], 1/2..1, -Pi/4..Pi/4], Map,InvMap];
```

$$Domain := \left[\left[[\,r, \theta\,], \frac{1}{2}..1, -\frac{1}{4}\,\pi..\frac{1}{4}\,\pi\right], Map, InvMap\right]$$

```
> PlotDomBound(Domain,scaling=constrained, xtickmarks=3);
```

See Figure 7.7

```
> seq(seq(unit_normal(Domain,i,End), End=[LOW,HIGH]), i=1..2);
```

$$\left[\frac{\cos(\,\theta\,)}{\sqrt{\%1}}, \frac{\sin(\,\theta\,)}{\sqrt{\%1}}\right], \left[-\frac{\cos(\,\theta\,)}{\sqrt{\%1}}, -\frac{\sin(\,\theta\,)}{\sqrt{\%1}}\right], \left[\frac{1}{2}\sqrt{2}, \frac{1}{2}\sqrt{2}\right], \left[\frac{1}{2}\sqrt{2}, -\frac{1}{2}\sqrt{2}\right]$$

$$\%1 := \cos(\,\theta\,)^2 + \sin(\,\theta\,)^2$$

```
> `ptolemy/SimpProc` := simplify;
```

ptolemy/SimpProc := simplify

Maple Worksheet Continued on Next Page

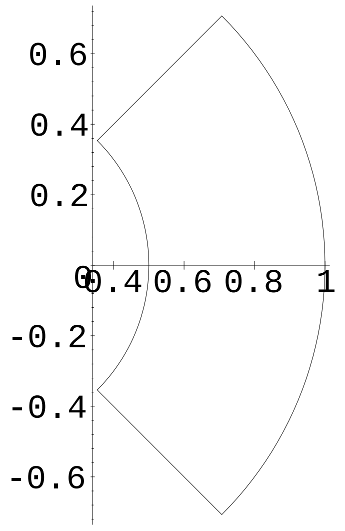

Figure 7.7: A Disk Sector Shaped Domain

```
> seq(seq(unit_normal(Domain,i,End), End=[LOW,HIGH]), i=1..2);
```

$$[\cos(\theta),\sin(\theta)],[-\cos(\theta),-\sin(\theta)],\left[\frac{1}{2}\sqrt{2},\frac{1}{2}\sqrt{2}\right],\left[\frac{1}{2}\sqrt{2},-\frac{1}{2}\sqrt{2}\right]$$

End of Maple Worksheet

Method of Implementation

In the case when `Domain` is of type *RecDomainType*, the procedure simply returns the answer directly. In this case, the unit normal is zero in every dimension except the `DimNum` dimension, where it is plus or minus one. In all other cases, the **unit_normal** first computes a basis of the tangent to the specified boundary and then uses this to compute the vector function that is normal to the boundary.

Computing a tangent to a surface can be easily done if the surface is parameterized. Simply take the derivative of the parameterized form of the surface with respect to one of the parameters. The result is a vector tangent to the surface in the direction along which the parameter increases. The collection of the derivatives with respect to each of the parameters form a basis of the tangent subspace.

For example, assume that the surface can be represented as

$$[y_1(t_1,t_2),y_2(t_1,t_2),y_3(t_1,t_2)].$$

Then the vector

$$\left[\frac{\partial y_1}{\partial t_1},\frac{\partial y_2}{\partial t_1},\frac{\partial y_3}{\partial t_1}\right]$$

is tangent to the surface and oriented in the direction of increasing t_1. Similarly,

$$\left[\frac{\partial y_1}{\partial t_2},\frac{\partial y_2}{\partial t_2},\frac{\partial y_3}{\partial t_2}\right]$$

is tangent to the surface and oriented in the direction of increasing t_2. If each of these vector functions is evaluated at the same point on the surface (and the vectors are imagined to emanate from this point), then the vectors form a basis of the plane tangent to the surface at the point.

Theorem 1 *Given a set of $n-1$ independent vectors in n-space, denoted by b_i. Let the the matrices B_j be the matrices formed by deleting the j^{th}-column from the matrix*

$$\begin{bmatrix} b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}.$$

Then the vector

$$\bar{b} := \left[|B_1|,-|B_2|,\ldots,(-1)^n|B_n|\right]$$

is perpendicular to all of the b_i vectors.

Proof: Consider the dot product of $\bar{b}$ and one of the vectors b_i. The result is

$$\bar{b} \cdot b_i = b_{i,1}\bar{b}_1 + \ldots + b_{i,n}\bar{b}_n.$$

Notice taht this is exactly the determinant of the matrix

$$\begin{bmatrix} b_i \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

which must be zero, since the row b_i appears twice in the matrix. ■

The procedure **unit_normal** uses Theorem 1 and the basis of the tangent to the surface to construct a vector valued function which is perpendicular to the surface at every point on the surface. The determinants are computed by using the procedure **linalg[det]**. This procedure computes determinants by performing row operations until the resulting matrix is triangular and then computing the product of the diagonal elements. The implementation is adequately efficient.

Of course, the final result must be normalized.

Finally, some care must be taken to pick the sign of the vector so that it points in and not out. This is done by checking the `DimNum` component of the result. If `End` equals `LOW` then this component should be positive, if `End` equals `HIGH` then this component should be negative.

Dependencies

This procedure has no direct dependencies on any other module except, of course, the loading and initialization modules.

spline

This procedure returns a member of a particular family of boundary splines.

spline(`Order`: *OrderType*, `End`: *EndType*, `Diriv`: ***nonnegint***)

This form of invocation returns the simplest polynomial spline defined over $[0, 1]$. The value of the spline and all of it derivatives up to the order specified by `Order` are zero at both zero and one, except for the derivative of order `Diriv` at the end specified by `End`. This derivative (or function value when `Diriv` is zero) is equal to 1.

spline(`Order`: *OrderType*, `End`: *EndType*, `Diriv`: ***nonnegint***, `MapInfo`: *MapInfoType*)

This invocation of the procedure specifies a map to be used in constructing a spline. In this case **spline** returns a function (actually a Maple ***procedure***) which is a polynomial in the inverse map of

$$\rho(z) := \frac{\exp(z)}{1+\exp(z)} \tag{7.1}$$

The resulting function is defined over the interval

$$[\phi^{-1}(-\infty), \phi^{-1}(\infty)]$$

and satisfies the same function value and derivative constraints as when the map is not specified. The point of specifying a map is to construct splines which when mapped on the strip about the real line have the same asymptotics as ρ.

It is an error to specify an order greater than zero at an end that corresponds to plus or minus infinity. However, the procedure does not explicitly check for or report this error.

Example Usage

Start of Maple Worksheet

```
> ptolemy[init]();
> spline := 'readlib(`ptolemy/spline`)';
```

$$spline := \mathrm{readlib}(\,ptolemy/spline\,)$$

```
> spline(0, LOW, 0);
```

$$x \to 1 - x$$

```
> spline(0, HIGH, 0);
```

$$x \to x$$

```
> Curves1 := [ seq(spline(1, LOW,i), i=0..1), seq(spline(1, HIGH,i), i=0..1) ];
```

$$Curves1 := \left[x \to (\,2\,x+1\,)\,(\,-1+x\,)^2, x \to x\,(\,-1+x\,)^2, x \to -x^2\,(\,-3+2\,x\,),\right.$$
$$\left. x \to x^2\,(\,-1+x\,)\right]$$

```
> plot(convert(Curves1,set), 0..1, color=WHITE);
> map(N -> map((T,N) -> T(N), Curves1, N), [0,1]);
```

$$[[1,0,0,0],[0,0,1,0]]$$

```
> map(N -> map((T,N) -> unapply(diff(T(x),x),x)(N), Curves1, N), [0,1]);
```

$$[[0,1,0,0],[0,0,0,1]]$$

```
> SubOrder2 := [0,3];
```

$$\mathit{SubOrder2} := [0,3]$$

```
> Map2 := ptolemy[LogRatioMap](-1,1,x,z);
```

$$\mathit{Map2} := \left[x \to \ln\left(\frac{x+1}{1-x}\right), z \to \frac{e^z-1}{1+e^z}, x \to \left(\frac{1}{2}(x+1)\sqrt{2}, \frac{1}{2}(1-x)\sqrt{2}\right)\right]$$

```
> Curves2 := [ spline(SubOrder2, LOW,0, Map2),
>    seq(spline(SubOrder2, HIGH,i, Map2), i=0..3) ];
```

$$\mathit{Curves2} := \left[x \to \frac{1}{16}(-1+x)^4, x \to -\frac{1}{16}(x-3)(x+1)(x^2-2x+5),\right.$$
$$x \to \frac{1}{8}(x+1)(-1+x)(x^2-4x+7),$$
$$\left.x \to -\frac{1}{8}(x+1)(x-3)(-1+x)^2, x \to \frac{1}{12}(x+1)(-1+x)^3\right]$$

```
> plot(convert(Curves2,set), -1..1, color=WHITE, axes=BOXED);
> map(N -> map((T,N) -> T(N), Curves2, N), [-1,1]);
```

$$[[1,0,0,0,0],[0,1,0,0,0]]$$

```
> map(N -> map((T,N) -> unapply(diff(T(x),x$N),x)(1), Curves2, N), [1,2,3]);
```

$$[[0,0,1,0,0],[0,0,0,1,0],[0,0,0,0,1]]$$

```
> Map3 := ptolemy[LogSinhMap](0,LOW, x,z);
```

$$\mathit{Map3} := [x \to \ln(\sinh(x)), z \to \operatorname{arcsinh}(e^z), x \to (\sinh(x), \operatorname{sech}(x))]$$

```
> SubOrder3 := [2,0];
```

$$\mathit{SubOrder3} := [2,0]$$

```
> Curves3 := [ seq(spline(SubOrder3, LOW,i, Map3), i=0..2),
>   spline(SubOrder3, HIGH,0, Map3) ];
```

$$\mathit{Curves3} := \left[x \to \frac{1+3\sinh(x)+3\sinh(x)^2}{(1+\sinh(x))^3}, x \to \frac{\sinh(x)(1+3\sinh(x))}{(1+\sinh(x))^3},\right.$$
$$\left.x \to \frac{1}{2}\frac{\sinh(x)^2}{(1+\sinh(x))^3}, x \to \frac{\sinh(x)^3}{(1+\sinh(x))^3}\right]$$

```
> plot(convert(Curves3,set), 0..14, color=white);
> map(T -> T(0), Curves3);
```

$$[1,0,0,0]$$

```
> map(T -> limit(T(x), x=infinity), Curves3);
```

$$[0,0,0,1]$$

```
> seq(map(T -> unapply(diff(T(x),x$N),x)(0), Curves3, N), N=1..2);
```

$$[0,1,0,0],[0,0,1,0]$$

```
> Map4 := [x -> ln(tan(x*Pi/2)), z -> 2/Pi*arctan(exp(z)),
>    x -> [sqrt(2/Pi)*sin(x*Pi/2), sqrt(2/Pi)*cos(x*Pi/2)]];
```

$$\mathit{Map4} := \left[x \to \ln\left(\tan\left(\frac{1}{2}\,x\,\pi\right)\right), z \to 2\,\frac{\arctan(\,e^z\,)}{\pi},\right.$$
$$\left. x \to \left[\mathrm{sqrt}\left(2\,\frac{1}{\pi}\right)\,\sin\left(\frac{1}{2}\,x\,\pi\right), \mathrm{sqrt}\left(2\,\frac{1}{\pi}\right)\,\cos\left(\frac{1}{2}\,x\,\pi\right)\right]\right]$$

```
> spline(0,LOW,0,Map4); spline(0,HIGH,0,Map4);
```

$$x \to \frac{\cos\left(\frac{1}{2}\,x\,\pi\right)}{\cos\left(\frac{1}{2}\,x\,\pi\right)+\sin\left(\frac{1}{2}\,x\,\pi\right)}$$

$$x \to \frac{\sin\left(\frac{1}{2}\,x\,\pi\right)}{\cos\left(\frac{1}{2}\,x\,\pi\right)+\sin\left(\frac{1}{2}\,x\,\pi\right)}$$

```
> Curves4 := [ seq(spline(1,LOW,i,Map4), i=0..1), seq(spline(1,HIGH,i,Map4), i=0..1) ];
```

$$\mathit{Curves4} := \left[x \to \frac{\cos\left(\frac{1}{2}\,x\,\pi\right)^2\left(\cos\left(\frac{1}{2}\,x\,\pi\right)+3\sin\left(\frac{1}{2}\,x\,\pi\right)\right)}{\left(\cos\left(\frac{1}{2}\,x\,\pi\right)+\sin\left(\frac{1}{2}\,x\,\pi\right)\right)^3},\right.$$
$$x \to 2\,\frac{\sin\left(\frac{1}{2}\,x\,\pi\right)\,\cos\left(\frac{1}{2}\,x\,\pi\right)^2}{\pi\left(\cos\left(\frac{1}{2}\,x\,\pi\right)+\sin\left(\frac{1}{2}\,x\,\pi\right)\right)^3},$$
$$x \to \frac{\sin\left(\frac{1}{2}\,x\,\pi\right)^2\left(3\cos\left(\frac{1}{2}\,x\,\pi\right)+\sin\left(\frac{1}{2}\,x\,\pi\right)\right)}{\left(\cos\left(\frac{1}{2}\,x\,\pi\right)+\sin\left(\frac{1}{2}\,x\,\pi\right)\right)^3},$$
$$\left. x \to -2\,\frac{\sin\left(\frac{1}{2}\,x\,\pi\right)^2\,\cos\left(\frac{1}{2}\,x\,\pi\right)}{\pi\left(\cos\left(\frac{1}{2}\,x\,\pi\right)+\sin\left(\frac{1}{2}\,x\,\pi\right)\right)^3}\right]$$

```
> plot(convert(Curves4,set), 0..1, color=white);
> map(N -> map((T,N) -> T(N), Curves4, N), [0,1]);
```

$$[[1,0,0,0],[0,0,1,0]]$$

