PyGDH Documentation

Release 0.4.3

Kenneth Higa

Feb 19, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION: IS GRID DISCRETIZATION HELPER RIGHT FOR YOU?

Grid Discretization Helper (abbreviated as PyGDH, and probably most easily pronounced as "pigged") simplifies the process of obtaining numerical solutions (presently by the finite volume method) to systems of (potentially nonlinear and time-dependent) discretized differential equations (and other types of equations) on one-dimensional and some two-dimensional domains. By managing many of the numerical details, PyGDH frees the user to concentrate on formulating the discretized mathematical problem.

There are many libraries and software packages with similar goals. PyGDH is not particularly efficient, nor is it suited to solving equations at high resolution or on domains with complicated shapes. However, it aims to be simple to use while providing substantial flexibility, and has already been used to solve fairly sophisticated problems.

PyGDH was developed for relatively small problems, for which the amount of computer time the can be saved by using more capable software is negligible compared to the amount of human time that can be saved by the simple approach that it offers. Some users may find it to be a useful tool for quickly getting intuition about a difficult problem before turning to more sophisticated packages.

In contrast with packages which work directly with the original mathematical problem statement, PyGDH requires users to provide the discretized form of the problem. Discretized forms suitable for PyGDH are often easy to obtain (see the sections on discretization), and working directly with the discretized equations gives users control over many numerical details. The discretized problems are represented as programs written in the Python programming language.

Prior programming experience is helpful, but not necessary, as the tutorial is meant to be a self-contained introduction to working with PyGDH. PyGDH is designed so that problem-solving programs can be nearly self-documenting and can be quickly understood and updated by a succession of users as the needs of a project evolve. The underlying Python programming language is popular and easy to learn, and remains fully-accessible to the user, making it possible to use PyGDH in very flexible ways.

As its name implies, PyGDH is limited to solving problems on domains that can be mapped to a rectangular grid (that is, with boundaries and divisions along which one spatial coordinate does not vary), and presently in one or two spatial dimensions. This allows spatial domains to be described in a simple way, but comes at the cost of flexibility, as PyGDH cannot be applied to domains of arbitrary shape.

While PyGDH is not yet used widely in the scientific community, it relies heavily on mature Python libraries and comes with a test suite that performs validation against analytical solutions.

CHAPTER

TWO

INSTALLATION

Windows, Mac OS X, GNU/Linux, and many other operating systems provide command-line interfaces. Installation and use of PyGDH, as described in this tutorial, will require limited use of your computer's command-line interface (CLI). For Microsoft Windows, this is the Command Prompt. For GNU/Linux, Mac OS X, and similar systems this might be a Terminal or xterm.

2.1 Introduction to command-line interfaces

CLIs allow users to interact with computer systems by typing commands. The system prompts the user for a command, the user responds by typing a command and pressing the "Enter" key, and the computer attempts to accomplish the given task. This may involve doing calculations and displaying results on the screen or writing results into a file. If the computer is unsuccessful, which can occur for a variety of reasons, it typically displays an error message. Whether successful or unsuccessful, once the attempt is complete, the computer returns to the start of the cycle and prompts the user for another command.

In the CLIs provided with many operating systems, simply giving the name of a program as a command causes the system to "run" the named program by starting the program and giving it control of the interface.

As an example, running the Windows "Command Prompt" program causes a window containing a CLI to appear. The system shows a command prompt that typically looks like:

```
C:\Users\UserName>
```

The Python program is typically named `python`, although this will vary among installations. If Python is already installed on your system, entering this in the Windows CLI starts Python in interactive mode:

```
C:\Users\UserName>python
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python prompts the user for commands as well, within the same interface. The >>> is Python's command prompt, which is completely distinct from those typed at the Windows command prompt. Commands entered here are instructions to Python, rather than to the Windows Command Prompt. For example, the following commands ask Python to load "libraries" that enhance its capabilities:

```
>>> import scipy
>>> import h5py
```

If these commands do not result in error messages, the required and suggested libraries are already installed on your computer, and you can skip the next section.

To exit from Python and return to the Windows command prompt, one may type `exit()` at the Python prompt:

```
>>> exit()
C:\Users\UserName>
```

The `exit()` command tells Python that its task is complete. When the operating system sees that a task has completed, it once again offers its command prompt to the user.

2.2 Python and required libraries

PyGDH is compatible with both Python 2 and Python 3.

For users of various versions of Microsoft Windows and Mac OS X, there are Python distributions such as Python(x,y), Anaconda, and Enthought Canopy that contain Python along with a number of common libraries. Some of these distributions are quite large.

PyGDH has been tested on Python 2.7.3 as packaged in Python(x,y) 2.7.3.1 on a computer running Microsoft Windows 7. The Python(x,y) website contains installation details.

However, users of GNU/Linux may prefer to install only the necessary packages through a package manager. For example, on the system on which PyGDH was initially developed (running a 64-bit PC installation of Debian GNU/Linux 6.0), the required and suggested Python packages can be installed by running from the command line (with sufficient permissions):

```
apt-get install python-scipy python-h5py
```

or for Python 3, the following might be needed:

```
apt-get install python3-scipy python3-h5py
```

On a CentOS 6 system, these packages can be installed by running from the command line (with sufficient permissions):

```
yum install scipy h5py
```

For users who wish to download and install the necessary packages individually, software may be downloaded from the Python and SciPy websites. NumPy is included with SciPy.

The HDF5 library and the h5py Python interface to this library are optional, but are recommended for efficient access to stored results. Please note that installation of HDF5 on Mac OS X is quite involved.

It is strongly recommended that users also install:

1. A "programmer's" text editor, such as Emacs or Notepad++ that understands Python code and provides syntax highlighting.
2. A plotting program, such as GNUPLOT, which is used in this tutorial, and optionally, matplotlib for generating plots from within Python.
3. Cython, a Python-to-C compiler, which will make the PyGDH library faster and help users to make their code faster. Cython installation instructions are available here, and as the page describes, a C compiler, such as gcc or Clang, must also be present (the Cython compiler produces C source code, from which the C compiler then produces a program in the native language of the computer).

2.3 PyGDH installation

The PyGDH distribution comes in two compressed formats: `pygdh-0.4.3.zip` for Windows, and `pygdh-VERSION.tar.gz` for most other systems.

Unpack the downloaded file into a directory (also known as a "folder"). This will create a subdirectory named `pygdh-0.4.3`. Make note of the location of this directory. On a Windows computer, this might be `C:\Users\UserName\Downloads\pygdh-VERSION`

Open your computer's CLI. Interactions with the CLI are always understood to be taking place "within" a present directory location, and it is most convenient to interact with files located in the present directory. Commands can be given to the CLI to change the present directory location. On Windows, Mac OS X, and GNU/Linux systems, the `cd` (Change Directory) command can be used to move to another directory. Under Windows, the appropriate command will be similar to:

```
cd C:\Users\UserName\Downloads\pygdh-0.4.3
```

while on Mac OS X or GNU/Linux, the appropriate command will be similar to:

```
cd /home/UserName/Downloads/pygdh-0.4.3
```

Once in the directory with the installation files, you may use a command such as `dir` (on Windows) or `ls` on Linux or Mac OS X to see a list of files in the directory:

```
08/14/2014  11:33 AM    <DIR>          .
08/14/2014  11:33 AM    <DIR>          ..
08/14/2014  11:33 AM    <DIR>          pygdh
08/13/2014  12:14 PM               248 PKG-INFO
06/25/2014  04:12 PM                16 setup.cfg
07/22/2014  09:51 AM               397 setup.py
               3 File(s)            661 bytes
```

The `setup.py` is used to tell Python about how PyGDH should be installed. If you wish to perform a system-wide installation, and you possess sufficient administrative privileges to do so, type the following at the command line to install:

```
python setup.py install
```

Alternatively, you can install PyGDH within your own user directories, for which no special administrative access is needed, by running:

```
python setup.py install --user
```

If you have installed Cython and a C compiler, and you wish to make use of the faster Cython-compiled PyGDH libraries, then you may also type:

```
python cython_setup.py install
```

or:

```
python cython_setup.py install --user
```

At present, this is not known to work on Mac OS X installations with Xcode.

To check that PyGDH has been properly installed, start Python as before (by typing `python` at the command line interface), and at the Python prompt, type:

```
import pygdh
```

If no errors were encountered, PyGDH is ready to use. The directory into which the installation files were unpacked is no longer needed and may be removed.

CHAPTER

THREE

TUTORIAL

This tutorial introduces Grid Discretization Helper through a series of examples.

3.1 Introductory remarks

This tutorial attempts to be self-contained. It introduces many of the elements necessary to use PyGDH effectively, including object-oriented programming, the Python language, discretization techniques, and validation of solutions. While many problems can be solved by using the information provided in earlier chapters, many details are discussed only in the later chapters.

PyGDH offers a flexible but simple approach to solving equations by helping users to express discretized equations in the form of a Python program. Some familiarity with Python is required in order to use PyGDH. This tutorial attempts to provide enough description to give the reader a working understanding of Python programming. The next chapter provides a brief introduction to the Python language, and additional concepts will be introduced as necessary when example code is examined. Readers seeking a deeper understanding of the Python language are encouraged to read the official Python documentation for their installed version of Python.

3.2 An introduction to Python

This chapter introduces a number of Python concepts, primarily through an example program that will be examined in detail in later chapters. Readers seeking a deeper understanding of the Python language are encouraged to read the official Python documentation.

3.2.1 Running Python programs

Programs written in the Python programming language are stored in the form of text files with names ending in "`.py`". The contents of these text files are called "source code." The Python system reads the source code and directs the computer to act according to the program instructions. Many of the code samples included here appear in our first example program, located at `pygdh/examples/ODE/ODE.py` in the PyGDH archive.

In order to execute source code written in the Python language, a user must ask the computer to start the Python system and direct it to read and carry out the instructions in the Python source code. Typically, both of these steps may be achieved by entering, for example,