```
> map(N -> map((T,N) -> unapply(diff(T(x),x),x)(N), Curves4, N), [0,1]);
```

$$[[0,1,0,0],[0,0,0,1]]$$

End of Maple Worksheet

Method of Implementation

Internally this module is divided into two procedures: the primary procedure **spline** and the procedure **fast_spline**. The primary procedure computes the range of the interval, specifically

$$\ell := \lim_{z\to-\infty} \phi^{-1}(z)$$
$$h := \lim_{z\to\infty} \phi^{-1}(z)$$

and it computes the inverse mapping of ρ,

$$\sigma(x) := \rho(\phi(x))$$

where ρ is defined as in Equation 7.1. Then the primary procedure passes this information to the procedure **fast_spline** to compute the actual result.

Other **PTOLEMY** procedures may call the function **fast_spline** directly in order to obtain greater efficiency when constructing many splines for the same map.

The spline_fast Function This procedure preforms the core operations involved in the result of **spline**.

fast_spline(`FullOrder:` *[nonnegint*, `nonnegint]`, `End:` *EndType*, `Deriv:` *nonnegint*, `RhoInv:` *algebraic*, `Coord:` *name*, `Low:` {*numeric, infinity*}, `High:` {*numeric, infinity*})

In order to understand the algorithm for computing the splines, let n_l be the required order of approximation at the low end, and let n_h be defined in the same way except for the high end. The procedure **fast_spline** starts by constraining the result to be a function of the form

$$f(x) = a_0 + a_1\,\sigma(x) + a_2\,\sigma^2(x) + \ldots + a_{n-1}\,\sigma^{n-1}(x) \tag{7.2}$$

where $n = n_l + n_h + 2$. It then constructs $n-1$ constraints of the form $f^{(k)}(\ell) = 0$ for $k \in \{0..n_l\}$ and $f^{(k)}(h) = 0$ for $k \in \{0..n_h\}$, excluding the derivative of order `Diriv` at the end `End`. Next it augments this system of equations with the constraint $f^{(d)}(v) = 1$ where d equals `Diriv` and v is either ℓ or h depending on the value of `End`. Then **fast_spline** uses Maple's **solve** procedure to compute the values of the a's that satisfy this system. Finally, it plugs these values back into Equation 7.2.

If either ℓ or h is not finite, the function evaluations must be performed in the limit. Also, if a function evaluation results in an error (e.g., division by zero) the result is recomputed as a limit.

Conditioning Concerns

Notice that **fast_spline** does all of this work with integer arithmetic, so the result is accurate even for large values of n. However, the ratio of the largest norm to the smallest norm of s the splines in a family grow exceedingly rapidly as the order of the family increases. As a result even though the splines are accurately computed they may not be useful for approximation when n is large.

For example if $n_l = n_h$ and the map is the log-ratio-map on the interval $[0, 1]$ then the ratio of the largest norm to the smallest norm of the splines in the family grows like $n_l\,!$. This is not a problem for second order systems. However, it could become a problem for fifth order systems.

Dependencies

The procedure **spline** utilizes the procedure **sub_order_expand** from the module ***order_ops***. And the procedure **fast_spline** invokes the procedure indicated by the global variable `ptolemy/SimpProc` on its result.

make_bases

This procedure constructs the collocation of bases required to satisfy the specified approximation order.

make_bases(`Coord`: *list(name)*, `MapInfo`: *list(MapInfoType)*,
`OrderSpec`: *list(OrderSpecType)*, `Var`: *name*, `OrderUsed`: *name*,
`MainBases`: *name*, `ExtraBases`: *name*)

The argument `Coord` specifies the coordinates in the original problem domain (i.e., the unmapped domain). The argument `MapInfo` specifies a list of maps, one per dimension. The argument `OrderSpec` specifies the required order of approximation for each state-variable.

The argument `Var` specifies the name to be used by **make_bases** to return the list of state-variable names. This information is extracted from `OrderSpec`.

The argument `OrderUsed` specifies the name to be used by **make_bases** to return the list of *OrderSpecType*'s actually used in this construction of the bases. The order used will never be less than the order requested, but if the order requested differs on the two ends of a particular dimension and the `MapInfo` provided for that dimension does not specify a factorization of the nullifier into factors corresponding to each end, then the order actually used will be higher than the order requested.

The argument `MainBases` specifies the name to be used by **make_bases** to return a table of the sinc bases. The table will have entries corresponding to each state-variable. The value of an entry will be an ordered list of the bases corresponding to each dimension. The actual sinc bases are the tensor product of each of the dimensional bases.

The sinc-bases returned are the sinc-functions in the mapped-to domain (i.e., the sinc functions composed with the inverse map) times the nullifier raised to the power specified in `OrderUsed`.

The argument `ExtraBases` specifies the name to be used by **make_bases** to return a table of the boundary spline bases. The table will have entries corresponding to each ordered pair of the state-variable and dimension. The value of this entry will be an ordered list of the portion of the extra bases corresponding to that dimension. If the order specification for this state-variable/dimension combination is $[\ell, h]$, then the first ℓ-bases will have nonzero values (or nonzero derivatives) at the low end of the range of this dimension and the last h-bases will have nonzero values at the high end.

The actual boundary spline bases are the tensor product of at least one entry from the table `ExtraBases` and either other sinc-bases or other extra bases.

Example

Start of Maple Worksheet

```
> with(ptolemy, make_bases);
```

$$[\, make\_bases \,]$$

```
> Map1 := ptolemy[LogRatioMap](0,Size,x,z);
```

$$Map1 := \left[x \to \ln\left(\frac{x}{Size - x}\right), z \to \frac{Size\, e^{z}}{1 + e^{z}}, x \to \left(\frac{x}{\sqrt{Size}}, \frac{Size - x}{\sqrt{Size}}\right) \right]$$

```
> readlib(forget): forget(make_bases);
> make_bases([y1,y2], [Map1,Map1], {[V, [1,0]]},
>   'Var', 'OrderUsed', 'MainBases', 'ExtraBases');
> Var;
```

$$[V]$$

```
> eval(MainBases);
```

table([

$$V = \left[\frac{V\_S1\ y1\ (Size - y1)}{Size}, V\_S2\right]$$

])

```
> eval(ExtraBases);
```

table([

$$(V, 1) = \left[\frac{(2\, y1 + Size)\,(Size - y1)^2}{Size^3}, \frac{y1\ (Size - y1)^2}{Size^2}, \frac{y1^2\ (3\, Size - 2\, y1)}{Size^3}, -\frac{y1^2\ (Size - y1)}{Size^2}\right]$$

$$(V, 2) = \left[\frac{Size - y2}{Size}, \frac{y2}{Size}\right]$$

])

```
> Map2 := ptolemy[LogRatioMap](-1,1,x,z);
```

$$Map2 := \left[x \to \ln\left(\frac{x + 1}{1 - x}\right), z \to \frac{e^{z} - 1}{1 + e^{z}}, x \to \left(\frac{1}{2}\,(x + 1)\sqrt{2}, \frac{1}{2}\,(1 - x)\sqrt{2}\right) \right]$$

```
> make_bases([y1,y2], [Map1,Map2], [[U, [1,[0,2]]], [V, [1,0]]],
>   'Var', 'OrderUsed', 'MainBases', 'ExtraBases');
> Var;
```

$$[U, V]$$

```
> eval(OrderUsed);
```

$$[[U, [1, [0, 2]]], [V, [1, 0]]]$$

```
> eval(MainBases);
```

table([

$$U = \left[U\_S1\ y1\ (1 - y1), \frac{1}{2}\ U\_S2\ (1 - y2)^2\right]$$

$$V = [\, V\_S1\ y1\ (1 - y1), V\_S2\,]$$

])

```
> eval(ExtraBases);
```

table([

$$(V,2) = \left[\frac{1}{2} - \frac{1}{2}\,y2, \frac{1}{2}\,y2 + \frac{1}{2}\right]$$

$$(U,1) = \left[(2\,y1+1)\,(-1+y1)^2, y1\,(-1+y1)^2, -y1^2\,(-3+2\,y1), y1^2\,(-1+y1)\right]$$

$$(V,1) = \left[(2\,y1+1)\,(-1+y1)^2, y1\,(-1+y1)^2, -y1^2\,(-3+2\,y1), y1^2\,(-1+y1)\right]$$

$$(U,2) = \left[-\frac{1}{8}\,(-1+y2)^3, \frac{1}{8}\,(y2+1)\,(y2^2-4\,y2+7), -\frac{1}{4}\,(-1+y2)\,(y2-3)\,(y2+1), \frac{1}{4}\,(y2+1)\,(-1+y2)^2\right]$$

])

```
> map(N -> map((T,N) -> unapply(T,y2)(N), ExtraBases[V,2],N), [-1,1]);
```

$$[[1,0],[0,1]]$$

```
> map(N -> map((T,N) -> unapply(T,y2)(N), ExtraBases[U,2],N), [-1,1]);
```

$$[[1,0,0,0],[0,1,0,0]]$$

```
> map(N -> map((T,N) -> unapply(diff(T,y2$N),y2)(1), ExtraBases[U,2],N), [1,2]);
```

$$[[0,0,1,0],[0,0,0,1]]$$

```
> Map3 := [x -> ln(x), z -> exp(z), x -> (x,1)];
```

$$Map3 := [\ln, \exp, x \to (x,1)]$$

```
> MainBases := 'MainBases'; ExtraBases := 'ExtraBases';
```

$$MainBases := MainBases$$

$$ExtraBases := ExtraBases$$

```
> make_bases([x], [Map3], [[V, [[2,0]]]],
>   'Var', 'OrderUsed', 'MainBases', 'ExtraBases');
> eval(ExtraBases);
```

table([

$$(V,1) = \left[\frac{1+3\,x+3\,x^2}{(x+1)^3}, \frac{x\,(1+3\,x)}{(x+1)^3}, \frac{1}{2}\,\frac{x^2}{(x+1)^3}, \frac{x^3}{(x+1)^3}\right]$$

])

```
> map(T -> unapply(T,x)(0), ExtraBases[V,1]);
```

$$[1,0,0,0]$$

```
> map(T -> limit(T,x=infinity), ExtraBases[V,1]);
```

$$[0,0,0,1]$$

```
> map(N -> map((T,N) -> unapply(diff(T,x$N),x)(0), ExtraBases[V,1], N), [1,2]);
```

$$[[0,1,0,0],[0,0,1,0]]$$

```
> Map4 :=
>    [x -> ln(tan(Pi*x/2)), z -> 2*arctan(exp(z))/Pi,
>     x -> (sqrt(2/Pi)*cos(Pi*x/2), sqrt(2/Pi)*sin(Pi*x/2))];
```

$$\mathit{Map4} := \Big[x \to \ln\left(\tan\left(\frac{1}{2}\,\pi\,x\right)\right), z \to 2\,\frac{\arctan(\mathrm{e}^z)}{\pi}, x \to \left(\mathrm{sqrt}\left(2\,\frac{1}{\pi}\right)\,\cos\left(\frac{1}{2}\,\pi\,x\right), \mathrm{sqrt}\left(2\,\frac{1}{\pi}\right)\,\sin\left(\frac{1}{2}\,\pi\,x\right)\right)\Big]$$

```
> MainBases := 'MainBases'; ExtraBases := 'ExtraBases';
```

$$\mathit{MainBases} := \mathit{MainBases}$$

$$\mathit{ExtraBases} := \mathit{ExtraBases}$$

```
> make_bases([x], [Map4], [[V, [[1,2]]]],
>    'Var', 'OrderUsed', 'MainBases', 'ExtraBases');
> Temp := ExtraBases[V,1];
```

$$\mathit{Temp} := \Big[\frac{\cos\left(\frac{1}{2}\,\pi\,x\right)^3\left(\cos\left(\frac{1}{2}\,\pi\,x\right)+4\sin\left(\frac{1}{2}\,\pi\,x\right)\right)}{\%1^4}, 2\,\frac{\sin\left(\frac{1}{2}\,\pi\,x\right)\,\cos\left(\frac{1}{2}\,\pi\,x\right)^3}{\pi\,\%1^4}, \sin\left(\frac{1}{2}\,\pi\,x\right)^2 \left(6\cos\left(\frac{1}{2}\,\pi\,x\right)^2+4\cos\left(\frac{1}{2}\,\pi\,x\right)\,\sin\left(\frac{1}{2}\,\pi\,x\right)+\sin\left(\frac{1}{2}\,\pi\,x\right)^2\right)/\%1^4,$$
$$-2\,\frac{\sin\left(\frac{1}{2}\,\pi\,x\right)^2\,\cos\left(\frac{1}{2}\,\pi\,x\right)\,\left(4\cos\left(\frac{1}{2}\,\pi\,x\right)+\sin\left(\frac{1}{2}\,\pi\,x\right)\right)}{\pi\,\%1^4}, 2\,\frac{\sin\left(\frac{1}{2}\,\pi\,x\right)^2\,\cos\left(\frac{1}{2}\,\pi\,x\right)^2}{\pi^2\,\%1^4}\Big]$$

$$\%1 := \cos\left(\frac{1}{2}\,\pi\,x\right)+\sin\left(\frac{1}{2}\,\pi\,x\right)$$

```
> map(N -> map((T,N) -> unapply(T,x)(N), Temp, N), [0,1]);
```

$$[[1,0,0,0,0],[0,0,1,0,0]]$$

```
> map(N -> map((T,N) -> unapply(diff(T,x), x)(N), Temp, N), [0,1]);
```

$$[[0,1,0,0,0],[0,0,0,1,0]]$$

```
> map(T -> unapply(diff(T,x$2), x)(1), Temp);
```

$$[0,0,0,0,1]$$

```
> Map5 := ptolemy[LogSinhMap](0,2,x,z);
```

$$Map5 := \left[x \to \ln\left(\sinh\left(\frac{1}{2}\,x\right)\right),\, z \to 2\,\mathrm{arcsinh}(\,e^{z}\,),\right.$$
$$\left. x \to \left(\sqrt{2}\sinh\left(\frac{1}{2}\,x\right),\, \sqrt{2}\,\mathrm{sech}\left(\frac{1}{2}\,x\right)\right)\right]$$

```
> make_bases([x], [Map5], [[V, [[1,0]]]],
>   'Var', 'OrderUsed', 'MainBases', 'ExtraBases');
> Temp := ExtraBases[V,1];
```

$$Temp := \left[\frac{1+2\sinh\left(\frac{1}{2}\,x\right)}{\left(1+\sinh\left(\frac{1}{2}\,x\right)\right)^2},\, 2\,\frac{\sinh\left(\frac{1}{2}\,x\right)}{\left(1+\sinh\left(\frac{1}{2}\,x\right)\right)^2},\, \frac{\sinh\left(\frac{1}{2}\,x\right)^2}{\left(1+\sinh\left(\frac{1}{2}\,x\right)\right)^2}\right]$$

```
> map(T -> unapply(T,x)(0), Temp);
```

$$[1, 0, 0]$$

```
> map(T -> limit(T,x=infinity), Temp);
```

$$[0, 0, 1]$$

```
> map(T -> unapply(diff(T,x),x)(0), Temp);
```

$$[0, 1, 0]$$

End of Maple Worksheet

Dependencies

This procedure uses the module ***map_info_ops*** and ***order_ops***, the function **spline**, and the variable ptolemy/SimpProc. The procedure **ptolemy/weight** in the module ***map_info_ops*** is used to compute the minimum nullifier which satisfies the requested accuracy order. The procedure **spline** computes splines with the appropriate boundary conditions. The procedure **sub_order_expand** in the module ***order_ops*** is used when determining how many extra bases must be constructed for each dimension. Finally, the procedure specified by the global variable ptolemy/SimpProc is used to simplify the extra bases.

get_SB_var

This function returns a sequence of all the leaves in the expression tree which are variables containing the substrings '_S' or '_B'.

get_SB_var(`Expression:` *collection(*{*algebraic, equation*}*)*)

Usually the result of this function would be included within a set in order to remove duplicates. The function works equally well with applied or unapplied differential forms.

The function considers each term in isolation so it will even work with equations with some terms applied and some not. However, such usage is highly discouraged.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy, get_SB_var);
                                    [get_SB_var]

> get_SB_var(
>   (D@@2)(V_S1)(phi1(x1))*diff(phi1(x1),x1)^2 * V_S2(x2) +
>   V_B1_1(x1) * D(V_S2)(phi1(x2)) * diff(phi1(x2),x2$2) = 0);
                              V_S1, V_S2, V_B1_1, V_S2

> {"};
                               { V_S1, V_S2, V_B1_1 }

> { get_SB_var((D@@2)(V_S1)*diff(phi1(x1),x1)^2 * V_S2) };
                                    { V_S1, V_S2 }

> get_SB_var(V_S1 + 0.76 = V_S2);
                                      V_S1, V_S2

> get_SB_var(cos(V_S1*V_B1));
                                      V_S1, V_B1
```

End of Maple Worksheet

Method of Implementation

This function recursively calls itself until its argument is of type *algebraic*. It then checks to see if its argument is a simple differential operator by calling **isDop**. If in fact the argument is a `D`-operator the function extracts the variable being operated upon and checks to see if contains either '_S' or '_B'. Otherwise, if the argument is a function the function name is checked. If the function name does not contain an '_S' or an '_B' then **get_SB_var** is recursively called on all of the functions arguments. Differential forms of the type `V_S1(V_S2)` are deemed to be nonsensical in the context of PDEs and therefor the arguments are not checked if the function name matches. If the expression is a ***name*** it is checked directly. Finally, if the expression contains more than one operand **get_SB_var** is called recursively on each of its operands.

Dependencies This function depends on **isDop** to extract the state-variable from `D`-operators.

collocate_main

This procedure converts expressions or equations into SB-notation. This conversion to SB-notation effectively collocates the governing equation.

collocate_bound(Expression: *collection({algebraic, equation})*, Coord: *list(name)*,
NewCoord: *list(name)*, MapInfo: *list(MapInfoType)*,
MainBases: *table*, ExtraBases: *table*, VarCollec: *collection(name)*)

The argument `Expression` specifies the collections of expressions and equations to be converted into SB-notation. The arguments `Coord` and `NewCoord` specify the names of the conditions in the original and mapped domains, respectively. These names are used in the mapping process and should be undefined symbols. The argument `MapInfo` specifies the map to be applied as part of the collocation process. The resulting n-D map is such that each output coordinate depends only on the corresponding input coordinate and this dependence is described by the corresponding entry in `MapInfo`. The arguments `MainBases` and `ExtraBases` specify the main bases and the extra bases. Finally, the argument `VarCollec` specifies the state-variables. The global variable `ptolemy/SimpProc` specifies a simplification procedure that is automatically applied, by **fast_map**, to the final results.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy,collocate_main);
```

$$[\,collocate\_main\,]$$

```
> Map := ptolemy[LogRatioMap](0,1,x,z);
```

$$Map := \left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (\,x, 1-x\,)\right]$$

```
> Maps := [Map, Map];
```

$$Maps := \left[\left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (\,x, 1-x\,)\right],\right.$$
$$\left.\left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (\,x, 1-x\,)\right]\right]$$

```
> BasesOrder := {[V, [1,0]]};
```

$$BasesOrder := \{\,[\,V, [\,1, 0\,]\,]\,\}$$

```
> MainBases[V] := [V_S1*x1*(1-x1), V_S2];
```

$$MainBases_V := [\,V\_S1\ x1\,(\,1 - x1\,),\ V\_S2\,]$$

```
> ExtraBases := table([
>   (V,1)=[(2*x1+1)*(1-x1), x1*(1-x1)^2, x1^2*(3-2*x1), x1^2*(1-x1)],
>   (V,2)=[1-x2,x2]]);
```

$$
\begin{aligned}
&\mathit{ExtraBases} := \mathrm{table}([\\
&\quad (V,1) =\\
&\quad \left[(2\,\mathit{x1}+1)(1-\mathit{x1}), \mathit{x1}\,(1-\mathit{x1})^2, \mathit{x1}^2\,(3-2\,\mathit{x1}), \mathit{x1}^2\,(1-\mathit{x1})\right]\\
&\quad (V,2) = [1-\mathit{x2}, \mathit{x2}]\\
&\quad])
\end{aligned}
$$

```
> collocate_main({}, [x1,x2],[y1,y2], Maps, MainBases,ExtraBases, V);
```

$$\{\,\}$$

```
> Simple1 := [ seq(D[1$i](V), i=0..2) ];
```

$$\mathit{Simple1} := [V, D_1(V), D_{1,1}(V)]$$

```
> collocate_main(Simple1, [x1,x2],[y1,y2], Maps, MainBases,ExtraBases, V);
```

$$
\begin{aligned}
\Bigg[&\mathrm{V\_B1}\left(\frac{3\,\mathrm{e}^{y1}+1}{(1+\mathrm{e}^{y1})^2}, \frac{\mathrm{e}^{y1}}{(1+\mathrm{e}^{y1})^3}, \frac{(\mathrm{e}^{y1})^2\,(3+\mathrm{e}^{y1})}{(1+\mathrm{e}^{y1})^3}, \frac{(\mathrm{e}^{y1})^2}{(1+\mathrm{e}^{y1})^3}\right)\,\mathit{V\_S2}\\
&+ \mathrm{V\_B1}\left(\frac{3\,\mathrm{e}^{y1}+1}{(1+\mathrm{e}^{y1})^2}, \frac{\mathrm{e}^{y1}}{(1+\mathrm{e}^{y1})^3}, \frac{(\mathrm{e}^{y1})^2\,(3+\mathrm{e}^{y1})}{(1+\mathrm{e}^{y1})^3}, \frac{(\mathrm{e}^{y1})^2}{(1+\mathrm{e}^{y1})^3}\right)\,\%1\\
&+ \frac{\mathrm{e}^{y1}\,\mathit{V\_S2}\,\mathit{V\_S1}}{(1+\mathrm{e}^{y1})^2} + \frac{\mathrm{e}^{y1}\,\%1\,\mathit{V\_S1}}{(1+\mathrm{e}^{y1})^2}, -\frac{(-1+\mathrm{e}^{y1})\,\mathit{V\_S2}\,\mathit{V\_S1}}{1+\mathrm{e}^{y1}}\\
&- \frac{(-1+\mathrm{e}^{y1})\,\%1\,\mathit{V\_S1}}{1+\mathrm{e}^{y1}} +\\
&\mathrm{V\_B1}\left(-\frac{-1+3\,\mathrm{e}^{y1}}{1+\mathrm{e}^{y1}}, -\frac{2\,\mathrm{e}^{y1}-1}{(1+\mathrm{e}^{y1})^2}, 6\,\frac{\mathrm{e}^{y1}}{(1+\mathrm{e}^{y1})^2}, -\frac{\mathrm{e}^{y1}\,(-2+\mathrm{e}^{y1})}{(1+\mathrm{e}^{y1})^2}\right)\,\mathit{V\_S2}\\
&+ \mathrm{V\_B1}\left(-\frac{-1+3\,\mathrm{e}^{y1}}{1+\mathrm{e}^{y1}}, -\frac{2\,\mathrm{e}^{y1}-1}{(1+\mathrm{e}^{y1})^2}, 6\,\frac{\mathrm{e}^{y1}}{(1+\mathrm{e}^{y1})^2}, -\frac{\mathrm{e}^{y1}\,(-2+\mathrm{e}^{y1})}{(1+\mathrm{e}^{y1})^2}\right)\,\%1\\
&+ \mathit{V\_S2}\,\mathrm{D}(\mathit{V\_S1}) + \%1\,\mathrm{D}(\mathit{V\_S1}), -2\,\mathit{V\_S2}\,\mathit{V\_S1} - 2\,\%1\,\mathit{V\_S1}\\
&- \frac{(-1+\mathrm{e}^{y1})\,(1+\mathrm{e}^{y1})\,\mathit{V\_S2}\,\mathrm{D}(\mathit{V\_S1})}{\mathrm{e}^{y1}}\\
&- \frac{(-1+\mathrm{e}^{y1})\,(1+\mathrm{e}^{y1})\,\%1\,\mathrm{D}(\mathit{V\_S1})}{\mathrm{e}^{y1}}\\
&+ \mathrm{V\_B1}\left(-4, 2\,\frac{-2+\mathrm{e}^{y1}}{1+\mathrm{e}^{y1}}, -6\,\frac{-1+\mathrm{e}^{y1}}{1+\mathrm{e}^{y1}}, -2\,\frac{2\,\mathrm{e}^{y1}-1}{1+\mathrm{e}^{y1}}\right)\,\mathit{V\_S2}\\
&+ \mathrm{V\_B1}\left(-4, 2\,\frac{-2+\mathrm{e}^{y1}}{1+\mathrm{e}^{y1}}, -6\,\frac{-1+\mathrm{e}^{y1}}{1+\mathrm{e}^{y1}}, -2\,\frac{2\,\mathrm{e}^{y1}-1}{1+\mathrm{e}^{y1}}\right)\,\%1\\
&+ \frac{(1+\mathrm{e}^{y1})^2\,\mathit{V\_S2}\,D^{(2)}(\mathit{V\_S1})}{\mathrm{e}^{y1}} + \frac{(1+\mathrm{e}^{y1})^2\,\%1\,D^{(2)}(\mathit{V\_S1})}{\mathrm{e}^{y1}}\Bigg]\\
&\%1 := \mathrm{V\_B2}\left(\frac{1}{1+\mathrm{e}^{y2}}, \frac{\mathrm{e}^{y2}}{1+\mathrm{e}^{y2}}\right)
\end{aligned}
$$

```
> Simple2 := [ seq(D[2$i](V), i=0..2) ];
```

$$\mathit{Simple2} := [V, D_2(V), D_{2,2}(V)]$$

```
> collocate_main(Simple2, [x1,x2],[y1,y2], Maps, MainBases,ExtraBases, V);
```

$$\left[\%1\,V\_S2 + \%1\,\mathrm{V\_B2}\left(\frac{1}{1+e^{y2}}, \frac{e^{y2}}{1+e^{y2}}\right) + \frac{e^{y1}\,V\_S2\,V\_S1}{(1+e^{y1})^2} + \frac{e^{y1}\,\mathrm{V\_B2}\left(\frac{1}{1+e^{y2}}, \frac{e^{y2}}{1+e^{y2}}\right)V\_S1}{(1+e^{y1})^2}, \%1\,\mathrm{V\_B2}(-1,1) + \frac{e^{y1}(1+e^{y2})^2\,\mathrm{D}(V\_S2)\,V\_S1}{(1+e^{y1})^2\,e^{y2}} + \frac{e^{y1}\,\mathrm{V\_B2}(-1,1)\,V\_S1}{(1+e^{y1})^2} + \frac{\%1\,\mathrm{D}(V\_S2)(1+e^{y2})^2}{e^{y2}}, \frac{\%1\,\mathrm{D}(V\_S2)(e^{y2}-1)(1+e^{y2})^3}{(e^{y2})^2} + \frac{e^{y1}(e^{y2}-1)(1+e^{y2})^3\,\mathrm{D}(V\_S2)\,V\_S1}{(1+e^{y1})^2(e^{y2})^2} + \frac{e^{y1}(1+e^{y2})^4\,D^{(2)}(V\_S2)\,V\_S1}{(1+e^{y1})^2(e^{y2})^2} + \frac{\%1\,D^{(2)}(V\_S2)(1+e^{y2})^4}{(e^{y2})^2}\right]$$

$$\%1 := \mathrm{V\_B1}\left(\frac{3e^{y1}+1}{(1+e^{y1})^2}, \frac{e^{y1}}{(1+e^{y1})^3}, \frac{(e^{y1})^2(3+e^{y1})}{(1+e^{y1})^3}, \frac{(e^{y1})^2}{(1+e^{y1})^3}\right)$$

```
> Laplacian := D[1,1](V) + D[2,2](V) = 0;
```

$$Laplacian := D_{1,1}(V) + D_{2,2}(V) = 0$$

```
> collocate_main(Laplacian, [x1,x2],[y1,y2], Maps,  MainBases, ExtraBases,V);
```

$$-2\,V\_S1\,V\_S2 + \frac{(1+e^{y1})^2\,D^{(2)}(V\_S1)\,V\_S2}{e^{y1}} + \frac{(1+e^{y1})^2\,D^{(2)}(V\_S1)\,\%1}{e^{y1}} + \mathrm{V\_B1}\left(\frac{3e^{y1}+1}{(1+e^{y1})^2}, \frac{e^{y1}}{(1+e^{y1})^3}, \frac{(e^{y1})^2(3+e^{y1})}{(1+e^{y1})^3}, \frac{(e^{y1})^2}{(1+e^{y1})^3}\right)$$
$$D^{(2)}(V\_S2)(1+e^{y2})^4 \Big/ (e^{y2})^2 + \frac{e^{y1}(e^{y2}-1)(1+e^{y2})^3\,\mathrm{D}(V\_S2)\,V\_S1}{(1+e^{y1})^2(e^{y2})^2} - \frac{(e^{y1}-1)(1+e^{y1})\,\mathrm{D}(V\_S1)\,V\_S2}{e^{y1}} + \frac{e^{y1}(1+e^{y2})^4\,D^{(2)}(V\_S2)\,V\_S1}{(1+e^{y1})^2(e^{y2})^2} +$$
$$\mathrm{V\_B1}\left(\frac{3e^{y1}+1}{(1+e^{y1})^2}, \frac{e^{y1}}{(1+e^{y1})^3}, \frac{(e^{y1})^2(3+e^{y1})}{(1+e^{y1})^3}, \frac{(e^{y1})^2}{(1+e^{y1})^3}\right)\mathrm{D}(V\_S2)$$
$$(e^{y2}-1)(1+e^{y2})^3 \Big/ (e^{y2})^2 - \frac{(e^{y1}-1)(1+e^{y1})\,\mathrm{D}(V\_S1)\,\%1}{e^{y1}} + \mathrm{V\_B1}\left(-4, 2\,\frac{-2+e^{y1}}{1+e^{y1}}, -6\,\frac{e^{y1}-1}{1+e^{y1}}, -2\,\frac{-1+2e^{y1}}{1+e^{y1}}\right)V\_S2$$
$$+\mathrm{V\_B1}\left(-4, 2\,\frac{-2+e^{y1}}{1+e^{y1}}, -6\,\frac{e^{y1}-1}{1+e^{y1}}, -2\,\frac{-1+2e^{y1}}{1+e^{y1}}\right)\%1 - 2\,V\_S1\,\%1 = 0$$

$$\%1 := \mathrm{V\_B2}\left(\frac{1}{1+e^{y2}}, \frac{e^{y2}}{1+e^{y2}}\right)$$

End of Maple Worksheet

Method of Implementation

This procedure recurses down the expression tree until it reaches a node that is of type *algebraic*. At that point it constructs arguments to **fast_map** and performs the mapping that is the primary operation for conversion to `SB`-notation. However, the variables corresponding to the boundary splines are left unmapped. This is done for consistency with **collocate_bound**.

The coordinates and maps passed to **fast_map** are constructed directly from the `Coord`, `NewCoord`, and `MapInfo` arguments. The dependencies of the new coordinates on the old (i.e., the `NewCoordDepend` argument to **fast_map**) are always the same, with each new coordinate depending only on the corresponding old coordinate. The `Subs` argument passed to **fast_map** is the sum of all of the bases.

The output state-variable names are simply the names of each of the sinc and boundary basis components. Each output variable depends only on the corresponding coordinate. So that the sinc basis components are mapped but the boundary spline bases components are not, the **fast_map** argument `OutVarDepend` specifies that the boundary spline bases components are dependent on the *input* coordinates and that the sinc bases components are dependent on the *output* coordinates.

This two-step mapping procedure where sinc bases are mapped in one step and boundary splines are mapped in the next step is more complicated than simply mapping all of the output state-variables in one step. It is done this way to allow better code sharing with **bound_collocate**.

Dependencies

This procedure depends on the procedure **fast_map** to do the core of the mapping process. It also uses the procedure **cross_prod** to construct the `Subs` argument of **fast_map**.

It also depends on the procedure **diff_to_pos** in the module ***b_ops*** to map the boundary splines as well as convert their representation to the positional notation.

collocate_bound

This procedure sorts boundary forms according to regions of application and converts them into SB-notation. This conversion to SB-notation is analogous to the task of performing collocation on governing equations, thus the name of the procedure.

collocate_bound(BoundCollec: *collection(BoundFormType)*, Coord: *list(name)*,
NewCoord: *list(name)*, MapInfo: *list(MapInfoType)*,
MainBases: *table*, ExtraBases: *table*,
Ord: *collection(OrderSpecType)*)
collocate_bound(BoundCollec: *collection(BoundFormType)*, Coord: *list(name)*,
NewCoord: *list(name)*, MapInfo: *list(MapInfoType)*,
MainBases: *table*, ExtraBases: *table*,
Ord: *collection(OrderSpecType)*, ONΩ_,, BOUND)

The result of this procedure is a set of lists, where each list corresponds to a boundary region. The first element in each list is a list indicating the region of application; all of the rest of the elements in each list are the converted equations or expression-tag pairs from the boundary forms applied in this region. Of course, the equations or expressions in the output have been converted to SB-notation.

The argument BoundCollec specifies the boundary forms to be converted into SB-notation.

The arguments Coord and NewCoord specify the names of the conditions in the original and mapped domains, respectively. These names are used in the mapping process and should be undefined symbols. The argument MapInfo specifies the map to be applied as part of the collocation process. For the resulting n-D map, each output coordinate depends only on the corresponding input coordinate and this dependence is described by the corresponding entry in MapInfo.

The arguments MainBases and ExtraBases specifies the main bases and each of the extra bases. The argument Ord specify the order of approximation being used. Implicitly this defines the number of extra bases along each boundary.

If the optional 8^{th}-argument is provided, it indicates that the procedure should assume that the extra bases are orthogonal at the points where the boundary forms will be applied. This would be the case if the forms are applied on the boundary (or if the extra bases are specially constructed to be orthogonal at the actual point of application). The result is that many subexpressions in the output are zero, and the result is typically much simpler.

However, for normal application of **PTOLEMY** the extra bases will *not* be orthogonal. So this behavior is *not* the default. However, the extra bases are nearly orthogonal and research-type users may wish to instruct the system to assume that the extra bases are orthogonal even when they are not. The expert user may be able to tell that this simplification reduces complexity enough to offset the increase in error.

The global variable ptolemy/SimpProc specifies a simplification procedure specified to be applied, by **fast_map**, to the final results.

Example Usage

Start of Maple Worksheet