```
python program.py
```

at a command-line interface (see *Introduction to command-line interfaces*, if needed), where `program.py` is the name of the Python program file. The Python system stops once the program has successfully finished, or if it detects an error in the program instructions. Program output might be displayed on the screen, or saved in computer files.

While Python programs are typically saved as text files so that they can be run and edited at will, one can also start the Python system (typically, by typing `python` in the system's command-line interface) and enter statements in the Python language manually. This interactive mode can be useful for exploring the Python language. From the Python interpreter, the `import` statement can be used to load and execute a Python source code. To run the program in a file named `program.py`, stored in the current directory, one may type

```
import program
```

To exit the Python system, one may type `exit()` or `quit()`

3.2.2 Python program structure

Python programs consist of simple statements that can be written in a single line, and compound statements occupying multiple lines. The Python system acts on program statements sequentially, whether they are taken from a text file or entered in interactive mode.

Successive simple statements share the same indentation. A compound statement involves lines that control its overall behavior, with the same indentation as preceding simple statements. Each of these controlling lines can be followed by sequences of associated statements, which are further indented.

This example of an "if-statement" demonstrates the use of indentation:

```
# Preceding the if-statement
print 'This is first'

# Beginning of the if-statement
if True:

   print 'This is second'

   # Nested if-statement
   if False:
      print 'This is never executed'
   else:
      print 'This is third'

# "Else" clause of the if-statement
else:

   print 'This is never executed'

# Exited from the if-statement
print 'This is fourth'
```

The lines beginning with `if` and `else` control the compound statement, and a second if-statement has been nested in the outer if-statement. The indentation indicates which statements are part of the compound statement, and which are not.

The character # indicates the beginning of a comment. The Python interpreter ignores all characters that follow on the same line. It is good programming practice to provide comments wherever they are likely to be helpful.

3.2.3 A note on variable assignment

Upon encountering a line with a variable name, followed by an assignment operator (an equal sign) and a value, as in:

```
a = 1
```

the Python system will create a variable with this name `a` and assign to it the given value `1`, if the variable has not yet been defined, or if a variable of this name has already been defined, it will be reassigned the value `1`.

The value on the right side is evaluated before the assignment is made. Therefore, it is legal to write statements such as:

```
a = a + n
```

which modifies the value of `a` according to the current value of `a` (variable names evaluate to their assigned values). In fact, this sort of statement is so common that there is an abbreviated form:

```
a += n
```

Note that Python is "case-sensitive", which means that capitalization matters. For example, `a` and `A` are completely distinct names.

3.2.4 Real numbers, floating-point numbers, and integers

Numerical calculations on computers are typically done with floating-point numbers, which are approximations to real numbers. The error in this approximation can sometimes produce significant "round-off" errors. Error-checking will be discussed in a later chapter.

Computers typically work with both floating-point numbers and integers. In Python, `1.0` and `1e0` (Python's notation for 1×10^0) are floating-point numbers, while `1` is an integer, and the two types of quantities are distinct. Integers take up less space in computer memory and mathematical operations on integers are faster than on floating-point numbers. However, programmers should be cautious, because using integers in numerical programs can sometimes lead to unexpected and undesirable results. For example, `1 / 2` might evaluate to `0` because division of one integer by another may be defined to produce another integer.

3.2.5 Strings

Variables can be associated with non-numerical data, such as strings. A string is an ordered sequence of characters (such as letters, digits, punctuation marks, and spaces), and is typically used to represent human-readable information. A literal string may be created in a few different ways; the simplest involves enclosing a sequence of characters within two single quotation marks, or two double-quotation marks, as in:

```
filename_root = 'ODE'
```

3.2.6 A note on functions

Python functions are sequences of Python instructions that are performed, or "called", as a unit when needed. Typically, a Python function is called by writing the function name, followed by one or more parameter values in parentheses. A function may yield, or "return", a result, which can be understood to take the place of the function call expression in the calling statement.

For example, one might have a function named `add` which takes two numbers as parameter values and returns their sum. If this function is called in the statement:

```
s = subtract(1,2)
```

the Python interpreter calls the function `subtract` with `1` and `2` as the first and second parameter values. The function `subtract` executes, performing a calculation, and returns the result `3`. The Python interpreter then evaluates the original assignment statement, but as if the function call expression had been replaced by its returned result:

```
s = -1
```

By recognizing frequently-used patterns of instructions as such and grouping them into functions, programmers can avoid repetition and the errors associated with repetition. Grouping instructions into functions with clearly-defined tasks also helps to keep programs organized, reducing programmer effort.

Before their first use, functions are defined in function definitions. The `subtract` function might have been defined in the following way:

```
def subtract(a,b):
    return a - b
```

The first line specifies the function name, and indicates that it is to be called with two parameter values. Unlike many other programming languages, Python does not ask the programmer to specify the type of argument expected (integer, floating point, etc.). The body of the function follows, indented relative to the first line. Any statements in this body are executed in order. Within this body, the argument variables specified in the first line of the function definition are initially assigned the parameter values given in the function call expression, in the order given (in the example above, the parameter order is not interchangeable). These variables exist only while this function is running, and they are distinct from any other variables of the same name that may appear elsewhere in the program.

A function finishes running if the end of the function body is reached, or if a "return statement" is encountered. As with other compound statements, the function definition ends just before the next statement with the same indentation as the first line of the function definition (or at the end of the file if there are no further statements). Return statements consist of the word `return`, followed by an expression is evaluated by the Python interpreter and returned to the calling statement as the function's "return value".

3.2.7 A brief discussion on object-oriented programming

Before proceeding further, "object-oriented" programming concepts should be discussed because of their importance in PyGDH and the underlying Python programming language. This section is meant to provide a brief introduction to these concepts.

"Object-oriented programming" is an approach to organizing computer programs, in which closely-associated data and computer instructions are organized into entities called "objects". Many readers may be more familiar with "procedural programming", in which computer instructions are organized into "procedures", "functions", or "subroutines" that manipulate data. Procedural programming is used within object-oriented programming, with objects providing an additional layer of organization by grouping instructions with related data. In the language of object-oriented programming, the data and computer instructions contained within an object are often called "members" of the object, and the instructions (in the form of functions) are called "methods".

One benefit of this approach is that the potentially complicated interactions among data and computer instructions can be contained within a single object. This is called "encapsulation", and this practice allows programmers to make use of an object's capabilities without detailed knowledge its internal details.

A "class" is a category, or "type", of object. A class definition tells the computer how instructions and data are grouped to form a particular type of object. A program might simultaneously make use of multiple objects that "belong" to the same class, which means that they all have the same basic structure as specified in the class definition. However, different objects belonging to the same class may contain different data and may possess different features in addition to those specified in their shared class definition.

A "derived" class is a class that is defined as a modification of an existing class; the existing class from which it is derived is called the "base" class. An object created according to the derived class is also an object of the base class

type, as it possesses the same basic qualities. In this way, the properties of the base class are said to be "inherited" by the derived class.

The structure of PyGDH fits nicely with these object-oriented concepts. PyGDH offers a flexible but simple approach to solving equations by helping users to express discretized equations in the form of a Python program. Users describe mathematical problems by defining derived classes, building on base classes defined by PyGDH. The structure inherited from the base classes helps users to express the problem descriptions.

Classes merely describe objects; in order to obtain numerical solutions with PyGDH, objects of the derived class types are then created (with storage space for data allocated in the process) and directed to obtain numerical solutions to specific problems.

Methods and object initialization

Methods are the computer instructions associated with objects, expressed as functions defined as part of a class. They typically appear as function definitions indented relative to their corresponding class statements. Among other things, they can serve as mechanisms for interacting with objects of that class.

All methods must be defined to accept at least one parameter, which by convention is called `self`. When a method accepts multiple parameters, the `self` parameter must be the first of these. Whenever a method is called, the object to which the method belongs is implicitly and automatically passed to the method as this first argument `self`. Values explicitly specified in method call expressions determine, in order, the values of the remaining argument variables in the method definition.

Here's an example of a class definition with a single method defined:

```
class Volume(pygdh.Volume):

    # Defining this method gives users an opportunity to define their own
    # mathematical operators, for later use in describing governing equations.
    def define_interpolants(self):

        # Operator for estimating first derivative at left boundary of volume
        self.dx_left = self.default_interpolant(-1.0, 1, 1)

        # Operator for estimating function value at the volume center
        self.interpolate = self.default_interpolant(0.0, 0, 2)

        # Operator for estimating first derivative at right boundary of volume
        self.dx_right = self.default_interpolant(1.0, 1, 1)
```

The first line of the class definition names the class, and subsequent indented lines describe the variables and methods that are class members. Every object of this class contains an independent set of variables with the specified names, and can be accessed through the specified methods.

The `__init__` method has special meaning to Python. This method is automatically called immediately after a new object of the class type is created, but before the new object is returned. This gives the programmer the opportunity to prepare the new object for use.

An object of a class is created by using an expression in which the class name is used like a function name, returning a new object of the class type. As with any other method, the (newly-created) object is itself implicitly passed as the `self` parameter, and the remaining argument variables are assigned the explicitly provided parameter values. For instance, in the example program, an object of the `ODE` class is created, and two parameters are explicitly passed:

```
problem = ODE([grid_obj], 1.0)
```

Object members and the dot operator

Now that an object of the `ODE` class has been created, one can interact with its members by using the "dot" operator. For example, writing:

```
problem.set_boundary_values()
```

instructs the Python interpreter to find and run the `set_boundary_values` method defined in the class to which `problem` belongs. Like all methods, `set_boundary_values` implicitly receives its containing object as its first argument, called `self`. This method was defined to accept only this implicit argument, so in the calling expression, its name is followed by an empty pair of parentheses containing no explicit parameter values. However, the parentheses are needed to indicate that the Python interpreter is to call the named method, rather than return the method (which is itself an Python object).

Data members are also accessible with the dot operator. For example, within a method, the assignment:

```
self.alpha = alpha
```

is understood in the following way: the expression `self.alpha` refers to the entity `alpha` that is contained within the entity `self`. As noted earlier, `self` refers to the object (of class `ODE` in the example program) to which the present method belongs. As the object `self` does not already possess a member named `alpha`, the Python interpreter automatically creates a new member variable with this name; this is typically how data members belonging to an object are created. Had `self.alpha` already existed, it would have been reassigned this new value.

Unlike in some other object-oriented languages, the name of an object member must always be used with the dot operator in order to access the object–Python does not offer a shortcut in definitions of methods belonging to the same class.

Scope

The parameter variable `alpha` in the following the `__init__` method in the example program

```
    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_problem' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, grid_sequence, alpha):

        # Save the argument value as an object member
        self.alpha = alpha

        # Prepare to run simulation on new 'Grid' object
        self.initialize_problem(grid_sequence)
```

is distinct from `self.alpha`. The `alpha` variable of the `__init__` routine is a "local" variable that is created when the `__init__` method is called and is discarded when the method execution has completed; it exists only while the method is running. In other words, the "scope" of the `alpha` variable (the part of the program in which the variable is accessible) is limited to the `__init__` method.

The `self.alpha` variable is a member of the object (named `self` within its own methods) and exists as long as the object itself exists. On a practical level, this means that multiple methods of the `self` object may use and make assignments to `self.alpha`, which becomes a useful tool for exchanging information among the short-lived function calls. In this case, `self.alpha` provides permanent storage for a user-specified simulation parameter.

3.2.8 Lists, arrays, and the index operator

Objects can contain data in the form of other objects. A list is a Python data structure that can hold an ordered sequence of arbitrary Python objects. A list may be modified–its elements can be added, removed, and redefined. Typically, PyGDH uses list objects as parameters when the user is allowed to specify an arbitrary number of related objects.

A literal list may be written by enclosing zero or more expressions (which evaluate to Python objects) in square brackets, with expressions separated by commas.

This defines an empty literal list:

```
domain_equations[0] = []
```

and this defines a literal list containing a single element, a string:

```
output_types = ['GNUPLOT']
```

Elements of a list may be accessed by writing the name of the variable associated with the list, followed by an index. An index can be as simple as an integer (or an expression evaluating to an integer) that describes the numerical position of an element in a list. The first element of a list has index `0`, the second has index `1`, and so on. This is called "zero-indexing". From the most recent example:

```
output_types[0]
```

has the string value `'GNUPLOT'`.

These index expressions can be used to either retrieve the values of list elements, or to set them. One could change the value of the first (and only) element of `output_types` by writing:

```
output_types[0] = 'HDF'
```

The index expressions can also be used for some other types of Python objects that contain other objects arranged in a specific order. In the example program:

```
y[0] = 0.0
```

references the first element of `y`, even though `y` in this particular case is a "NumPy array," and not a list. NumPy arrays can be multidimensional and are typically faster than lists at the cost of requiring their contents to be of a single type (such as floating point numbers).

Negative indices may also be used with lists and arrays. They indicate element positions, counting backward from the last element. The last element can be referenced with the index `-1`, the second to last with the index `-2`, and so on, as in:

```
y[-1] = 1.0
```

This can be a convenient because it offers a way to refer to the end of such an object without explicitly indicating how many elements the object contains.

3.2.9 Dictionaries

Sometimes, it is more natural to store objects associated with more general "keys" rather than numerical indices. A dictionary is an object that associates pairs of "keys" and "values". The values can be arbitrary Python objects, and many types of Python objects can be used as keys. While dictionaries are more flexible than lists, the elements of a list have a well-defined order, but the entries in a dictionary do not have a guaranteed order. Dictionaries are also slower than lists.

A literal dictionary can be created by enclosing key/value pairs of the form `key:value` within curly brackets, with pairs separated by commas. For example:

```
self.equation_scalar_counts = {self.ode: 1}
```

defines a dictionary with a single key/value pair, associating `self.ode` (an object representing the `ode` method belonging to the `self` object) with the integer `1`.

As with lists, a value in a dictionary may be referenced with an index expression. This is done by writing the dictionary name, followed by the corresponding key enclosed in square brackets. As each key in a dictionary must be unique, the dictionary returns a single value. For example:

```
self.equation_scalar_counts[self.ode]
```

has the value `1`. Similarly, one can associate a key in a dictionary with a new value by using the same form on the left side of an assignment statement, as in:

```
self.equation_scalar_counts[self.ode] = 2
```

If the key does not already exist in the dictionary, the Python interpreter automatically creates a new entry. If the key already exists, it is associated with the new value.

3.2.10 For loops and Boolean values

It is frequently useful to repeatedly run sequences of instructions, with each sequence differing only slightly from the last. This can often be achieved with a looping statement.

A Python for-loop executes its body (indented relative to the first line) once for each element of a sequence, with a specified variable assigned to successive element in successive loop iterations. For example, in:

```
for vol_number in range(1, self.grid[0].unit_count-1):
    vol = self.grid[0].unit_with_number[vol_number]
    self.unknown[0][0,vol.number] = True
```

the variable `vol_number` takes on successive element values from the value of `range(1, self.grid[0].self.unit_count-1)`. The `range` function, used in this way, returns a list of consecutive integers beginning with the first parameter `1` and ending at the last element, `self.grid[0].self.unit_count-1` in this case, minus one. Here, the first statement inside the for-loop assigns the element of `set.unit_with_number` (which happens to be an array) with the index `vol_number`, to the variable `vol`. In this way, the for-loop body successively acts on many elements of `self.unit_with_number`.

The quantities `True` and `False` are "Boolean" values and are frequently used to represent quantities that have one of two possible values, in particular the results from logical operations such as:

```
a == 1
```

where == is the equality operator, which is distinct from the assignment operator, =, described at the beginning of this chapter.

3.2.11 List comprehensions

There is frequently a need to use a for-loop to generate a new list based on an existing sequence, so a special abbreviated notation, called a "list comprehension", is provided for this purpose. This is used in the following example:

```
unit_classes=[ Volume for i in range(volume_count) ]
```

The entire expression is enclosed in square brackets. Each successive element of the new list is obtained by evaluating the first expression (here, `Volume`, which happens to be an Python object representing the `Volume` class definition, not shown here) with the loop variable (here `i`) assigned to successive values of the given sequence (here, obtained by evaluating `range(volume_count)`, which provides a list of successive integers from `0` to `volume_count-1`). In this particular case, the first expression is not dependent on the loop variable, so this list comprehension simply evaluates to a list with `volume_count` elements, each one associated with the object representing the `Volume` class definition.

3.2.12 Suggested exercises

1. Write a `for` statement (being sure to indent appropriately after the first line) which separately prints the integers from 7 through 16. Hint: a `print` statement such as `print i` prints the value of a variable `i` on the screen. Also, a blank line is used to signal the definition of a compound statement given in the command-line interface.
2. Write a `for` statement which separately prints the even integers from 4 through 18.
3. Write a list comprehension that returns the same result as `range(2,10)`
4. Write a function `print_elements` which separately prints each element contained in a list provided as a parameter. As with a `for` statement, be sure to indent appropriately after the first line. Call this function on the literal list `['zero', 1, 2.0]`.

3.3 An ODE with position as the independent variable

This chapter provides a detailed look at a Python program that uses PyGDH to solve a simple example problem, consisting of an ODE with given boundary values:

$$\frac{d^2y}{dx^2} = -\alpha^2 y(x)$$
$$y(0) = 0$$
$$y(1) = 1$$

The example problem is provided as `pygdh/examples/ODE/ODE.py` within the PyGDH archive. While there are many details, users do not need to write programs from scratch–program templates for typical mathematical problems are provided with PyGDH.

3.3.1 Program structure

Three major steps must be performed in any program that uses PyGDH to solve equations. First, one must define the spatial domain on which the governing equations are to be solved. Second, one must describe the rest of the structure of the mathematical problem, including the equations to be solved, boundary conditions, and, for a time-dependent problem, initial conditions. Finally, one must create an object that represents the specific problem to be solved (declaring the values of all model parameters, where needed), and direct this object to solve the mathematical problem and report results.

The statements

```
import pygdh
import numpy
import math
```

appear at the top of this file, and indicate that the Python interpreter will need access to PyGDH and NumPy (for matrix math) in order to run this program. These are "packages" or "modules" that, in effect, contain additional Python source code. By creating packages and modules, programmers simplify the process of making their source code available for use by other programmers.

Spatial domain definition

The spatial problem domains in PyGDH on which the equations are to be solved (a one-dimensional region in this example) are described with `Grid`-derived objects. Each of these objects describes a one-dimensional or two-dimensional spatial region on which equations are to be solved. For simplicity, PyGDH works only with regions based on rectangular grids in user-defined coordinate spaces.

The `Grid` class is defined by PyGDH, and includes many data structures and methods needed to represent a spatial problem domain. Objects should not be created directly from the `Grid` class defined by PyGDH. Instead, the `Grid` base class from the `pygdh` module should be specialized for the specific problem of interest by creating a derived class (named `Domain` in the example problem). The `Grid` class serves as a template and helps to organize problem-solving programs. The user-defined spatial domain class can be given any name that is acceptable to Python; in this example, it is called `Domain`.

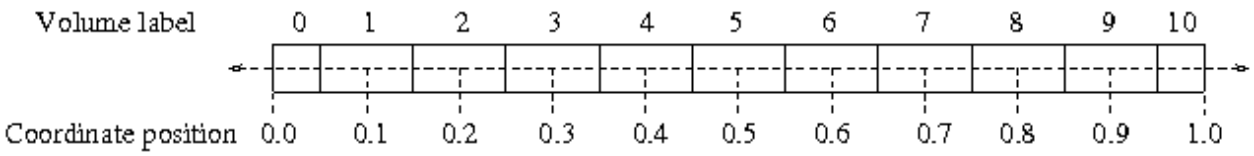
The `Domain` class definition begins with a line naming the class,