```
> with(ptolemy,collocate_bound);
```

$$[\,collocate\_bound\,]$$

```
> Map := ptolemy[LogRatioMap](0,1,x,z);
```

$$Map := \left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (\,x, 1-x\,)\right]$$

```
> Maps := [Map, Map];
```

$$Maps := \left[\left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (\,x, 1-x\,)\right], \left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (\,x, 1-x\,)\right]\right]$$

```
> BasesOrder := {[V, [1,0]]};
```

$$BasesOrder := \{\,[\,V, [\,1, 0\,]\,]\,\}$$

```
> MainBases[V] := [V_S1*x1*(1-x1), V_S2];
```

$$MainBases_V := [\,V\_S1\ x1\,(\,1 - x1\,),\, V\_S2\,]$$

```
> ExtraBases := table([
>   (V,1)=[(2*x1+1)*(1-x1)^2, x1*(1-x1)^2, x1^2*(3-2*x1), x1^2*(1-x1)],
>   (V,2)=[1-x2,x2]]);
```

$$ExtraBases := \text{table}([$$
$$(\,V, 1\,) =$$
$$\left[(\,2\,x1 + 1\,)\,(\,1 - x1\,)^2, x1\,(\,1 - x1\,)^2, x1^2\,(\,3 - 2\,x1\,), x1^2\,(\,1 - x1\,)\right]$$
$$(\,V, 2\,) = [\,1 - x2, x2\,]$$
$$])$$

```
> collocate_bound({}, [x1,x2],[y1,y2], Maps, MainBases,ExtraBases, BasesOrder);
```

$$\{\,\}$$

```
> Bound1 := {[1,LOW, V, TagA], [1,LOW, D[1](V), TagB],
>   [2,LOW, V = cos(Pi*x1)], [2,HIGH, V = sin(Pi*x1)]};
```

$$Bound1 := \Big\{[\,1, LOW, V, TagA\,], [\,1, LOW, D_1(\,V\,), TagB\,], [\,2, LOW, V = \cos(\,\pi\,x1\,)\,], [\,2, HIGH, V = \sin(\,\pi\,x1\,)\,]\Big\}$$

```
> collocate_bound(Bound1, [x1,x2],[y1,y2], Maps, MainBases,ExtraBases,BasesOrder);
```

$$\Big\{[\,[\,\%3\ V\_S2 + \%3\ \%1, TagA\,], V, [\,-1, 0\,]\,],$$
$$\left[\%3\ \%1 = \cos\left(\frac{\pi\, e^{y1}}{1+e^{y1}}\right), V, [\,2, -1\,]\right],$$

$$\left[\frac{e^{y1}\,\%1\,V\_S1}{(1+e^{y1})^2}+\%3\,\%1=\cos\left(\frac{\pi\,e^{y1}}{1+e^{y1}}\right),V,[0,-1]\right],$$
$$\left[\frac{e^{y1}\,\%1\,V\_S1}{(1+e^{y1})^2}+\%3\,\%1=\sin\left(\frac{\pi\,e^{y1}}{1+e^{y1}}\right),V,[0,1]\right],$$
$$\left[\%3\,\%1=\sin\left(\frac{\pi\,e^{y1}}{1+e^{y1}}\right),V,[-1,1]\right],[[\%2\,\%1,\,TagB],V,[-2,1]],$$
$$[[\%3\,\%1,\,TagA],V,[-1,1]],[[\%3\,\%1,\,TagA],V,[-1,-1]],$$
$$[[\%2\,\%1,\,TagB],V,[-2,-1]],\left[\%3\,\%1=\sin\left(\frac{\pi\,e^{y1}}{1+e^{y1}}\right),V,[2,1]\right],$$
$$\left[\%3\,\%1=\cos\left(\frac{\pi\,e^{y1}}{1+e^{y1}}\right),V,[-1,-1]\right],$$
$$[[\%2\,V\_S2+\%2\,\%1,\,TagB],V,[-2,0]]\Big\}$$
$$\%1:=\mathrm{V\_B2}\left(\frac{1}{1+e^{y2}},\frac{e^{y2}}{1+e^{y2}}\right)$$
$$\%2:=\mathrm{V\_B1}\left(-6\frac{e^{y1}}{(1+e^{y1})^2},-\frac{2\,e^{y1}-1}{(1+e^{y1})^2},6\frac{e^{y1}}{(1+e^{y1})^2},-\frac{e^{y1}\,(-2+e^{y1})}{(1+e^{y1})^2}\right)$$
$$\%3:=\mathrm{V\_B1}\left(\frac{3\,e^{y1}+1}{(1+e^{y1})^3},\frac{e^{y1}}{(1+e^{y1})^3},\frac{(e^{y1})^2\,(3+e^{y1})}{(1+e^{y1})^3},\frac{(e^{y1})^2}{(1+e^{y1})^3}\right)$$

```
> Bound2 := {[1,LOW, D[1](V) + V = sin(x2)], [1,HIGH, D[2](V) + D[1](V), TagC]};
```

$$Bound2:=\Big\{$$
$$[1,LOW,D_1(V)+V=\sin(x2)],[1,HIGH,D_2(V)+D_1(V),TagC]\Big\}$$

```
> collocate_bound(Bound2, [x1,x2],[y1,y2], Maps,
>   MainBases, ExtraBases,BasesOrder,ON_BOUND);
```

$$\Big\{[\mathrm{V\_B1}(0,1,0,0)\,\mathrm{V\_B2}(1,0)+\mathrm{V\_B1}(1,0,0,0)\,\mathrm{V\_B2}(1,0)=0,V,$$
$$[-2,-1]],\Big[\mathrm{V\_B1}(0,1,0,0)\;V\_S2$$
$$+\mathrm{V\_B1}(0,1,0,0)\,\mathrm{V\_B2}\left(\frac{1}{1+e^{y2}},\frac{e^{y2}}{1+e^{y2}}\right)+\mathrm{V\_B1}(1,0,0,0)\;V\_S2$$
$$+\mathrm{V\_B1}(1,0,0,0)\,\mathrm{V\_B2}\left(\frac{1}{1+e^{y2}},\frac{e^{y2}}{1+e^{y2}}\right)=\sin\left(\frac{e^{y2}}{1+e^{y2}}\right),V,$$
$$[-2,0]\Big],[[$$
$$\mathrm{V\_B1}(0,0,1,0)\,\mathrm{V\_B2}(-1,1)+\mathrm{V\_B1}(0,0,0,1)\,\mathrm{V\_B2}(0,1),\,TagC$$
$$],V,[2,1]],[$$
$$\mathrm{V\_B1}(0,1,0,0)\,\mathrm{V\_B2}(0,1)+\mathrm{V\_B1}(1,0,0,0)\,\mathrm{V\_B2}(0,1)=$$
$$\sin(1),V,[-2,1]],\Big[\Big[\frac{\mathrm{V\_B1}(0,0,1,0)\,\mathrm{D}(V\_S2)\,(1+e^{y2})^2}{e^{y2}}$$
$$+\mathrm{V\_B1}(0,0,1,0)\,\mathrm{V\_B2}(-1,1)+\mathrm{V\_B1}(0,0,0,1)\;V\_S2$$
$$+\mathrm{V\_B1}(0,0,0,1)\,\mathrm{V\_B2}\left(\frac{1}{1+e^{y2}},\frac{e^{y2}}{1+e^{y2}}\right),\,TagC\Big],V,[2,0]\Big],[[$$

$$\mathrm{V\_B1}(0,0,1,0)\,\mathrm{V\_B2}(-1,1) + \mathrm{V\_B1}(0,0,0,1)\,\mathrm{V\_B2}(1,0),\ \mathit{TagC}$$
$$],V,[2,-1]]\Big\}$$

End of Maple Worksheet

Method of Implementation

The procedure makes several passes through the collection of boundary forms. On its first pass it determines which dimensions are used by any of the boundary forms, groups all the boundary constraints according to the boundary on which they are applied, and determines for each dimension which ends have boundary forms applied to them.

Then the procedure uses this information to construct a list of all the regions for which results will need to be reported. These regions are not specified in list of $\{-1, 0, 1\}$ notation for the final output of this routine. Rather the result is specified as a list of boundaries of the form `[Dim,End]`. This allows for easy referencing of the input boundary forms associated with each region. For example, if for a two-dimentional domain the boundary forms are applied at `[1,HIGH]` and `[2,LOW]` then the list would be of the form

```
[[[1,LOW]], [2,LOW], [[1,LOW],[2,LOW]],
  [[1,HIGH],[2,LOW]], [[1,HIGH],[2,HIGH]]]
```

Figure 7.8 illustrates this example.

This set of affected regions is constructed creating a set of all of the boundaries affected, i.e., a set with entries of the form `[Dim,End]`. The set structure is used to eliminate any duplicates. Then for each affected boundary the procedure **cross_prod** is called to construct a set of all of the regions in that boundary. These sets are then coalesced into one set, which has the effect of removing all duplicates.

The procedure then proceeds to perform the collocation one region at a time. This is done primarily by evoking the procedure **fast_map** on each boundary form occurring in any group of boundary forms which contains the region. Finally, the procedure **diff_to_pos** (from the ***b_ops*** module) is called to convert the boundary variables into positional notation. If the optional 8th argument is specified, this conversion will be done as though the boundary forms are exactly on the boundary instead of merely close to the boundary.

Quite a bit of extra code is required to ensure that **fast_map** and **diff_to_pos** do not operate on the tags of form derived from *BoundTagTypes*. Although **PTOLEMY** code never creates tags that would be altered by these operations, the tag may be any Maple object and it is important to preserve them unaltered in the output.

It might seem that this is more complicated than simply collocating the equations for each boundary and then grouping the results by regions. It is. However, boundary forms cannot be collocated the same across all regions on the same boundary. It is necessary to construct a different collocation for each regions in which a single boundary form is applied.

The primary reason for this is to allow for important symbolic simplifications. Specifically in each boundary region some of the bases will be zero at all of the collocation points. It is not strictly wrong to represent all of these bases in the final result; they

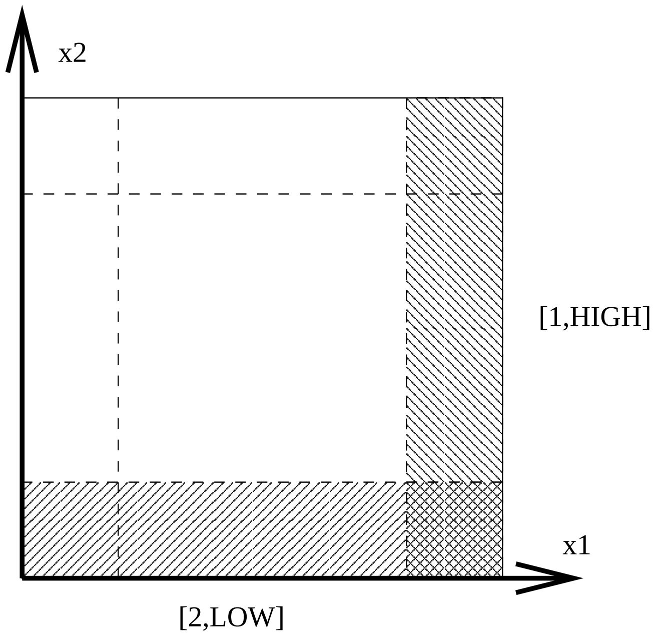

Figure 7.8: An Example of Two Boundaries Affecting Five Regions

will simply have zero contribution to the final matrix. However, the added complexity is prohibitive. There are many more boundary regions than domains in most higher-dimensional problems and often many more boundary forms per boundary region than governing equations per domain, so simplifying the boundary constraints is critical.

The fast_map Call The coordinates and map passed to **fast_map** are constructed directly from the `Coord`, `NewCoord`, and `MapInfo` arguments. The dependencies of the new coordinates on the old (i.e., the `NewCoordDepend` argument to **fast_map**) are always the same, specifically each new coordinate depends only on the corresponding old coordinate.

The substitution performed by the procedure **fast_map** is the sum of all of the bases which are nonzero in the current region. Symbolically, this is constructed by determining which one-dimensional bases components are nonzero in each dimension and then calling **cross_prod** to construct a list of all nonzero n-dimensional bases. A one-dimensional bases component is nonzero in the region if it is a boundary spline or if it is a sinc basis that is not perpendicular to one of the boundaries of the region.

An example may clarify the previous paragraph. For a two-dimensional problem in region `[-1,-1]`, `[-1,1]`, `[1,-1]`, and `[1,1]` a sample substitution would be

`V` → `V_B1*V_B2`.

For the regions `[-1,0]` and `[1,0]` a sample substitution might be

`V` → `V_B1*V_B2 + V_B1*V_S2`.

Similarly for the regions `[0,-1]` and `[0,1]` a sample substitution might be

`V -> V_B1*V_B2 + V_S1*V_B2.`

In contrast a sample substitution involving all of the bases would be

`V -> V_B1*V_B2 + V_B1*V_S2 + V_S1*V_B2 + V_S1*V_S2.`

The output state-variables are merely the set of bases components used in the substitution. Each output state-variable depends only on the one coordinate in the dimension for which it is a bases component. However, it is necessary at this stage to only map the output variables which correspond to sinc bases components, since the output state-variables corresponding to boundary splines will undergo mapping as part of the conversion to positional notation.

This two-step mapping procedure where sinc bases are mapped in one-step and boundary splines are mapped in the next step is more complicated than simply mapping all of the output state-variables in one step. The advandate is that it affords some extra efficiency when the boundary forms will be applied exactly on the boundary. Specifically, mapping of some boundary splines may be avoided in this case. Unlike the efficiency gained by using only the nonzero subset of the bases, this is not a compelling gain in efficiency.

It is not clear that this efficiency gain justifies the added complexity. Other alternatives would require redesigning the **b_ops** LLF.

Undetected Errors

It is an error to have a boundary constraint that involves a derivative of higher order than the corresponding order of the bases. Detection of this error is a fair amount of work. Since this procedure is intended as a low level support routine it is assumed that the higher level code which invoked this procedure will have ensured that the arguments do not cause this error. As a result, rechecking for this error would be wasteful.

However, users who call this procedure directly should be careful when assigning the meaning to the result.

Dependencies

This procedure depends on the procedure **fast_map** to do the core of the mapping process. It also uses the procedure **cross_prod** to construct the `Subs` argument when calling **fast_map**.

It also depends on the procedure **diff_to_pos** in the module ***b_ops*** to map the boundary splines as well as convert their representation to the positional notation.

assign_bound

This procedure assigns boundary constraints to collocation points.

Any assignment that does not assign two constraints to the same point is algebraically correct, but some assignments yield less approximation error than others. This is because the boundary conditions are being applied at collocation points a small distance from the boundary and some of the constraints are more sensitive to this approximation than others.

This procedure sorts the constraints according to the order of their sensitivity with respect to the point of application and assigns the most sensitive constraints to collocation points nearest the boundary. Because only the order of the sensitivity is used, instead of the exact sensitivity, and because directional sensitivities are not distinguished in corner regions, the result is *not* the assignment that minimizes the error caused by collocating boundary conditions a small distance from the boundary. This algorithm tends to avoid assignments that cause particularly large errors.

Typically, the error caused by applying boundary conditions a small distance from the boundary will be of the order of the approximation error for the PDE, so minimizing (or even eliminating) this source of error will not significantly increase the accuracy of the overall solution. All that is needed is to ensure that this source of error is not unacceptably large.

assign_bound(`Region`: *list(integer)*,
`Constraint`: *collection({equation, [algebraic, anything]})*,
`Order`: *collection(OrderSpecType)*)

The argument `Region` specifies in which boundary region the constraints will be applied. Each dimension of the domain is divided into three parts, i.e., low-end extra collocation points, main collocation point, and high-end extra collocation points. The result is that an n-dimensional domain is divided into 3^n regions. Each region is specified by indicating for each dimension in which of the three divisions the region occurs. The three divisions are represented by -1, 0, and 1, respectively. Figure 7.9 illustrates how each region would be specified for a two-dimentional domain.

The argument `Constraint` specifies all of the constraints to be assigned collocation points in this region. The procedure **assign_bound** is not designed to be called with some of the constraints and then later called with additional constraints.

The result of this procedure will be a set of override directives. If the constraint is of the type *equation* the corresponding override directive will be an *OverRideType*. However, if the constraint is a two element list, the override directive will be like an *OverRideType*, except that instead of an equation in the first element will be the constraint exactly as provided. The functionality is provided for override directives that will be converted into *OverCoupleType*s by the caller. The second element in such constraints is to be used to label the constraint so that it may later be identified. Within the context of normal **PTOLEMY** usage this identification would be done to matched two halves of a coupling equation.

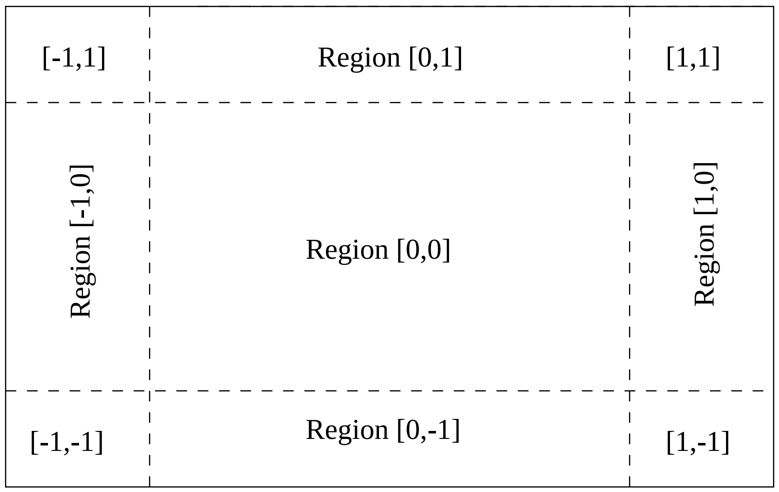

Figure 7.9: Each Region and its Identifying List for a Two-Dimensional Domain

Example

Start of Maple Worksheet

```
> with(ptolemy,assign_bound);
```

$$[\, assign\_bound \,]$$

```
> Bound := [V_B1 * V_S2 = K1, D(V_B1) * V_S2 = K2];
```

$$Bound := [\, V\_B1\ V\_S2 = K1, \mathrm{D}(\ V\_B1\ )\ V\_S2 = K2\,]$$

```
> assign_bound([-1,0], Bound, [V,[1,0]]);
```

$$\{\,[\,\mathrm{D}(\ V\_B1\ )\ V\_S2 = K2, V, [\,-2, 0\,]\,], [\ V\_B1\ V\_S2 = K1, V, [\,-1, 0\,]\,]\,\}$$

```
> Bound := [V_B1*V_S2 = K1, D(V_B1)*V_S2 + U_B1*U_S1= K2, U_B1*U_S2 = K3];
```

$$Bound := [V\_B1\ V\_S2 = K1, \mathrm{D}(\ V\_B1\ )\ V\_S2 + U\_B1\ U\_S1 = K2,\\ U\_B1\ U\_S2 = K3]$$

```
> assign_bound([1,0], Bound, {[V,[1,0]],[U,[0,0]]});
```

$$\{[\ U\_B1\ U\_S2 = K3, V, [\,2, 0\,]\,],\\ [\,\mathrm{D}(\ V\_B1\ )\ V\_S2 + U\_B1\ U\_S1 = K2, U, [\,1, 0\,]\,],\\ [\ V\_B1\ V\_S2 = K1, V, [\,1, 0\,]\,]\}$$

```
> Bound := [V_S1 * V_B2 = K1, D(V_B1) * V_S2 = K2];
```

$$Bound := [\ V\_S1\ V\_B2 = K1, \mathrm{D}(\ V\_B1\ )\ V\_S2 = K2\,]$$

```
> assign_bound([-1,1], Bound, [V,[1,1]]);
```

$$\{[\mathrm{D}(\ V\_B1\ )\ V\_S2 = K2, V, [\,-2, 0, 0, 2\,]\,],$$

$$[\,V\_S1\ V\_B2 = K1, V, [\,-2, 0, 0, 1\,]\,]\}$$

```
> Bound := [V_B1*V_B2 = K1, D(V_B1)*V_B2 + U_B1*U_B2= K2, U_B1*U_B2 = K3];
```

$$Bound := [\,V\_B1\ V\_B2 = K1, \mathrm{D}(\ V\_B1\ )\ V\_B2 + U\_B1\ U\_B2 = K2,\ U\_B1\ U\_B2 = K3\,]$$

```
> assign_bound([-1,1], Bound, {[V,[1,0]],[U,[0,0]]});
```

$$\{[\ U\_B1\ U\_B2 = K3, V, [\,-2, 0, 0, 1\,]\,],\ [\,\mathrm{D}(\ V\_B1\ )\ V\_B2 + U\_B1\ U\_B2 = K2, U, [\,-1, 0, 0, 1\,]\,],\ [\ V\_B1\ V\_B2 = K1, V, [\,-1, 0, 0, 1\,]\,]\}$$

```
> Bound := [V_B1*V_B2 = K1, V_B1*V_B2 = K2];
```

$$Bound := [\ V\_B1\ V\_B2 = K1,\ V\_B1\ V\_B2 = K2\ ]$$

```
> assign_bound([1,1], Bound, [V,[0,0]]);
```

$$\{\,[\ V\_B1\ V\_B2 = K2, V, [\,1, 0, 0, 1\,]\,], [\ V\_B1\ V\_B2 = K1, V, [\,1, 0, 0, 1\,]\,]\,\}$$

```
> Bound := [V_B1*V_B2 = K, V_B1*V_B2 = K];
```

$$Bound := [\ V\_B1\ V\_B2 = K,\ V\_B1\ V\_B2 = K\ ]$$

```
> assign_bound([1,1], Bound, [V,[0,0]]);
```

$$\{\,[\ V\_B1\ V\_B2 = K, V, [\,1, 0, 0, 1\,]\,]\,\}$$

```
> Bound1 := [[V_B1*V_S2, `Label A`], [D(V_B1)*V_S2, `Label B`]];
```

$$Bound1 := [[\ V\_B1\ V\_S2, Label\ A\,], [\,\mathrm{D}(\ V\_B1\ )\ V\_S2, Label\ B\,]]$$

```
> Bound2 := [[D(V_B1)*V_S2, `Label A`], [V_B1*V_S2, `Label B`]];
```

$$Bound2 := [[\,\mathrm{D}(\ V\_B1\ )\ V\_S2, Label\ A\,], [\ V\_B1\ V\_S2, Label\ B\,]]$$

```
> Result1 := assign_bound([1,0], Bound1, [V,[0,1]]);
```

$$Result1 := \{[[\,\mathrm{D}(\ V\_B1\ )\ V\_S2, Label\ B\,], V, [\,1, 0\,]\,],\ [[\ V\_B1\ V\_S2, Label\ A\,], V, [\,1, 0\,]\,]\}$$

```
> Result2 := assign_bound([1,0], Bound2, [V,[0,1]]);
```

$$Result2 := \{[[\,\mathrm{D}(\ V\_B1\ )\ V\_S2, Label\ A\,], V, [\,1, 0\,]\,],\ [[\ V\_B1\ V\_S2, Label\ B\,], V, [\,1, 0\,]\,]\}$$

```
> map(x -> x[1][2] = [x[1][1], op(2..3,x)], Result1);
```

$$\{\,Label\ A = [\ V\_B1\ V\_S2, V, [\,1, 0\,]\,], Label\ B = [\,\mathrm{D}(\ V\_B1\ )\ V\_S2, V, [\,1, 0\,]\,]\ \}$$

```
> LHS :=
>   subs(map(x -> x[1][2] = [A,x[1][1], op(2..3,x)], Result1),
>     [`Label A`, `Label B`]);
```

$$LHS := [[\,A,\ V\_B1\ V\_S2, V, [\,1, 0\,]\,], [\,A, \mathrm{D}(\ V\_B1\ )\ V\_S2, V, [\,1, 0\,]\,]\,]$$

```
> RHS :=
>   subs(map(x -> x[1][2] = [B,x[1][1], op(2..3,x)], Result2),
>     [`Label A`, `Label B`]);
```

$$RHS := [[B, \mathrm{D}(\ V\_B1\ )\ V\_S2, V, [1,0]], [B,\ V\_B1\ V\_S2, V, [1,0]]]$$

```
> seq(LHS[i] = RHS[i], i=1..2);
```

$$[A,\ V\_B1\ V\_S2, V, [1,0]] = [B, \mathrm{D}(\ V\_B1\ )\ V\_S2, V, [1,0]],$$
$$[A, \mathrm{D}(\ V\_B1\ )\ V\_S2, V, [1,0]] = [B,\ V\_B1\ V\_S2, V, [1,0]]$$

End of Maple Worksheet

Method of Implementation

As stated in the introduction to this section any assignment of constraints to collocation points is valid as long as no collocation point is assigned more than one constraint. However, applying boundary constraints at interior points instead of on the boundary introduces a source of error other than the fundamental error of approximation.

For any assignment this source of error will decrease with the same order as the error of approximation. Typically the range of this error over the set of possible assignments will include the inherent error of approximation. So it is important to pick a good assignment, but if this source of error is less than the inherent error of approximation there is little incentive to find a better approximation.

For a given constraint the error caused by applying it at a collocation point instead of on the boundary will be larger for collocation points further from the boundary. If the constraint is of the form

$$\mathcal{L}(V)(x) = k$$

then the error introduced by applying the constraint off the boundary is proportional to $|\mathcal{L}(V)(x_c) - \mathcal{L}(v)(x_b)|$ where x_c is the collocation point and x_b is the projection of x_c onto the boundary (or boundaries). So it is desirable to assign the constraints that change the most rapidly near the boundary to the points that are actually nearest to the boundary.

An ideal approach might be to compute $|\mathcal{L}(V)(x_c) - \mathcal{L}(V)(x_b)|$ for every combination of constraint and admissible collocation point and then employ a selection criterion that either minimized the bound on the total error or at least approximately minimized the bound on the total error. As this point in the problem solution process where **assign_bound** is invoked the number of grid points to be used, i.e., the values of N, are not known so it is not possible to compute the error associated with any particular assignment.

However, the exponential order at which $|\mathcal{L}(V)(x_c) - \mathcal{L}(V)(x_b)|$ decreases (as a function of N) will be equal to the order of the first nonzero derivative of $\mathcal{L}(V)$ at the boundary in the direction $x_b - x_c$. It would be reasonable then to compute the series of $\mathcal{L}(V)$ with respect to the appropriate coordinate and determine the first nonzero derivative.

For linear problems this approach is equivalent to computing the first nonzero derivative of each of the terms in $\mathcal{L}(V)$ and using the minimum. In this case $\mathcal{L}(V)$ will be made up of linear combinations of terms of the form