```
# Objects of this class describe the spatial domains on which problems will be
# solved.
class Domain(pygdh.Grid):
```

followed by indented text that describes various aspects of the class, and ends when another line with the same indentation as the first line is encountered, or otherwise upon encountering the end of the file. The expression `pygdh.Grid` in the first line of the class definition is a reference the `Grid` class contained within the `pygdh` package (which was made accessible by the earlier `import` statement). Its appearance in the class definition indicates that the new `Domain` class will be a derived class that builds on the base class `Grid`. Any object of the `Domain` class type is therefore also an object of the `Grid` class.

Users should define an `__init__` method. PyGDH itself does not access the `__init__` routines of `Grid`-derived objects, so users may customize these methods to the needs of the problem. However, the user-defined `__init__` routines in these derived classes must call the `initialize_grid` method defined by the `Grid` base class, which is needed to set up data structures for calculations.

The parameters supplied to the `initialize_grid` method establish the problem domain. This example applies the "finite volume method" on a one-dimensional spatial domain. From this perspective, a one-dimensional spatial domain consists of adjacent, non-overlapping, one-dimensional "volumes". Each of these volumes is associated with a unique coordinate value, which will be called the "position" of the volume. The set of these coordinate values defines the one-dimensional coordinate grid. For convenience, each volume is also given an integer label. The domain used in this example is illustrated below:



Please note that the volumes on the boundaries are half the width of the interior volumes. This is done so that the coordinate positions associated with these volumes are also the positions of the domain boundaries.

The call to `initialize_grid` can be written as

```
        self.initialize_grid(name,
                             # The independent variable will be named 'y'
                             field_names=['y'],
                             # The dependent variable will be named 'x',
                             # and 'y' will be computed at the specified number
                             # of equally-spaced values of 'x'
                             coordinate_values={
                                 'x': numpy.linspace(0.0, 1.0, volume_count)},
                             # Each unit in 'Domain' will be described by a
                             # 'Volume' object

                             unit_classes=[
                                 Volume for i in range(volume_count) ]

                             )
```

As the user-defined class has not also defined an `initialize_grid` method, the method defined by the base class is still accessible as `self.initialize_grid`; the derived class has "inherited" this property. This method is called with a number of parameters, some of which have the appearance of assignment statements. These "keyword arguments" have several benefits. They can improve clarity when passing multiple parameters, and they may be specified in any order, since the Python interpreter no longer needs to rely on the order in which parameter values are given in order to make initial assignments for argument variables.

Please note that this particular statement spans multiple lines in order to keep the line widths within 80 characters, which is encouraged in Python programming practice, as some users are limited to this screen width. Simple (as opposed to compound) Python statements are usually restricted to single lines. However, splitting a statement in this way is permitted when the line breaks occur within pairs of parentheses or square or curly brackets.

The first parameter to `initialize_grid` is a descriptive label for the new object, represented as a string. This will be used to label this domain in the program output, and can be used within the program for clarity.

The `field_names` parameter is a list of strings, which give descriptive names to the dependent variables for which solutions are to be obtained. As with any list object, the order in which these names are given implicitly assigns a unique integer label to each of these variables. These numerical labels will be the primary means by which the various variables will be later referenced. In this example, there is only one dependent variable, named `'y'`. Its position in the `field_names` parameter implicitly associates it with the index `0`.

The `coordinate_values` parameter is a dictionary, which associates descriptive names (in the form of strings) for independent spatial variables with a sequence of corresponding coordinate values. Here, `'x'` is the only key, and it is associated with the values obtained by evaluating:

```
numpy.linspace(0.0, 1.0, volume_count)
```

Here, the `linspace` function from the `numpy` package creates an array containing a sequence of numbers (the number of which is given by the value of `volume_count`) that are evenly-spaced between `0.0` and `1.0`, inclusive. This results in the sequence of volume positions illustrated in the diagram above.

A `Grid`-derived spatial domain is assembled out of smaller units. The `unit_classes` parameter is a sequence, with one element per unit making up the spatial domain. Each element refers to a class that defines the type of spatial unit object that will be associated with the corresponding volume in the domain. This allows for a great deal of flexibility, and in two-dimensional domains, this parameter further describes the physical layout of the domain. For the present example, the `Domain` class definition uses units described by a user-defined `Volume` class, which is a derived class of the `pygdh.Volume` class (in the `pygdh` package and so distinct from the `Volume` class defined in the present program), and starts with the line:

```
# Objects of this class are the units from which the spatial domain will be
# constructed.
class Volume(pygdh.Volume):
```

Like the `pygdh.Grid` class, the `pygdh.Volume` class is defined by PyGDH and intended as a template from which users can create derived classes with the same basic structures.

Mathematical problem definition

The description of the mathematical problem is expressed in a user-defined derived class of the `Problem` class, which is also provided by PyGDH. Here, the derived class has been given the user-defined name `ODE`. Any object created based on the `ODE` class is also an object of the `Problem` class. A user should define an `__init__` method for this derived class, as in this example

```
    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_problem' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, grid_sequence, alpha):

        # Save the argument value as an object member
        self.alpha = alpha

        # Prepare to run simulation on new 'Grid' object
        self.initialize_problem(grid_sequence)
```

As with the user-defined `Domain` class, PyGDH does not access `__init__` in the present class definition, so the method may be customized for the needs of the problem. However, it should call the `initialize_problem` method defined by the `Problem` base class. This method must be provided with a single parameter, a sequence of user-defined `Grid`-derived objects that describe the spatial domains on which equations are to be solved. The order in which these objects appear in this sequence implicitly associates each object with an integer label, with the first having label `0`, the second having label `1`, and so on.

To support code readability and reuse, one can also use the `name` parameters specified in the calls to the `Grid.initialize_grid` method as keys in the `Problem.grid_number_with_name` and `Problem.grid_with_name` dictionaries in order to retrieve corresponding `Grid` sequence numbers and `Grid` objects.

Setting prescribed value boundary conditions

The spatial domain of this example problem was defined in terms of the coordinate `'x'` and runs from `0.0` to `1.0`. Typically, information about the solution at the domain boundaries is needed in order to completely specify the problem to be solved. The boundary conditions for this example are

$$y(0) = 0$$
$$y(1) = 1$$

PyGDH allows the user to strictly enforce solution values on a boundary. This is accomplished by defining a method in the `Problem`-derived class named `set_boundary_values`, which takes only the `self` argument:

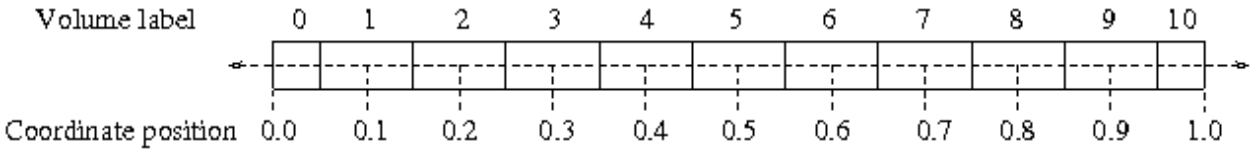
```
    # This method is used to define boundary conditions in which the solution
    # value is specified. This must be consistent with the contents of the
```

(continues on next page)

(continued from previous page)

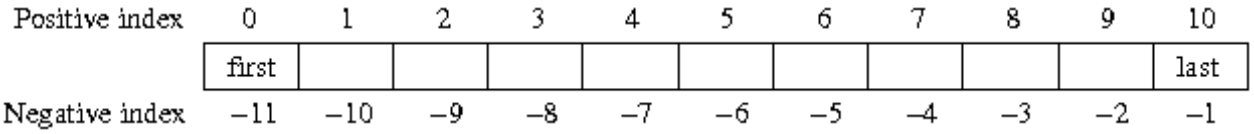
```
    # 'declare_unknowns' methods defined in the corresponding 'Grid' objects.
    def set_boundary_values(self):

        # Defined for clarity; there is only one 'Grid' object, with index 0,
        # and one solution field 'y', with index 0
        y = self.grid[0].field[0][0]

        # "Left" domain boundary
        y[0] = 0.0

        # "Right" domain boundary
        y[-1] = 1.0
```

The first line defines a temporary variable for clarity, with the same name as the descriptive label given to `initialize_grid`. The complicated-looking expression will be explained in the next chapter; for now, it is enough to know that `'y'` is a one-dimensional array that contains one value for each volume element in the domain. The first (0th) element of this array corresponds to the volume at the boundary `0.0`, and the last element corresponds to the volume at the boundary `1.0`, in the same diagram:



The 0th element of the NumPy array `y` is referenced by following the variable name with the integer label `0` in square brackets. Since this is done on the left side of an assignment statement, the specified array element is given the value of the expression on the right side of the assignment statement (`0.0`).

The second assignment is similar, and references the end of the array by using the negative index notation, which provides an alternative method of labeling the volumes:



In situations such as this, it is much more convenient to use these negative indices, rather than having to be aware of the total number of elements in the array when using a non-negative index.

Declaring unknowns

PyGDH must be made aware of the locations in the domain at which the solution is unknown and therefore should be computed. This should be done within a user-defined method named `declare_unknowns` in the `Problem`-derived object. This method must take only `self` as an argument:

```
    # This method is required. It informs PyGDH of the unknown quantities for
    # for which it must solve.
    def declare_unknowns(self):

        # "Left" boundary
        vol = self.grid[0].unit_with_number[0]
        self.unknown[0][0,vol.number] = False
```

(continues on next page)

(continued from previous page)

```
        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            vol = self.grid[0].unit_with_number[vol_number]
            self.unknown[0][0,vol.number] = True