```
(D@@p0)(S_V0) *···* (D@@pj)(B_Vj) *···* (D@@pn)(S_Vn)
```

and checking for the first nonzero derivative of such terms is particularly easy.

Consider any of the boundary spline, which can be easily determined. As N becomes large those constraints with boundary splines that have a small exponential order of change near the boundary will eventually dominate the error due to applying boundary constraints off the boundary. So a reasonable alternate strategy is to assign these constraints to the collocation points nearest the boundary.

This task is made easier by the fact that x_c and x_b only differ in the direction (or directions) perpendicular to the boundary.

This is approximated by sorting the equations according to the order of their variability near the corner or edge, sorting the points according to their distance from the corner, and then pairing the constraints and points in this order. The result is not optimal, but since for large N this source of error tends to be dominated by a few (often one) more sensitive constraint, simply assigning these constraints to the points nearest the boundary yields an almost optimal assignment.

Design Limitations

This procedure assigns individual equations to individual collocation points. The idea that this is a useful thing to do is based on several assumptions about the way **PTOLEMY** works. It would be reasonable to design a sinc-collocation system which applied more than one collocation constraint at a single collocation point, but it would require nontrivial changes to the much of **PTOLEMY**.

Even within the context of **PTOLEMY**'s design this procedure assumes that in each region there are more points than equations and that the system of equations is well-defined. Since the number of boundary bases in each region is the same as the number of collocation points, this assumption is often justified.

However, it is possible to have an overconstrained system of boundary constraints in a corner region even when the original problem is properly constrained. Consider a problem defined over the region $[0,1] \times [0,1]$ with Dirichalay conditions specified on both the upper and the right boundaries. Suppose that the boundary condition are

$$f(x_1, 1) = g(x_1) := \frac{1+\sqrt{1-x_1}}{2}$$
$$f(1, x_2) = h(x_2) := \frac{1-\sqrt{1-x_2}}{2}.$$

This is a well-constrained PDE since $g(1) = 1/2 = h(1)$. (Figure 7.10 illustrates these boundary conditions). When the order of the extra bases is $[0,0]$, then in the region $(1,1)$ there is exactly one basis, namely $x_1 \cdot x_2$ or, in SB-notation `V_B12_4`. However, two constraints will arise in this region, one from each edge. Specifically, `V_B12_4 = g(x1)` and `V_B12_4 = h(x2)`. Thus, there are more constraints than bases.

This problem is uncommon when the order of the extra bases is greater than 1. However, a new problem arises then. Since $g(x_1) \neq h(x_2)$ except at the corner, these two constraints are inconsistent. Even if constant terms (i.e., terms not involving state-variables) are evaluated at the boundary, this is a system of consistent, but not independent, constraints.

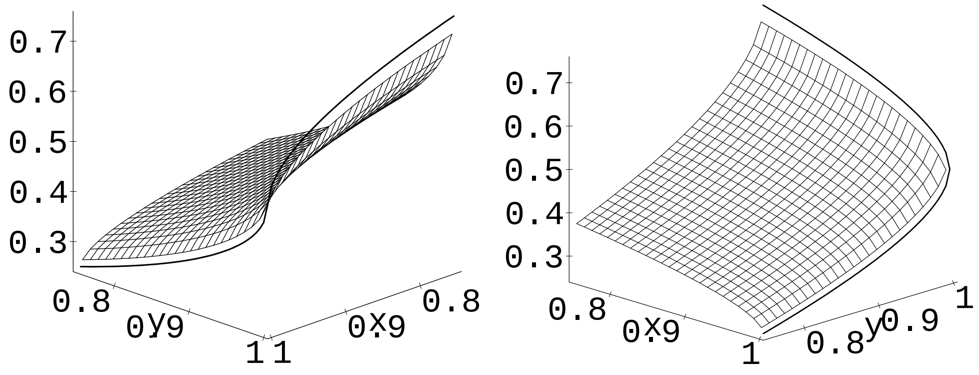

Figure 7.10: Dirchalay Boundary Conditions on Adjoining Edges

Probably the most desirable solution would be to develop a new module, perhaps called **resolve_bound**, which replaces a system of constraints with the "nearest" well-conditioned system. The method could be a variation on using Singular Value Decomposition (SVD) on the original system to generate a new system that is the inverse of the generalized inverse of the original system. The most difficult issue here is handling symbolic constants and coefficients that vary along the boundary.

At the moment **PTOLEMY** provides the user with a chance to edit the boundary constraints manually. When a problem arises, the user can typically achieve nearly optimal results by removing one of the redundant constraints in the affected corner region.

Dependencies

This module uses the procedure **ptolemy/sub_order_expand** from the ***order_ops*** module and the procedure **cross_prod** to construct a list of all of the collocation points in the specified region. This module also uses the procedure **ptolemy/de_order** out of the module ***pde_order*** to determine which boundary splines occur in each constraint.

b_ops

This module provides procedures for manipulating the B-variable constructs used to represent the boundary spline bases components.

b_point_to_num(`Point`: *list(integer)*, `Ord`: *list(SubOrderType)*, `BoundSet`: *name*)
b_num_to_point(`Num`: *posint*, `BoundSet`: *list(posint)*, `Ord`: *list(SubOrderType)*)

In the construction of *LinKronType*'s, bases corresponding to specific sets of collocation points are assigned names based on the set of bases components that are boundary splines and the sequence number within the group of bases having the same set of boundary spline components. The sequence numbers are assigned in order, assuming that the last index varies most rapidly.

The procedure **b_point_to_num** and **b_num_to_point** provide utilities for converting between this type of specification of boundary points and the cartesian coordinate method used by the collocation routines. The procedure **b_point_to_num** requires a cartesian type specification and order information and returns the sequence number and the set of boundary components. The set of boundary components is assigned to the variable specified by `BoundSet`. The procedure **b_num_to_point** requires the sequence number, the set of boundary components, and order information and returns a cartesian type point specification.

collocate_Bvar(`Exp`: *collection({algebraic, equation})*, `BName`: *collection(name)*,
`Coord`: *list(name)*, `NewCoord`: *list(name)*, `MapInfo`: *list(MapInfoType)*,

`ExtraBases`: *table*)
collocate_Bvar(`Exp`: *collection({algebraic, equation})*, `BName`: *collection(name)*,
`Coord`: *list(name)*, `NewCoord`: *list(name)*, `MapInfo`: *list(MapInfoType)*,

`ExtraBases`: *table*, `OrderSpec`: *collection(OrderSpectype)*,
`Region`: *list(integer)*)

The procedure **collocate_Bvar** collocates any B-variables in the argument `Exp` *and converts the variables into the positional notation.*

The argument `BName` specifies all of the boundary spline bases component names. These are in effect the state-variables of the mapping associated with collocation. The arguments `Coord` and `NewCoord` specify the coordinate names in the original domain and the mapped-to domain. The argument `MapInfo` specifies the map associated with the collocation process. Finally, the argument `ExtraBases` provides a table for looking up the boundary spline bases components. The table is expect to have entries for indexes of the form `[V,Dim]` where `V` ranges over the set of *problem* state-variables (i.e., not the set of B-variables) and `Dim` ranges over the set of problem dimensions. This is the form of table returned by **make_base**.

If the optional arguments `OrderSpec` and `Region` are specified then collocation is performed on the boundary corresponding to the region specified by `Region`. Collocation on the boundary generally causes some simplification of the results, because some terms are constant on the boundary. The simplification is especially pronounced when some of the weighting terms are zero on the boundary.

Examples

The first example illustrates how to convert between sequence number specifications and cartesian coordinate specifications. Referring to Figure 7.11 should help with understanding this example.

Start of Maple Worksheet

```
> ptolemy[init]();
> readlib(`ptolemy/b_ops`):
```

```
> Ord := [[1,0], [0,1]];
```

$$Ord := [[1,0],[0,1]]$$

```
> `ptolemy/b_point_to_num`([-2,0], Ord, 'Bound'); Bound;
```

$$1$$

$$[1]$$

```
> `ptolemy/b_point_to_num`([-1,0], Ord, 'Bound');
```

$$2$$

```
> `ptolemy/b_point_to_num`([1,0], Ord, 'Bound');
```

$$3$$

```
> `ptolemy/b_point_to_num`([0,-1], Ord, 'Bound'); Bound;
```

$$1$$

$$[2]$$

```
> `ptolemy/b_point_to_num`([0,1], Ord, 'Bound');
```

$$2$$

```
> `ptolemy/b_point_to_num`([0,2], Ord, 'Bound');
```

$$3$$

```
> `ptolemy/b_point_to_num`([-2,-1], Ord, 'Bound'); Bound;
```

$$1$$

$$[1,2]$$

```
> `ptolemy/b_point_to_num`([-1,-1], Ord, 'Bound');
```

$$4$$

```
> `ptolemy/b_point_to_num`([1,-1], Ord, 'Bound');
```

$$7$$

```
> seq(`ptolemy/b_num_to_point`(i, [1,2], Ord), i=1..9);
```

$$[-2,-1],[-1,-1],[1,-1],[-2,1],[-1,1],[1,1],[-2,2],[-1,2],[1,2]$$

```
> seq(`ptolemy/b_num_to_point`(i, [1], Ord), i=1..3);
```

$$[-2,0],[-1,0],[1,0]$$

```
> seq(`ptolemy/b_num_to_point`(i, [2], Ord), i=1..3);
```

$$[0,-2],[0,-1],[0,1]$$

End of Maple Worksheet

The next example illustrates the process of collocating the boundary spline variables.

Start of Maple Worksheet

```
> with(ptolemy, LogRatioMap);
```

$$[\,LogRatioMap\,]$$

```
> readlib(`ptolemy/b_ops`):
> Map := LogRatioMap(0,1,x,z);
```

$$Map := \left[x \to \ln\left(\frac{x}{1-x}\right), z \to \frac{e^z}{1+e^z}, x \to (\,x, 1-x\,)\right]$$

```
> ExtraBases := table(
>   [(V,1) = [(2*x1+1)*(1-x1)^2, x1*(1-x1)^2, x1^2*(3-2*x1), x1^2*(1-x1)],
>    (V,2) = [1-x2, x2]]);
```

$ExtraBases :=$ table([

$(\,V, 1\,) =$

$[(\,2\,x1 + 1\,)\,(\,1 - x1\,)^2, x1\,(\,1 - x1\,)^2, x1^2\,(\,3 - 2\,x1\,), x1^2\,(\,1 - x1\,)]$

$(\,V, 2\,) = [\,1 - x2, x2\,]$

Maple Worksheet Continued on Next Page

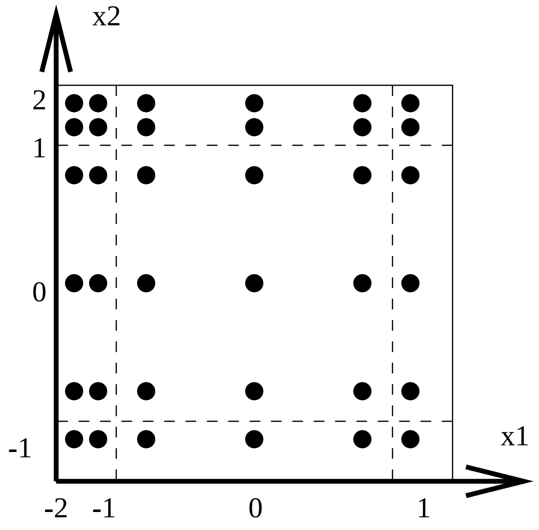

Figure 7.11: A Stylized Illustration of the Collocation Points in the Example

```
                ])

> Exp1 := [ seq((D@@i)(V_B1), i=0..4) ];
```

$$Exp1 := [\,V\_B1, \mathrm{D}(\,V\_B1\,), D^{(2)}(\,V\_B1\,), D^{(3)}(\,V\_B1\,), D^{(4)}(\,V\_B1\,)\,]$$

```
> Exp2 := [ seq((D@@i)(V_B2), i=0..2) ];
```

$$Exp2 := [\,V\_B2, \mathrm{D}(\,V\_B2\,), D^{(2)}(\,V\_B2\,)\,]$$

```
> `ptolemy/collocate_Bvar`(Exp1, {V_B1,V_B2}, [x1,x2],[z1,z2],[Map,Map], ExtraBases);
```

$$\left[\mathrm{V\_B1}\left(\frac{3\,e^{z1}+1}{(1+e^{z1})^3}, \frac{e^{z1}}{(1+e^{z1})^3}, \frac{(e^{z1})^2\,(3+e^{z1})}{(1+e^{z1})^3}, \frac{(e^{z1})^2}{(1+e^{z1})^3}\right),\right.$$
$$\mathrm{V\_B1}\left(-6\,\frac{e^{z1}}{(1+e^{z1})^2}, -\frac{2\,e^{z1}-1}{(1+e^{z1})^2}, 6\,\frac{e^{z1}}{(1+e^{z1})^2}, -\frac{e^{z1}\,(-2+e^{z1})}{(1+e^{z1})^2}\right),$$
$$\mathrm{V\_B1}\left(6\,\frac{-1+e^{z1}}{1+e^{z1}}, 2\,\frac{-2+e^{z1}}{1+e^{z1}}, -6\,\frac{-1+e^{z1}}{1+e^{z1}}, -2\,\frac{2\,e^{z1}-1}{1+e^{z1}}\right),$$
$$\left.\mathrm{V\_B1}(\,12, 6, -12, -6\,), 0\right]$$

```
> `ptolemy/collocate_Bvar`(Exp2, {V_B1,V_B2}, [x1,x2],[z1,z2],[Map,Map], ExtraBases);
```

$$\left[\mathrm{V\_B2}\left(\frac{1}{1+e^{z2}}, \frac{e^{z2}}{1+e^{z2}}\right), \mathrm{V\_B2}(\,-1, 1\,), 0\right]$$

```
> `ptolemy/collocate_Bvar`(Exp2, V_B1, [x1,x2],[z1,z2],[Map,Map], ExtraBases);
```

$$[\,V\_B2, \mathrm{D}(\,V\_B2\,), D^{(2)}(\,V\_B2\,)\,]$$

```
> `ptolemy/collocate_Bvar`(Exp2, V_B2, [x1,x2],[z1,z2],[Map,Map], ExtraBases);
```

$$\left[\mathrm{V\_B2}\left(\frac{1}{1+e^{z2}}, \frac{e^{z2}}{1+e^{z2}}\right), \mathrm{V\_B2}(\,-1, 1\,), 0\right]$$

```
> Exp := [ op(Exp1), op(Exp2) ];
```

$$Exp := [V\_B1, \mathrm{D}(\,V\_B1\,), D^{(2)}(\,V\_B1\,), D^{(3)}(\,V\_B1\,), D^{(4)}(\,V\_B1\,), V\_B2,$$
$$\mathrm{D}(\,V\_B2\,), D^{(2)}(\,V\_B2\,)]$$

```
> BName := {V_B1, V_B2};
```

$$BName := \{\,V\_B1, V\_B2\,\}$$

```
> BasesOrder := [V, [1,0]];
```

$$BasesOrder := [V, [1, 0]]$$

```
> `ptolemy/collocate_Bvar`(Exp, BName,
>   [x1,x2],[z1,z2],[Map,Map], ExtraBases, BasesOrder, [-1,0]);
```

$$\left[\mathrm{V\_B1}(\,1, 0, 0, 0\,), \mathrm{V\_B1}(\,0, 1, 0, 0\,), \mathrm{V\_B1}(\,-6, -4, 6, 2\,),\right.$$
$$\left.\mathrm{V\_B1}(\,12, 6, -12, -6\,), 0, \mathrm{V\_B2}\left(\frac{1}{1+e^{z2}}, \frac{e^{z2}}{1+e^{z2}}\right), \mathrm{V\_B2}(\,-1, 1\,), 0\right]$$

```
> `ptolemy/collocate_Bvar`(Exp, BName,
>   [x1,x2],[z1,z2],[Map,Map], ExtraBases, BasesOrder, [0,1]);
```

$$\left[\mathrm{V\_B1}\left(\frac{3\,e^{z1}+1}{(1+e^{z1})^3}, \frac{e^{z1}}{(1+e^{z1})^3}, \frac{(e^{z1})^2\,(3+e^{z1})}{(1+e^{z1})^3}, \frac{(e^{z1})^2}{(1+e^{z1})^3}\right),\right.$$
$$\mathrm{V\_B1}\left(-6\,\frac{e^{z1}}{(1+e^{z1})^2}, -\frac{2\,e^{z1}-1}{(1+e^{z1})^2}, 6\,\frac{e^{z1}}{(1+e^{z1})^2}, -\frac{e^{z1}\,(-2+e^{z1})}{(1+e^{z1})^2}\right),$$
$$\mathrm{V\_B1}\left(6\,\frac{-1+e^{z1}}{1+e^{z1}}, 2\,\frac{-2+e^{z1}}{1+e^{z1}}, -6\,\frac{-1+e^{z1}}{1+e^{z1}}, -2\,\frac{2\,e^{z1}-1}{1+e^{z1}}\right),$$
$$\left.\mathrm{V\_B1}(12,6,-12,-6), 0, \mathrm{V\_B2}(0,1), \mathrm{V\_B2}(-1,1), 0\right]$$

```
> `ptolemy/collocate_Bvar`(Exp, BName,
>   [x1,x2],[z1,z2],[Map,Map], ExtraBases, BasesOrder, [-1,-1]);
```

$$[\mathrm{V\_B1}(1,0,0,0), \mathrm{V\_B1}(0,1,0,0), \mathrm{V\_B1}(-6,-4,6,2),$$
$$\mathrm{V\_B1}(12,6,-12,-6), 0, \mathrm{V\_B2}(1,0), \mathrm{V\_B2}(-1,1), 0]$$

```
> `ptolemy/collocate_Bvar`(Exp, BName,
>   [x1,x2],[z1,z2],[Map,Map], ExtraBases, BasesOrder, [1,1]);
```

$$[\mathrm{V\_B1}(0,0,1,0), \mathrm{V\_B1}(0,0,0,1), \mathrm{V\_B1}(6,2,-6,-4),$$
$$\mathrm{V\_B1}(12,6,-12,-6), 0, \mathrm{V\_B2}(0,1), \mathrm{V\_B2}(-1,1), 0]$$

End of Maple Worksheet

Method of Implementation

The b_point_to_num Procedure This procedure first scans the argument `Point` for nonzero elements in order to construct the set of boundary components. The set of boundary components is actually assigned to the argument `BoundSet` as a list so as to preserve the order.

Then the procedure makes a second pass through the components of `Point` which are in the boundary set in order to figure the sequence number, but this time in the reverse order. Each relevant coordinate is compared to the corresponding order information to decide if it on the "LOW" end or the "HIGH" end of this dimension. In either case a simple (though different) formula is used to determine the sequence number within this dimension. The overall sequence number is accumulated by summing the produce of each dimensional sequence number with the dimension's stride, where the stride is defined to be the change in sequence number for a unit change in the corresponding dimension. The stride may be computed on the fly from the value of the stride for the previous dimension.

Because it is easier to compute the sequence number in zero-based arithmetic the procedure computes each the sequence number in each dimension assuming zero-based counting. Then the final result is incremented to convert it to one-based counting system used in PTOLEMY.

The b_num_to_point Procedure This procedure first computes the stride for every dimension in the boundary set and then on the second pass computes the coordinate number associated with each dimension in the boundary set. This is necessary because the stride must be computed from the last dimension towards the first while the coordinates must be computed from the first dimension towards the last.

The coordinates are computed by first getting the sequence number in the current dimension, then checking to see if the result is on the "LOW" end or the "HIGH" end, and then using the inverse of the formulas used by **b_point_to_num**. The sequence number in the current dimension is computed by performing an integer division of the current remainder and the corresponding stride; the remainder is used for the next dimension.

Because this calculation is easier and faster to perform for zero-biased array indexing (as opposed to one-based array indexing) the sequence number is first decremented by one.

The final point should contain zeros in the dimensions not in the boundary set. This result is constructed by using a `subsop` to insert the coordinates for dimensions in the boundary set into a list of zeros.

The collocate_Bvar Procedure This procedure first calls **get_D_forms** to extract a set of all of the D-operators involving B-variables. If this set is empty then the procedure skips all other operations. Otherwise, the procedure collocates each D-operator involving B-variables and substitutes the results back into the expression.

Each D-operator involving B-variables is collocated by:

1. Determining the order of the D-operator and parsing the variable name to determine the dimension associated with the bases component. The procedure **get_D_forms** is called for this purpose.

2. Extracting the associated boundary splines from the argument `ExtraBases` and applying the D-operation to each boundary spline.

3. Mapping the result using **fast_map**.

4. Applying the B-variable at these mapped boundary splines in order to represent the result in positional notation.

If the optional seventh and eighth arguments indicate that collocation is to be applied on the boundary then the mapping of the arguments is more complex but the other steps are unchanged. Specifically if the order of differentiation is less than the order of approximation on the specified end, then the value of applying the D-operator to each of the boundary splines can be determined simply by their sequence number. Specifically, the extra bases are constructed so that at each end point all of their derivatives up to the order of approximation are zero except for one derivative which is exactly equal to one.

However, if the order of differentiation is greater than order of approximation then the mapping of the extra bases is performed in exactly the same manner as when the collocation occurs in the interior. The limit of the result as the corresponding new coordinate approaches plus or minus infinity is used.

Dependences

The procedures **b_point_to_num** and **b_num_to_point** call **order_expand**, in the module ***order_ops***, to expand the order information provided by the argument `Ord`.

The procedure **collocate_Bvar** calls the procedure **get_D_forms** to create a set of all differential forms involving B-variables. It also calls **isDop** to extract the B-variable associated with the current differential form. Finally it calls **fast_map** to actually map the extra bases.

If the collocation is performed on the boundary then the procedure will also call the **order_expand** to expand the order information provided in the argument `OrderSpec`, but the procedure might not call **fast_map**.

kron_ops

This LLF provides procedures for converting equations and expressions from `SB`-notation to Kronecker product notation. The procedures that are indented primarily for external usage are:

kron_eq(`Eq`: *equation*, `RowCoords`: *list(nonzeroint)*, `Stack`: *list(name)*, `Weight`: *table*, `HName`: *list(name)*)
kron_exp(`Exp`: *algebraic*, `Sign`: {*-1, 1*}, `RowCoords`: *list(nonzeroint)*, `Stack`: *list(name)*, `Weight`: *table*, `HName`: *list(name)*)
kron_accumulate(`Exp`: *algebraic*, `Sign`: {*-1, 1*}, `RowCoords`: *list(nonzerint)*, `Weight`: *table*, `HName`: *list(name)*, `Row`: *table*, `RHS`: *name*)

The procedure **kron_eq** returns a row vector and a "scalar" term called the RHS. The equation formed by equating the product of the row vector and the column vector (specified in the argument `Stack`) with the RHS is equivalent to the constraint specified by the argument `Eq`.

The behavior of **kron_exp** is similar, except that the product of the row vector and the column vector minus the RHS is equivalent to `Sign*Exp`. In both cases the result is represented in the form *[list(algebraic), algebraic]*, where the first element is the row vector and the second element is the RHS.

The procedure **kron_accumulate** is equivalent to the procedure **kron_exp**, except that the results are "accumulated" in the arguments `Row` and `RHS`. That is, the results are added to the the initial values of the variables named by these argument.

In each case the row vector contains elements that are Kronecker product representations of matrices. That is the row vector is a vector of matrices so it is typically also a matrix. Stated differently the Kronecker products notation allows the representation of a collection of row vectors of numbers by a single row vector of Kronecker products.

Since each row vector of numbers is associated with a specific collocation point, each row vector of Kronecker products is associated with a collection of collocation points. The argument `RowCoords` indicates 1) which coordinates vary over the collection of collocation points, 2) the order of the rows, i.e., collocation points. The order of the rows is constrained to be some cartesian ordering (see "cartesian ordering" in the glossary) so the only ordering information that must be specified is the order of the coordinates and the order of the collocation points within each coordinate. The order of the rows is further restricted so that the collocation points within each coordinate must be either ascending or descending. This information is encoded into the `RowCoords` argument by specifying the coordinate number for the coordinates that vary over this set of rows in the order used to specify the cartesian ordering. If the collocation points are in descending order for the particular coordinate the negative of the coordinate number is used.

For example, consider a three-dimensional problem and a collection of rows such that only the first and third coordinates vary over the collection collocation points. Then `RowCoords` should equal `[1,3]`, `[-1,3]`, `[1,-3]`, `[-1,-3]`, `[3,1]`, `[-3,1]`, `[3,-1]`, or `[-3,-1]` depending on the ordering of the rows. If the rows are in the "natural" order then the value of `RowCoords` should be `[1,3]`. If, however, the rows are ordered so that the first coordinate varies most rapidly and the third coordinate actually decreases with increasing row number, then the value of `RowCoords` should be `[-3,1]`.

The argument `Stack` specifies the column vector to be multiplied by the row vector of Kronecker product terms. Alternately, it specifies the order of stack-variable names, to be used in the row vector formulation. The stack-variable names are constrained to a fixed set of variations on the state-variable names, specifically if `V` is one of the state-variable names then the stack variable names may be `V` and `cat('V_B', Comb, '_', Num)` where `Comb` is an ordered list of the dimensions which are *constant* for this group of bases and `Num` indicates which subgroup of bases are being specified. So in effect, `Stack` only indicates the order of the bases subgroups. This is why this argument is not not needed by the procedure **kron_accumulate**.

The argument `Weight` is a table of weights indexed by the state-variable name and the dimension. Finally, the argument `HName` is a list of the variables names which should be used to denote the sampling width. These variables are introduced into the diagonal components of the result during the conversion of differential forms to Kronecker product notation.

Support Procedures

In addition there are two lower-level procedures in this module that might be called by the user, but that were designed primarily to support the three procedures described in the introduction to this section.

kron_term(`Term:` *algebraic*, `Weight:` *table*, `HNames:` *list(name)*,
`RowOrient:` *list({-1, 0, 1})*)
kron_term(`Term:` *algebraic*, `Weight:` *table*, `HNames:` *list(name)*,
`RowOrient:` *list({-1, 0, 1})*, `Permutation:` *list(posint)*)
split_term(`Term:` *algebraic*, `SFacts:` *name*, `BFacts:` *name*, `Coeff:` *name*,
`StateVar:` *name*)

The procedure **kron_term** converts a single term into a Kronecker product format. The procedure **split_term** groups all of the factors in a term according to whether they contain `S`-variables, `B`-variables, or neither. Because it is necessary to understand the material in the next section in order to more fully describe these procedures a complete description of the interface is deferred to the subsection titled "Method of Implementation" on page on page 293.

Constraints on the Form of the Input

All of the procedures in this module assume that the `SB`-notation expressions have been normalized to a form compatible with the following grammar.

$$\begin{aligned} equation &:= expression = expression \\ expression &:= term(+term)^* \\ term &:= factor(*term)^* \\ factor &:= \{\mathbf{Constant}, \mathbf{SVarDOp}, \mathbf{BVar}\} \\ SVarDOp &:= \{\mathbf{D}(SVarDOp), \mathbf{SVar}\} \end{aligned}$$

where **Constant** is any expression that is constant with respect to the SB-variables (but typically not with respect to the coordinates), **SVar** is the stack variable name of one of the sinc-bases component groups (e.g., V_S1), **BVar** represented one of the boundary spline bases component groups (in positional notation), and **D** is Maple's D-operator.

Employing this assumption, the module implements a *recursive descent parser.* This lexically oriented approach has several drawbacks. The form of the expressions is critical; mathematically equivalent but syntactically nonconforming expressions will cause errors (possibly even undetected errors). In my opinion the appropriate elegant solution is to employ modern algebra to construct a class of expression whose unique normal form is the desired result. See [6] for several examples of normalizing various classes of expressions.

Developing such an approach is a small research project in its own right, whereas implementing this recursive descent parser was at most a couple of days work.

Examples

Start of Maple Worksheet

```
> ptolemy[init]();
> readlib(`ptolemy/kron_ops`):
```

```
> Stack := [V, seq(cat(`V_B1_`,i), i=1..4), seq(cat(`V_B2_`,i), i=1..4),
>   seq(cat(`V_B12_`,i),i=1..16)];
```

$$Stack := [V, V\_B1\_1, V\_B1\_2, V\_B1\_3, V\_B1\_4, V\_B2\_1, V\_B2\_2, V\_B2\_3, V\_B2\_4, V\_B12\_1, V\_B12\_2, V\_B12\_3, V\_B12\_4, V\_B12\_5, V\_B12\_6, V\_B12\_7, V\_B12\_8, V\_B12\_9, V\_B12\_10, V\_B12\_11, V\_B12\_12, V\_B12\_13, V\_B12\_14, V\_B12\_15, V\_B12\_16]$$

```
> W1 := exp(z1)/(1+exp(z1))^2;
```

$$\mathrm{W1} := \frac{e^{z1}}{(1+e^{z1})^2}$$

```
> W2 := exp(z2)/(1+exp(z2))^2;
```

$$\mathrm{W2} := \frac{e^{z2}}{(1+e^{z2})^2}$$

```
> Weight := table([(V,1) =W1, (V,2) = W2]);
```

$$Weight := \mathrm{table}([\; (V,1) = \frac{e^{z1}}{(1+e^{z1})^2} \quad (V,2) = \frac{e^{z2}}{(1+e^{z2})^2} \;])$$

```
> HName := [H1,H2];
```

$$HName := [H1, H2]$$

```
> Sp1 := 1/(1 + exp(z1)); Sp2 := exp(z2)/(1 + exp(z2));
```

$$Sp1 := \frac{1}{1+e^{z1}}$$

$$Sp2 := \frac{e^{z2}}{1+e^{z2}}$$

```
> Eq := D(V_S1)*V_S2*exp(z1)/(1+exp(z1))  + D(V_S1)*V_B1(Sp1,Sp2)= K;
```

$$Eq := \frac{D(\ V\_S1\ )\ V\_S2\ e^{z1}}{1+e^{z1}} + D(\ V\_S1\ )\ V\_B1\left(\frac{1}{1+e^{z1}}, \frac{e^{z2}}{1+e^{z2}}\right) = K$$

```
> `ptolemy/kron_eq`(Eq, [-1,2], Stack,Weight,HName);
```

$$\left[\left[\mathrm{Diag}\left(\frac{(\ e^{z1}\ )^2\ e^{z2}}{(\ 1+e^{z1}\ )^3\ H1\ (\ 1+e^{z2}\ )^2}\right)(\ I1\_rev\ \&K\ I0\ ), 0, 0, 0, 0,\right.\right.$$
$$\mathrm{Diag}\left(\frac{e^{z1}}{H1\ (\ 1+e^{z1}\ )^3}\right)(\ I1\_rev\ \&K\ C\ ),$$
$$\mathrm{Diag}\left(\frac{e^{z1}\ e^{z2}}{H1\ (\ 1+e^{z1}\ )^2\ (\ 1+e^{z2}\ )}\right)(\ I1\_rev\ \&K\ C\ ), 0, 0, 0, 0, 0, 0, 0, 0,$$
$$\left.\left.0, 0, 0, 0, 0, 0, 0, 0, 0\right], K\right]$$