        # "Right" boundary
        vol = self.grid[0].unit_with_number[-1]
        self.unknown[0][0,vol.number] = False
```

Here, the local variable `vol` is defined (and re-defined) for clarity.

The associated expressions involve both the dot operator and the index operator. The operations associate from left-to-right, so the first definition is equivalent to:

```
vol = ((self.grid[0]).unit_with_number)[0]
```

That is, the newly-created local variable `vol` is given the value of the 0th element of the `unit_with_number` member of the `self` object.

The member `unit_with_number` is a sequence that is automatically defined for any `Grid`-derived object. Each element of this sequence is a `Volume` object that represents a single volume in the domain. The first `Volume` object in this list is associated with the most negative value (and therefore one domain boundary) in the appropriate entry of the `coordinate_values` argument to `initialize_grid`. The last `Volume` object in `unit_with_number` is associated with the other domain boundary, and the most positive value in the `coordinate_values` argument.

Each `Grid` object contains a list named `unknown`, each element of which corresponds, in order, to corresponding `Grid` sequence number. Each of these elements is itself an NumPy array, with the first index corresponding to the position in the list of variables named by the `field_names` argument to `initialize_grid`, and the second index corresponding to the corresponding position in `unit_with_number`. In this example, only one field variable was given, with the descriptive string `'y'`. As the first and only element in the list of field variables, it was implicitly assigned the label `0`. This same label is used to reference this variable throughout the data structures used by PyGDH, including in the `unknown` lists. Therefore, `unknown[0]` corresponds to the status of the field named `'y'` on the `Volume` of interest.

Each element of `unknown` is given the value `True` if the corresponding field value is unknown, or `False` otherwise. The `unknown` value associated with a variable on a given `Volume` should only be `False` if the `set_boundary_values` method provides a value for that variable on that particular `Volume`. In this particular case, boundary values were given on both ends of the domain, so PyGDH should not attempt to solve for values at these locations.

By default, the `unknown` element corresponding to every field variable is set to `True` on every `Volume` object, so it is only actually necessary to specify where it should instead be `False`. However, for clarity, the assignments are made here for every `Volume`.

The `unit_count` member of objects belonging to the `Grid` class is automatically determined by `initialize_grid`, and gives the total number of spatial units associated with the `Grid`-derived object. This is the total number of elements in `unit_with_number` sequence, with numbers `0` through `unit_count-1`. The for-loop here is written to visit all but the first and last elements in `unit_with_number`, which involves numbers `1` through `unit_count-2`. This list of successive values is produced by the expression `range(1, self.unit_count-1)`.

Describing equation dimensions

PyGDH must be made aware of how many scalar equations are represented by each user-defined governing equation method. This is done in a required method named `set_equation_scalar_counts`, in which the user creates

a new dictionary, named `equation_scalar_counts`, as a member of the `Problem`-derived object. This dictionary associates methods representing governing equations, with integers indicating the number of scalar equations that they represent. Functions are themselves objects, and the name of a function, written alone, is a reference to the function object itself. In this particular case, there will be only one governing equation method, named `ode` and defined in the `ODE` class, and it will be associated with a single scalar equation. For the present, it is sufficient to note that this method is a discretized representation of the given governing equation.

```
    # This is a required method that notifies PyGDH about the number of scalar
    # values returned by each governing equation method.
    def set_equation_scalar_counts(self):

        # This is a dictionary associating governing equation methods with
        # the number of scalar values that they return
        self.equation_scalar_counts = {self.ode: 1}
```

In more complicated situations, it might be convenient to have a method that represents a vector equation, in which case the associated number of scalar equations must reflect the vectorial nature of the returned value.

Associating equations with spatial units

Just as it was necessary to specify the unknown field variables associated with each `Volume`, it is necessary to specify which equations govern the behavior of the variables on each `Volume`. This is done in a user-defined method of the `Problem`-derived class. This method must be named `assign_equations`, and it must take only `self` as an argument.

In the example, one can see that the structure of this method is reminiscent of the `declare_unknowns` method of the `Grid`-derived class.

```
    # This is a required method that notifies PyGDH about the methods that
    # describe governing equations, and indicates where in the domain the
    # equations apply. This must be consistent with the 'declare_unknowns'
    # methods of the corresponding 'Grid' objects.
    def assign_equations(self):

        # Defined for clarity, equations to be defined on the only Grid,
        # with index 0
        domain_equations = self.equations[0]

        # "Left" domain boundary
        domain_equations[0] = []

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            domain_equations[vol_number] = [self.ode]

        # "Right" domain boundary
        domain_equations[-1] = []
```

This method is part of the `Problem` class because the methods representing the governing equations are also part of the `Problem` class, which simplifies interactions among multiple `Grid` objects.

The automatically-created `equations` member of the `Problem`-derived object is a sequence, with one element for each `Grid`-derived object supplied to the `initialize_problem` method. The elements of the `equations` sequence in turn correspond to the `Grid`-derived objects in the list provided to `initialize_problem`, and are numbered in the same sequence. In this example, there is only one `Grid` object, describing the spatial domain and

having the index `0`. Here, a local variable `domain_equations` is used to clarify references to the corresponding part of the `equations` member.

The elements of the `equations` sequence are themselves sequences, with one for each unit object associated with the corresponding `Grid` object, and ordered in the same way as the `unit_with_number` sequence of the same `Grid` object. Each element of these sequences is a list of references to functions, each of which describes one or more governing equations for the corresponding unit. In this example, the first and last units, with indices `0` and `-1`, have values that are known from the `set_boundary_values` method (and as noted in the `declare_unknowns` method), and so these are associated with empty lists. Dependent variable behavior on all other units, on the interior of the domain, is governed by the `ode` method, which corresponds to a single scalar equation, as noted in `equation_scalar_counts`.

Since this method is a member of the `Problem` object, and not a `Grid` object, the `self` variable here refers to the `Problem` object. In order to provide convenient access to `Grid` object(s), a `Problem`-derived object automatically defines a member named `grid`, which is a list of associated `Grid`-derived objects. This list object is identical in content and order to the argument originally passed to the `initialize_problem` method. In this example, there is only one `Grid` object, so it is implicitly assigned the index `0`. This number is now used to reference the one `Grid` object in the list `grid`. Then, as before, the `unit_count` member of the `Grid` object may be accessed directly.

Solver specification and execution

Now that the mathematical description is complete, one may create an object of the `ODE` class, automatically invoking the `ODE.__init__` method to initialize data structures and set the simulation parameters of interest. This object is then instructed to solve the mathematical problem that it describes and to report the results:

```
# Create the 'Grid' object representing the problem domain
unit_count = 11

grid_obj = Domain('domain',unit_count)

# Create an object instance, using 11 volumes and setting alpha = 1.0
problem = ODE([grid_obj], 1.0)

# The names of all output files will begin with this string
filename_root = 'ODE'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

# Calculate solution and write output to file
problem.solve_time_independent_system(filename_root, output_types)
```

The `solve_time_independent_system` method, defined in the `Problem` class, accepts a descriptive string from which it will determine the output filename(s), as well as a list of strings indicating the file formats in which the output should be written. The method then solves the discretized problem and writes the result to the output file(s).

At the computer's command-line interface, if `ODE.py` is in the present directory, the program can be run by typing:

```
python ODE.py
```

In this example, the resulting numerical solution is automatically saved a human-readable text file named `ODE_domain.gnuplot`. The contents can then be further processed or plotted. A few different file formats are available, and the present format is a simple one that is understandable by many different plotting programs, including GNUPLOT, a common plotting program on GNU/Linux systems.

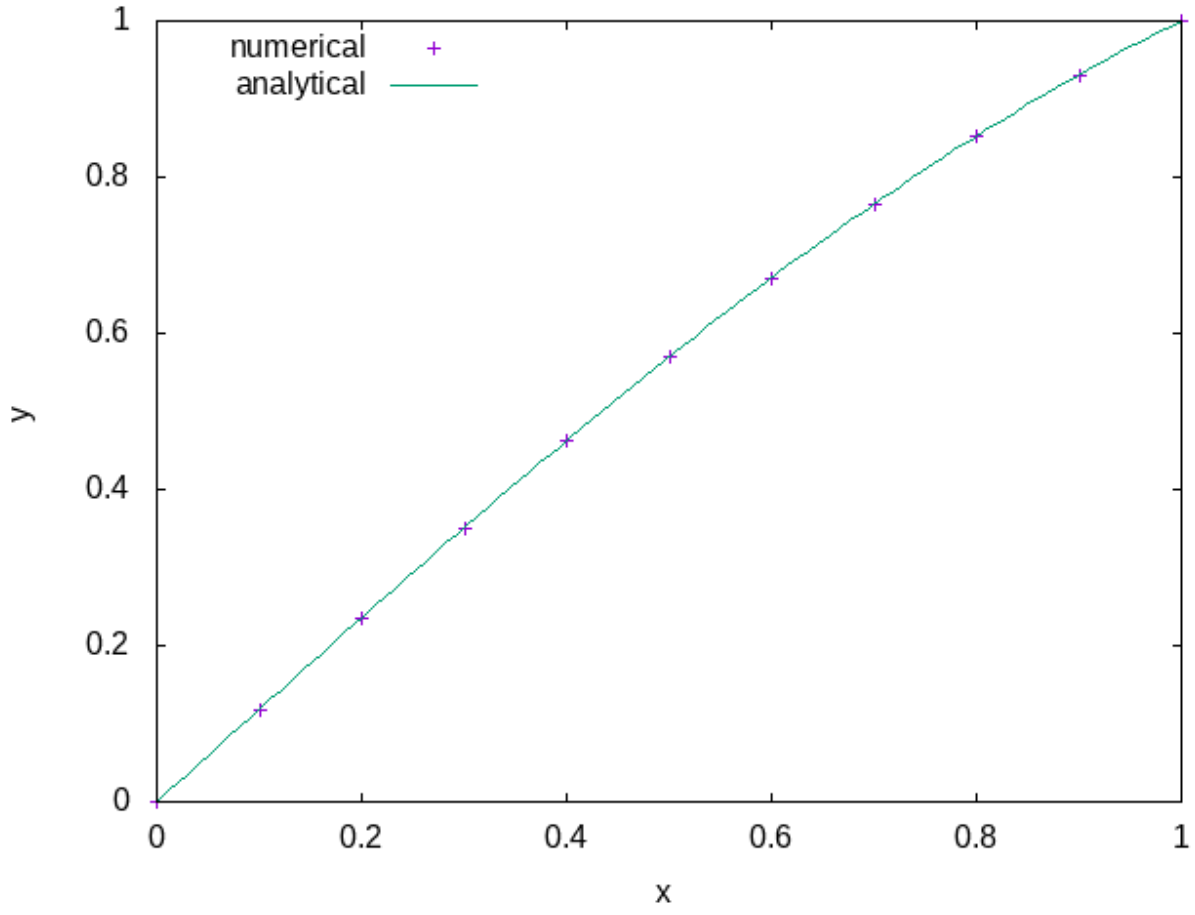
GNUPLOT can be provided with a sequence of commands stored in a text file, which guide the program to produce plots. Here, the following GNUPLOT script:

```
set term png
set output 'ODE.png'
set key left top
set xlabel 'x'
set ylabel 'y'
plot alpha=1., 'ODE_domain.gnuplot' title 'numerical', sin(alpha*x)/sin(alpha) title
↪'analytical'
set output
```

is stored in a file named `ODE.gpl` and executed by typing:

```
gnuplot ODE.gpl
```

at the computer's command-line interface. This produces the following plot of the numerical solution along with the exact solution:



where the exact solution for this problem is given by

$$y(x) = \frac{\sin(\alpha x)}{\sin(\alpha)} \tag{3.1}$$

for $\alpha = 1$.

Please note that on some systems, the `png` "terminal" (specifying the output format) will not be available, and another

terminal must be used. For example, the first line might be replaced with `set terminal postscript color`. Running GNUPLOT in interactive mode by typing `gnuplot` at the system's command-line interface, then typing `set terminal` at the `gnuplot>` prompt will provide a list of available options. One may type `exit` or `quit` to leave GNUPLOT and return to the system's command-line interface.

3.3.2 Suggested exercises

1. Run the code and plot the results.
2. Although the source code has not been explained in detail, try to read through the comments (lines starting with the # character) and try to modify the program to instead use $\alpha = 10$. Do your results continue to agree well with the analytical solution (3.1)? If you are using GNUPLOT, a text editor may be used to modify the GNUPLOT script to plot the appropriate analytical solution values.
3. Modify the program to use a domain divided into 6 volumes. Do your results continue to agree well with the analytical solution? What if 21 volumes are used?
4. Change the boundary values, so that

$$y(0) = 1$$
$$y(1) = 0$$

Do you get what you expect?

3.4 Solving an ODE with an initial condition

Using PyGDH to solve an ODE with time as the independent variable is typically simpler than using it to solve an ODE with position as the independent variable. However, this tutorial addressed boundary-value problems earlier because the motivation behind much of the design of PyGDH is clearer from the perspective of solving boundary value problems. Solving an ODE with time as the independent variable reuses the same machinery, much of which might appear to be excessive without the broader context.

An ODE with no spatial dependence may be solved by creating a `Grid` object containing only a single `Volume`. The spatial aspects of this `Volume` are not relevant, but its data structures will be used to store solution values. Time-discretized equations describing the evolution of these values are then specified, and these equations are then solved for a series of timesteps, giving numerical solutions to the discretized equations. Unlike with discretization of the spatial domain, as described in the previous chapter, the user is responsible for explicitly providing the time-discretized equations.

A future chapter discusses the simultaneous solution of problems that make use of multiple `Grid` objects. This approach can be used to simultaneously solve PDEs and ODEs with time as the independent variable.

This chapter will consider the equation

$$\frac{dy}{dt} = -\alpha y$$

with initial condition

$$y(0) = 1$$

3.4.1 "Spatial" domain

Since this problem has no spatial dependence, the `pygdh.Volume` class may be used directly–there is no need to first create a derived class because there is no need to perform interpolation. Nor is it necessary to declare independent

spatial variables to `initialize_grid` in a problem with no spatial dependence. The `__init__` routine of the `Grid`-derived object can then be simply defined as:

```
def __init__(self,name):

    # This 'Grid' object only contains a single 'Volume' object
    self.initialize_grid(name, field_names=['y'],
                         # As there is no spatial dependence, it is not
                         # necessary to declare independent spatial
                         # variables
                         unit_classes=[pygdh.Volume],
                         stored_solution_count=2)
```

3.4.2 Discretization of a time-dependent equation

This section will discuss a simple approach to discretization of an ODE in time. The discretized equation obtained here will suggest additional machinery needed to obtain a numerical solution.

Computer solutions of time-dependent problems typically involve finding approximate numerical solutions at successive moments in time, called "timesteps". Taken together, these approximately describe the time-dependent solution. Typically, the quality of the approximation can be improved by using smaller intervals between timesteps (the "timestep size").

Under these "time discretization" schemes, time is not a continuous variable, so quantities such as time derivatives are approximated by comparing solutions at different timesteps. For this reason, solutions calculated at previous timesteps are generally needed in order to compute approximate solutions to time-dependent problems.

For simplicity, this example will use a simple first-order backward difference in time (note that there are many well-known schemes that are both more accurate and more complicated). Under this scheme, the differential equation

$$\frac{\partial y}{\partial t}(t) = g(t, y(t))$$

is approximated by

$$\frac{y(t_0 + \Delta t) - y(t_0)}{\Delta t} \approx g(t_0 + \Delta t, y(t_0 + \Delta t))$$

This is called a first-order scheme because the error in approximating the time derivative will be proportional to the size of the timestep to the first power, when the timestep is sufficiently small.

Moving all terms to one side gives an expression in the form expected by PyGDH:

$$F(t, y) = \frac{y(t_0 + \Delta t) - y(t_0)}{\Delta t} - g(t_0 + \Delta t, y(t_0 + \Delta t)) \approx 0.$$

Since there is no spatial dependence in this problem, it is unnecessary to integrate this expression in order to obtain a finite-volume form, as was done in the previous chapter.

From this, it is apparent that it will be necessary to incorporate information at two different points in time, t_0 and $t_0 + \Delta t$. PyGDH must be notified that it must keep track of both the present solution and the solution at the previous timestep. This is declared to PyGDH through the parameter `stored_solution_count` to the `initialize_problem` method. By default, PyGDH only tracks the solution at the present time, but in this case, PyGDH must provide storage for one previous solution:

```
    def __init__(self, alpha):

        # Save the argument value as an object member
        self.alpha = alpha

        grid_obj = FalseDomain('false_domain')

        # Initialize data structures, among other things
        self.initialize_problem([grid_obj],stored_solution_count=2)
```

As before, the discretized equation is described to PyGDH by defining a method:

```
    def ode_in_time(self, vol, residual):

        # There is only one solution field, with index 0, and with one
        # ``Volume``, with label 0
        y = self.grid[0].field[0][0,0]
        y_1 = self.grid[0].field[-1][0,0]

        residual[0] = (y - y_1)*self.inverse_timestep_size + self.alpha*y
```

In this method, local variables are defined for clarity. The data structures that they reference will be discussed now in detail.

Solution data storage

The definition of the local variable `y` references a number of entities. As described earlier, the dot and index operators associate from left-to-right, so the definition is equivalent to

```
y = ( ( ( ( self.grid ) [0] ) . field ) [0] ) [0,0]
```

Reading this from the innermost parentheses to the outermost parentheses, this can be interpreted in the following way:

1. Look at the `grid` member of `self` (a `Problem`-derived object). The `grid` member is a list of `Grid` objects, identical in composition and order to the list passed to the `initialize_problem` method called in `ODE_in_time.__init__`. In this example, only one `Grid` object is used.
2. Look at this first (0th), and only, element of `self.grid`, which is a `Grid` object. As there is only one `Grid` object used in this problem, as determined by the parameter of `initialize_problem`, it has the index `0`.
3. Look at the `field` member of the `Grid` object `self.grid[0]`. The `field` member is a "double-ended queue", or "deque", which is a sequence that may be accessed just as a list is accessed. It is faster for certain operations, at the cost of requiring more memory. The `field` member has one entry for every stored numerical solution; in this case, there are two, as indicated by the `past_solution_count` parameter to `initialize_problem`. These correspond to solutions at the present and previous timestep.
4. Look at the value of the first (0th) element of the deque `self.grid[0].field`. PyGDH automatically manages the `field` object, so that the index `0` always corresponds to the present timestep, and negative indices can be used to intuitively access past solutions; `-1` corresponds to the previous timestep, and so on. In this case, the solution at the present timestep is desired. Each element of `self.grid[0].field`, such as the one presently being considered, is a two-dimensional array of solution values.
5. Look at the element `[0,0]` in the two-dimensional solution array `self.grid[0].field[0]`. The element at position `(i,j)` in a two-dimensional array (named, for example, `M`) can be accessed as `M[i,j]`, and as

usual, index numbering begins at `0`. The first array index indicates the independent variable, numbered according to its position in the `field_names` keyword parameter passed to `initialize_grid`. In this example, there is only one, with the name `y`. The second index to the two-dimensional array indicates the `Volume` corresponding to the spatial position at which the selected variable has the stored value. In this example, the `Grid` object has only one `Volume`, which does not have spatial information, but rather is used only to provide storage for solution values within the framework provided by PyGDH.

6. Associate the local variable named `y` with the value of `self.grid[0].field[0][0,0]`, the solution value at the present time.

The local variable `y_1` is defined similarly, except that it accesses the `-1` element of the `field` deque, so that `y_1` is associated with the solution value from the previous timestep.

Finally, the `inverse_timestep_size` member is automatically calculated by PyGDH from the timestep size provided to `solve_time_dependent_system`. Multiplication by the inverse timestep size is preferable to division by the timestep size because floating-point multiplication is the faster operation. Alternatively, one could instead multiply the entire expression by the `timestep_size` member, also automatically defined by PyGDH.

3.4.3 Declaring equations

As usual, PyGDH must be made aware of any method describing a discretized governing equations. The governing equation in this example describes a single solution variable associated with a single `Volume` object on a single `Grid`: .. must be .. This discretized equation describes the evolution of a single dependent variable associated with a single `Volume` object:

```
    def ode_in_time(self, vol, residual):

        # There is only one solution field, with index 0, and with one
        # ``Volume``, with label 0
        y = self.grid[0].field[0][0,0]
        y_1 = self.grid[0].field[-1][0,0]

        residual[0] = (y - y_1)*self.inverse_timestep_size + self.alpha*y
```

3.4.4 Initial condition

Since the example problem is time-dependent, the initial condition must be provided. This must be done in a user-defined method named `set_initial_conditions` within the definition of `Problem`-derived class. This method must have `self` as its only parameter.

The `set_initial_conditions` methods should set the values for the present solution, with index `0` in the `field` deque. It should set the values of the dependent variables at all locations at which they are unknown.

The initial value of this problem is set for the single solution value associated with the single `Volume`. As before, this value is accessed directly by using both array indices.