```
> `ptolemy/kron_eq`(Eq, [1,-2], Stack,Weight,HName);
```

$$\left[\left[\mathrm{Diag}\left(\frac{(\ e^{z1}\ )^2\ e^{z2}}{(\ 1+e^{z1}\ )^3\ H1\ (\ 1+e^{z2}\ )^2}\right)(\ I1\ \&K\ I0\_rev\ ), 0, 0, 0, 0,\right.\right.$$
$$\mathrm{Diag}\left(\frac{e^{z1}}{H1\ (\ 1+e^{z1}\ )^3}\right)(\ I1\ \&K\ C\_rev\ ),$$
$$\mathrm{Diag}\left(\frac{e^{z1}\ e^{z2}}{H1\ (\ 1+e^{z1}\ )^2\ (\ 1+e^{z2}\ )}\right)(\ I1\ \&K\ C\_rev\ ), 0, 0, 0, 0, 0, 0, 0, 0,$$
$$\left.\left.0, 0, 0, 0, 0, 0, 0, 0, 0\right], K\right]$$

```
> readlib(forget)(`ptolemy/kron_ops`);
> `ptolemy/kron_eq`(Eq, [2,1], Stack,Weight,HName);
```

$$\left[\left[\mathrm{Diag}\left(\frac{(\ e^{z1}\ )^2\ e^{z2}}{(\ 1+e^{z1}\ )^3\ H1\ (\ 1+e^{z2}\ )^2}\right)P(\ 2, 1\ )(\ I1\ \&K\ I0\ ), 0, 0, 0, 0,\right.\right.$$
$$\mathrm{Diag}\left(\frac{e^{z1}}{H1\ (\ 1+e^{z1}\ )^3}\right)P(\ 2, 1\ )(\ I1\ \&K\ C\ ),$$
$$\mathrm{Diag}\left(\frac{e^{z1}\ e^{z2}}{H1\ (\ 1+e^{z1}\ )^2\ (\ 1+e^{z2}\ )}\right)P(\ 2, 1\ )(\ I1\ \&K\ C\ ), 0, 0, 0, 0, 0, 0, 0,$$
$$\left.\left.0, 0, 0, 0, 0, 0, 0, 0, 0, 0\right], K\right]$$

```
> `ptolemy/kron_eq`(Eq, [-2,1], Stack,Weight,HName);
```

$$\left[\left[\mathrm{Diag}\left(\frac{(\ e^{z1}\ )^2\ e^{z2}}{(\ 1+e^{z1}\ )^3\ H1\ (\ 1+e^{z2}\ )^2}\right)P(\ 2, 1\ )(\ I1\ \&K\ I0\_rev\ ), 0, 0, 0, 0,\right.\right.$$
$$\mathrm{Diag}\left(\frac{e^{z1}}{H1\ (\ 1+e^{z1}\ )^3}\right)P(\ 2, 1\ )(\ I1\ \&K\ C\_rev\ ),$$

$$\mathrm{Diag}\left(\frac{e^{z1}\,e^{z2}}{H1\,(1+e^{z1})^2\,(1+e^{z2})}\right)\mathrm{P}(2,1)\,(\,I1\ \&\mathrm{K}\ C\_rev\,),0,0,0,0,0,$$
$$0,0,0,0,0,0,0,0,0,0,0,0\Big],K\Big]$$

```
> `ptolemy/kron_eq`(Eq, [1], Stack,Weight,HName);
```

$$\Bigg[\Bigg[\mathrm{Diag}\left(\frac{(e^{z1})^2}{(1+e^{z1})^3\,H1}\right)(\,I1\ \&\mathrm{K}\ R0\,),0,0,0,0,\mathrm{Diag}\left(\frac{e^{z1}}{H1\,(1+e^{z1})^3}\right)I1,$$
$$\mathrm{Diag}\left(\frac{e^{z1}\,e^{z2}}{H1\,(1+e^{z1})^2\,(1+e^{z2})}\right)I1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,$$
$$0,0,0,0\Big],K\Big]$$

```
> `ptolemy/kron_eq`(Eq, [2], Stack,Weight,HName);
```

$$\Bigg[\Bigg[\mathrm{Diag}\left(\frac{e^{z1}\,e^{z2}}{(1+e^{z1})\,(1+e^{z2})^2}\right)(\,R1\ \&\mathrm{K}\ I0\,),0,0,0,0,$$
$$\mathrm{Diag}\left(\frac{1}{1+e^{z1}}\right)(\,R1\ \&\mathrm{K}\ C\,),\mathrm{Diag}\left(\frac{e^{z2}}{1+e^{z2}}\right)(\,R1\ \&\mathrm{K}\ C\,),0,0,0,0,0,$$
$$0,0,0,0,0,0,0,0,0,0,0,0\Big],K\Big]$$

```
> `ptolemy/kron_eq`(Eq, [], Stack,Weight,HName);
```

$$\Bigg[\Bigg[\mathrm{Diag}\left(\frac{e^{z1}}{1+e^{z1}}\right)(\,R1\ \&\mathrm{K}\ R0\,),0,0,0,0,\mathrm{Diag}\left(\frac{1}{1+e^{z1}}\right)R1,$$
$$\mathrm{Diag}\left(\frac{e^{z2}}{1+e^{z2}}\right)R1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\Big],K\Big]$$

End of Maple Worksheet

Method of Implementation

The recursive descent parser is implemented by the procedures described in the next three subsections. In addition to parsing, **kron_accumulate** converts the information in `RowCoords` to the information needed to provide the `RowOrient` and `Permutation` arguments of **kron_term**.

kron_aaccumulate This procedure first converts the information contained in `RowCoords` into a form that is easier to use, although less compact. First the absolute value of each element of `RowCoords` is stored in a local variable, called `RowDim`. This list of dimensions that vary over the current collection of rows is then sorted. If the sorted order is not the same as the order specified in `RowCoords` then a permutation matrix will be required in the output and the procedure will compute the information for this matrix.

Explaining the construction of the permutation matrix requires the introduction of new notation. For coordinates that vary in this region define the *region coordinate number* to be the position number of the coordinate within the subset of coordinates that vary. In this step the coordinates must be ordered as they are in the definition of the the cartesian ordering of the rows; this may not be the natural ordering of the coordinates. Next define the *domain coordinate number* to be the position of a coordinate in the natural coordinate ordering. For coordinates that vary in this region define the *ordered region coordinate number* to be the position number of the coordinate within the subset of coordinates that vary in this region when this subset of coordinates is ordered using the natural ordering.

Then define f to be the function that maps the "region coordinate number" to the "domain coordinate number." Similarly define g to be the function that maps the "domain coordinate number" to the "ordered region number." It follows that `RowDim` directly Specifies f and sorting `RowDim` directly specifies g^{-1}. In order to define the permutation matrix it is necessary to compute $f^{-1} \circ g^{-1}$ over an ordered sequence of inputs. Computing f^{-1} is performed by copying the information contained in `RowDim` into a table.

The first parsing operation performed by the procedure is to determine if `Exp` is a sum of terms or a single term. It then calls the procedure **kron_term** for each term in the equation. If the result from **kron_term** is an algebraic expression then the term was a constant, with respect to the `SB`-variables and the result is subtracted from the RHS. If the result is of type ***[name,algebraic]***, then the result is a single block in the row vector and is added to the corresponding table entry in `Row`. If the result is of type ***set([name, algebraic])*** then the result is a set of matrix blocks each one of which is added to the appropriate entry in `Row`.

kron_term The procedure **kron_term** returns three different types depending on the input. These three types may be aggregated into the following return type

$$\{\texttt{algebraic,[name,algebraic],set([name,algebraic])}\}.$$

Each of these types corresponds to one of the following circumstances.

1. If the term is constant with respect to the `SB`-variables then it is not associated with any stack variable and the result will be of type ***algebraic***. In fact, in this case the result will be the input unchanged.

2. If the term depends only on sinc bases components (i.e., contains `S`-variable but not `B`-variables) then the term is associated with the stack variable with the same name as the state-variable. In this case the result will be a list containing the associated stack variable name and the result of converting the term into Kronecker product notation.

3. If the term depends on some boundary splines (i.e., contains some `B`-variables) then the term will represent more than one element in the matrix. This is because in positional format each `B`-variable represents several bases component groups, specifically, one bases component group for each argument. In this case the result will be a set of Kronecker product blocks, each represented as a list containing the associated stack variable name and the associated Kronecker product block.

   If the term contains more than one `B`-variable then each `B`-variable defines multiple bases component groups. In this case the result must contain a nonzero block for each of the elements in the cross-product of the set of bases component groups defined by each `B`-variable. In addition, the name of each associated stack variable depends on all the dimensions for which the corresponding basis components are boundary splines.

The first operation of the procedure is to call **split_term** to separate the factors of the term. If the term does not have factors containing `S`-variables nor factors containing `B`-variables then the result is a constant term and no further work is required.

Otherwise the procedure constructs the Kronecker product portion of the result. In the case where the term contains `B`-variables and the result will represent several different matrix blocks, each matrix block will still have the same Kronecker product portion, that is each result differs only in the coefficient part. This is because the arguments of the `B`-variables specify only the portion of the coefficient that differs from one basis component groups to the next.

To construct the Kronecker product portion of the result the procedure first determines which coordinates vary in the column direction. This is necessary for determining the shape of the matrices in the result, i.e., whether an individual factor in the Kronecker product is a row vector, column vector, or a full matrix. The dimensions that do not vary in the column direction are those dimensions that are not associated with `S`-variables.

While determining which coordinates vary in the column direction the procedure also determines the order of differentiation applied to each `S`-variable. This information is used to determine which matrix will be used in the Kronecker product, e.g., $I^{(0)}$, $I^{(1)}$, $I^{(2)}$, and so on. Notice that the order of differentiation of the `B`-variables is always zero since differentiation of the original equations in the directions where the bases component was a boundary spline was performed by directly differentiating the boundary spline. That is the arguments of the `B`-variables already represent the effect of any differentiation in the corresponding dimension.

Combining information about which of the coordinates vary in the column direction with information about which coordinates vary in the row direction, information about whether the coordinates decrease or increase in the row ordering, and information about the order of the differentiation in each dimension, the procedure constructs the Kronecker product portion of the result. Information about which coordinates vary in the row direction and about the ordering of the rows with respect to specific coordinates is extracted from the argument `RowOrient`.

Next the procedure checks to see if the optional argument `Permutation` was provided. If so it checks the type of the argument and constructs the permutation matrix to be inserted between the diagonal matrices and Kronecker product portion of the matrix. Finally, the diagonal matrix to represent the coefficient portion term is constructed.

In the case where the term contains `B`-variables the appropriate coefficient will be the product of the term coefficient and the coefficient which vary from one bases component group to the next, i.e., the portions of the coefficient specified as arguments to of the `B`-variables. These coefficients are constructed by calling **cross_prod** with the arguments from each `B`-variable. The result is multiplied by the constant portion of the term and the result is paired with the corresponding stack variable name.

If the symbolic representation of the diagonal matrix is constructed using factors from different sources, then the procedure specified by the global variable **ptolemy/SimpProc** is applied to the expression before it is used in the final result. However, if the coefficient in merely copied from the input no simplification is performed, the assumption being that any desired simplification has already been performed.

split_term The procedure **split_term** divides each of the factors of the term, specified by the argument `Term`, into one of three lists. This procedure also returns the name of the state-variable associated with this term.

Because the problem is assumed to be linear there must be only one state-variable associated with each term, that is all of the `SB`-variables will be associated with the same state-variable. This name is assigned to the variable named by the argument `StateVar`. The result assigned to the variable named by the argument `SFacts` is a list of the factors that are `D`-operators involving one of the `S`-variables. The result assigned to the variable named by `SFacts` is a similar list for the factors which are `B`-variables. And the result assigned to the variable named by `Coeff` will be the product of all the other factors.

This procedure calls **get_SB_var** to get a list of the variables names. It then separates this list into `S`-variables (.i.e., those containing "_S") and `B`-variables(i.e., those containing "_B"). Next it examines these lists to extract the state-variable name and assigns it to the variable specified by `StateVar`. In the second pass the procedure checks each factor to see if it contains one of the `S`-variables or one of the `B`-variables. Those that contain neither are part of the coefficient.

Dependencies

The procedure **split_term** calls the procedure **get_SB_var** to extract the names of the bases components.

The procedure **kron_term** calls **isDop** to extract the name of the bases component from each factor containing `S`-variables. The name is required in order to parse out the associated dimension information and for the call to **de_order** which is used to figure out what order 'I'-matrix is required. Finally, **cross_prod** may be called to construct a list of all the output results when the term contains `B`-variables.

to_string

This function converts a Maple structure to a string.

to_string(Obj)

Earlier versions of Maple's **write** procedure produced highly unnormalized representations of floating point numbers. Even in the current version of Maple, the **cat** procedure will not handle floating point numbers. In addition this function converts commas into spaces in lists to make the result more easily read form C.++

As Maple is improved the need for this function should disappear.

Example

Start of Maple Worksheet

```
> with(ptolemy,to_string);
                                  [to_string]

> to_string(sqrt(2.));
                                  1.414213562

> to_string(evalf(E^3));  evalf(E^3);
                                  20.08553691
                                  20.08553691

> to_string(evalf(Pi*10^9));
                                  3141592654

> to_string(evalf(1/Pi));
                                  0.3183098861

> to_string(evalf(1/Pi)*10^(-2));
                                 0.003183098861

> to_string(evalf(1/Pi)*10^(-3));
                                3.183098861e − 4
```

```
> to_string(evalf([[1/sqrt(2), 1/sqrt(3)], [2, 3]]));
                      [[0.7071067810 0.5773502693] [2 3 ]]

> to_string(Diag(exp(z1), exp(z2)));
                           Diag(exp(z1 ) exp(z2 ) )
```

End of Maple Worksheet

Bibliography

[1] ALFRED V AHO, JOHN E. HOPCROFT, J. D. U. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass, 1983.

[2] CHAR, B. W. *Maple V: Language Reference Manual.* Springer-Varlag, 1992.

[3] HECK, A. *Introduction to Maple.* Springer-Varlog, 1996.

[4] JOHN ABBOTT, ANDRE VAN LEEUWEN, A. S. Objectives of openmath. Technical Report 12, Research Institute for Applications of Computer Algebra, 1996.

[5] JOHN LUND, K. L. B. *Sinc Methods for Quadrature and Differential Equations.* Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[6] KEITH O. GEDDES, S. R. CZAPOR, G. L. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, Boston, 1992.

[7] PARKER, K. *Automatic Setup of Sinc-Collocation for Partial Differential Equations.* PhD thesis, University of Utah, 1996.

[8] SCHREIBER, R. Block algorithms for parallel machines. In *Numerical Algorithms for Modern Parallel Computer Architectures*, M. Schultz, Ed. Springer-Verlag, 1980, pp. 197–207.

[9] SEDGEWICK, R. *Algorithms in C++.* Addison-Wesley, Reading, Mass., 1992.

[10] STENGER, F. *Numerical Methods Based on Sinc and Analytic Functions.* Springer-Verlag, New York, 1993.

[11] WIRTH, N. *Programming in Modula-2.* Springer-Verlag, New York, 1982.

[12] ZAYED, A. I. *Advances in Shannon's Sampling Theorem.* CRC Press, Boca Raton, 1993.