```
    def set_initial_conditions(self):

        # Set the value of the single dependent variable directly
        self.grid[0].field[0][0,0] = 1.0
```

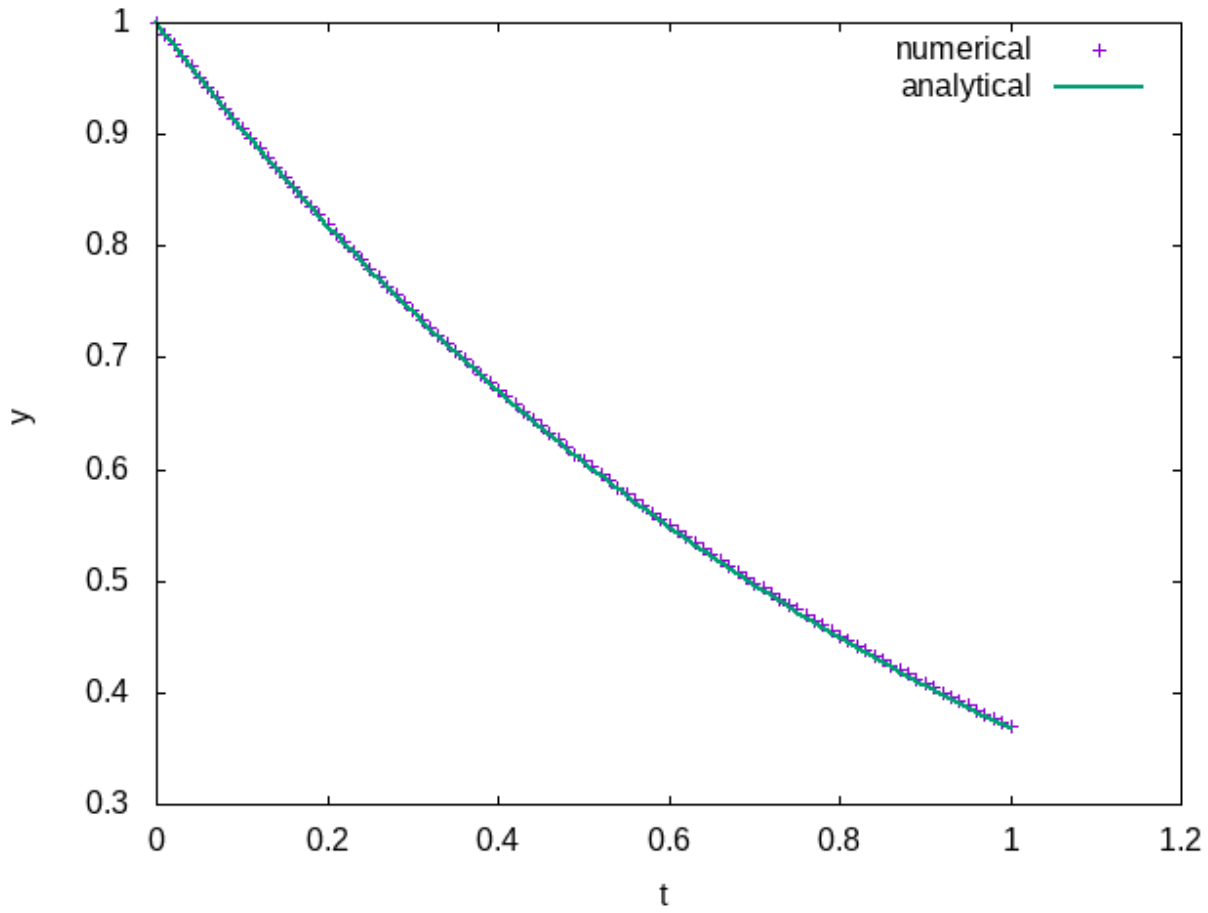
3.4.5 Solving a time-dependent problem

As in the first example, an object of the `Problem`-derived class is created, and it is directed to obtain solutions. For a time-dependent system, a solver method that recognizes the time-dependent nature of this problem must be used and information about the simulation interval must be provided.

```
# Create an object instance, setting alpha = 1.0
problem = ODE_in_time(1.0)

number_of_timesteps = 100
timestep_size = 0.01

# The names of all output files will begin with this string
filename_root = 'ODE_in_time'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

# Calculate solution and write output to file
problem.solve_time_dependent_system(filename_root, output_types,
                                    timestep_count=number_of_timesteps,
                                    timestep_size=timestep_size)
```

As a first-order timestepping scheme was used, a relatively small timestep size is needed to obtain a faithful numerical solution. The results are plotted below against the analytical solution,

$$y(t) = \exp(-\alpha t)$$



The plot above was generated with the following GNUPLOT script:

```
set term png
set output 'ODE_in_time.png'
set ylabel 'y'
set xlabel 't'
plot 'ODE_in_time_false_domain.gnuplot' using 1:2 title 'numerical', exp(-x) title
↪'analytical' lw 2
set output
```

3.4.6 Suggested exercises

1. Halve and double the timestep size to observe the effect of truncation error.
2. Solve the same problem, but use a second-order trapezoidal rule, for which a differential equation of the form

$$\frac{dy}{dt} = f(t, y)$$

has the discretized approximate form

$$\frac{y(t + \Delta t) - y(t)}{\Delta t} = \frac{f(t, y(t)) + f(t + \Delta t, y(t + \Delta t))}{2} + \mathcal{O}(\Delta t^2)$$

The method describing the governing equation should evaluate and store

$$y(t + \Delta t) - y(t) - \frac{f(t, y(t)) + f(t + \Delta t, y(t + \Delta t))}{2} \Delta t,$$

which asks PyGDH to find $y(t + \Delta t)$ so that the above expression evaluates to zero.

Confirm that the result is consistent with the analytical solution. Halve the timestep size to observe the effect of truncation error.

3. Solve the same problem, but using a second-order Runge-Kutta scheme. Traditionally, for a differential equation of the form

$$\frac{dy}{dt} = f(t, y),$$

a second-order Runge-Kutta scheme is typically expressed in a form such as:

$$\begin{aligned} k_1 &= f(t, y)\Delta t \\ k_2 &= f(t + \Delta t/2, y(t) + k_1/2)\Delta t \\ y(t + \Delta t) &= y(t) + k_2 + \mathcal{O}(\Delta t^3) \end{aligned}$$

For implementation with PyGDH, this can be arranged into a form such as

$$y(t + \Delta t) - y(t) - k_2 = 0.$$

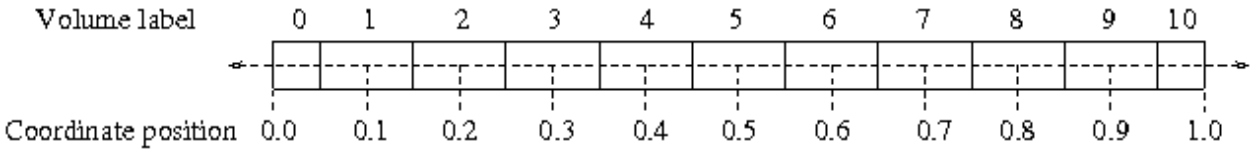
The intermediate results k_1 and k_2 can be stored in locally-defined variables for clarity.

Confirm that the result is consistent with the analytical solution. Halve the timestep size to observe the effect of truncation error.

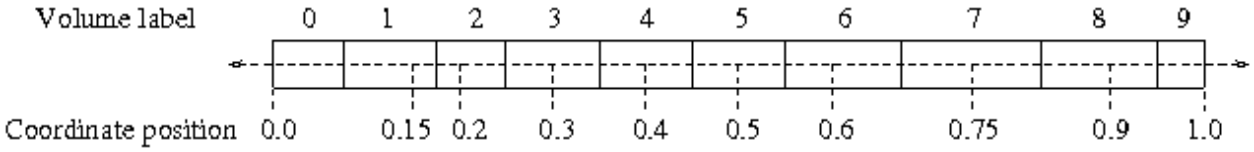
3.5 Discretization by the finite volume method in one dimension

PyGDH assists the user in obtaining numerical solutions to differential equations (or systems of differential equations) defined on some domain. Currently, PyGDH is only designed to support the finite volume method. In this approach, the problem domain is divided into a finite number of non-overlapping volumes that, taken together, completely fill the domain. One may then seek an approximate solution in which a single value per volume is used to characterize the behavior of any one solution variable.

In the previous chapter, a diagram of a typical one-dimensional domain was shown:



More generally, the grid can have irregular spacing, as in:



Each volume is associated with a unique coordinate value, stored as the `coordinate` member of the corresponding `Volume` object. These coordinate values were given in the `coordinate_values` parameter value when the containing `Grid` object was created. As shown, volume boundaries are located midway between these coordinate values. Domain boundaries also coincide with volume boundaries for the volumes adjacent to the domain boundaries.

3.5.1 Methods representing discretized equations

Governing equations are represented by user-defined methods defined in a `Problem`-derived class, such as in the example:

```
    # This method represents a governing equation and is named and defined by
    # the user. It must accept three particular parameters and must store
    # results in the data structure associated with the last parameter.
    def ode(self, vol, residual):

        # Defined for clarity; there is only one 'Grid' object, with index 0,
        # and one solution field 'y', with index 0
        y = self.grid[0].field[0][0]

        # FVM representation of governing equation
        residual[0] = (vol.dx_right(y) - vol.dx_left(y)
                       + vol.interpolate(y) * self.alpha**2 * vol.volume)
```

The discretized equations are first expressed in the form $F(x, y(x)) = 0$, where x represents the independent variable(s), and $y(x)$ represents the unknown dependent variable(s). Then the corresponding discretized equation method must then compute the value of $F(x, y(x))$ and store the appropriate number of scalar results (which must be consistent with the corresponding entry in `equation_scalar_count`) as the elements of sequence passed as the second explicit parameter (called `residual` here).

As seen in the description of the user-defined `assign_equations` method in the previous chapter, each `Volume` object is associated with methods that describe the equations that govern the solution values over the corresponding volume. This approach provides the programmer with great flexibility, although it is inconvenient to define separate equation methods for each `Volume`. Instead, PyGDH is designed in a way that makes it relatively simple to describe discretized governing equations without the explicit use of discretization formulas. This makes it possible to define a single governing equation method that can be applied to each `Volume`, regardless of location.

A governing equation method must accept a unit object as its first explicit parameter. The unit object provides location information and typically defines operators that may be used by the governing equation method in place of explicit discretization formulas.

Discretization of a time-independent equation by the finite volume method

Under the finite volume method, discretized equations are obtained by integration over volumes of finite size. Consider the governing equation from the first example program, describing a one-dimensional domain:

$$\frac{d^2y}{dx^2} = -\alpha^2 y(x).$$

This is rearranged into the form $f(x, y(x)) = 0$, where x represents the independent variable, and $y(x)$ is the solution:

$$f(x, y(x)) = \frac{d^2y}{dx^2} + \alpha^2 y(x) = 0.$$

One then integrates both sides of the equation over a region with lower bound a and upper bound b, to obtain, in general:

$$\int_a^b f(x, y(x))\, dx = \int_a^b 0\, dx = 0$$
$$F(x, y(x)) = 0$$

where

$$F(x, y(x)) = \int_a^b f(x, y(x))\, dx$$

For the first example, this is:

$$F(x, y(x)) = \int_a^b \left(\frac{d^2y}{dx^2} + \alpha^2 y(x) \right) \, dx = 0$$
$$F(x, y(x)) = \left[\frac{dy}{dx} \right]_a^b + \alpha^2 \int_a^b y(x) \, dx = 0.$$

A method representing a discretized equation to PyGDH computes an approximation to the function $F(x, y(x))$ and stores the results. As each `Volume` is associated with only a single value of each solution field variable, interpolation over these values is typically needed to approximate integrals.

The integrals can be approximated in a number of ways. Interpolants of various orders may be used over volumes to approximate the spatial variation of fields, and the values of different factors within a single integrand may be obtained by different interpolation formulas–in the simplest case, a factor might be approximated by a constant value over the volume. Various methods of numerical integration may be used. The objects defined by PyGDH assists the user in representing the discretized equations, but the method of discretization is left completely to the user, and has consequences for solution speed, accuracy, and stability.

In the example, the integrand is replaced by a constant, its average value over the volume (indicated by angle brackets). This yields

$$F(x, y(x)) = \left[\frac{dy}{dx} \right]_a^b + \alpha^2 \langle y \rangle (b - a) = 0$$

As a very simple approximation, the average value of the integral will be replaced by the value of the integral at the center of the volume:

$$F(x, y(x)) \approx \left[\frac{dy}{dx} \right]_a^b + \alpha^2 y_{\text{center}} (b - a) \approx 0$$

Representation as a method

This final form is then represented as Python source code in the `ode` method. As required by PyGDH, this method accepts a unit object of interest (here, a `Volume`) as the first explicit argument `vol`, and an array `residual`. The user-defined operators computed for each `Volume` in the `define_interpolants` method will be used to obtain the needed values of the independent variable and its derivatives. The governing equation method is then a direct translation of the approximate form for $F(x, y(x))$ into a Python expression:

```
# This method represents a governing equation and is named and defined by
# the user. It must accept three particular parameters and must store
# results in the data structure associated with the last parameter.
def ode(self, vol, residual):

    # Defined for clarity; there is only one 'Grid' object, with index 0,
    # and one solution field 'y', with index 0
    y = self.grid[0].field[0][0]

    # FVM representation of governing equation
    residual[0] = (vol.dx_right(y) - vol.dx_left(y)
                   + vol.interpolate(y) * self.alpha**2 * vol.volume)
```

The local variable `y` is created for clarity. The data structure that it references will be discussed in the next chapter, but `y` itself is a one-dimensional array that holds one value of the independent variable for every `Volume` in the domain. This is the form of the required input for the interpolation routines. These were defined and given the names `dx_right`, `dx_left`, and `interpolate` in `Volume.define_interpolants`. As described in the next

section, these operators approximately compute the derivatives at the right and left volume boundaries, and the value of the solution variable at the center of the volume.

The `self.alpha` variable, as discussed in an earlier chapter, is a member of the user-defined `Problem`-derived object. Its value is assigned directly from the `alpha` argument variable in the `__init__` method. This value was stored as an object member because the argument variable of a method does not exist after a method has finished running. An object member, on the other hand, remains accessible as long as the object exists, and is accessible by the other object methods.

The addition, subtraction, multiplication, and division operators have the same precedence, or order of operations, as in algebra. In Python, exponentiation is represented by the `**` operator. It has a higher precedence than multiplication and division operators. The exponent is an integer, rather than a floating-point number (such as `2.0`) because exponentiation with an integer power is performed more quickly (through multiplication), rather than though calculation by logarithms.

Since the user-defined `Volume` class is derived from `pygdh.Volume` as defined by PyGDH, each `Volume` object has an automatically-defined member named `volume` which gives the size of the associated volume in the dimensional space represented by the `Volume` object. In one dimension, this `volume` is just the length associated with the `Volume` object (here, $b - a$), whereas in two dimensions, `volume` is the area associated with the `Volume` object.

Finally, in the `assign_equations` method, `ode` was associated with a single scalar value in the `equation_scalar_counts` dictionary. This was appropriate, because the solution is governed by a single scalar equation at any point in the domain. Therefore, the array associated with the second explicit parameter `residual` (which is really a portion of a much larger array) contains only one element. The first element of an array has index `0`, so the result of calculating $F(x, y(x))$ is stored in `residual[0]`. In order to keep the line length to under 80 characters, the entire expression is enclosed within parentheses, which allows the expression to be split over multiple lines.

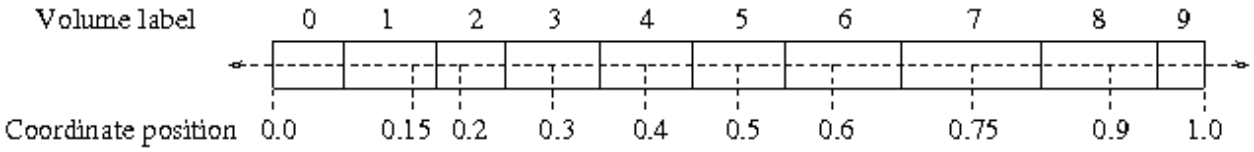
Interpolation in one dimension

Interpolation is frequently needed because for a given solution variable, only one value is used to characterize the solution behavior associated with any given `Volume` object. The `Interpolant` objects can calculate interpolants (and derivatives of a specified order) to a specified accuracy, at particular positions. These are typically defined in the `Volume.define_interpolant` methods. This allows formulas to be determined once, when they are automatically created as part of a `Grid` object, and then used repeatedly in the solution process. By computing and storing the interpolation formulas once, before any other equations are solved, this approach saves a very significant amount of computing time later.

Normalized relative position

When defining interpolation operators and describing governing equations, spatial position is indicated to PyGDH in terms of a coordinate system that is local to the `Volume` of interest. In one dimension, the normalized relative position ranges from `-1.0` to `1.0` over a one dimensional region associated with a `Volume` object. The "left" boundary, associated with a smaller coordinate value, corresponds to `-1.0`, and the "right" boundary corresponds to `1.0`.

For an `Volume` object on the interior of a `Grid`, the `coordinate` position of the associated `Volume` corresponds to `0.0`. Since this position is not necessarily located midway between the "left" and "right" boundaries (see the diagram of an irregular grid), the true distances associated with positive and negative normalized relative positions of equal magnitude may not be equal. However, the true positions vary linearly with the normalized relative positions in their respective directions away from `0.0`.



In one dimension, for a `Volume` adjacent to a domain boundary, the normalized relative position `0.0` corresponds to the midway point between the "left" and "right" boundaries, which is different from the `coordinate` position. The domain boundaries coincide with the `Volume` boundaries.

Defining interpolants

In order to facilitate position-independent representations, it is helpful to define consistently-named methods (across multiple `Volume` objects) that compute a certain type of result within the corresponding volume. For example, one might wish to compute an approximate derivative at the right boundary of each `Volume`. The appropriate formula will be different for each `Volume`, as each `Volume` has a different set of neighbors, but each `Volume` can store the appropriate formula as a method with the same name. In this way, the governing equation methods can be written very generally, as functions of the `Volume` object of interest, and the appropriate derivative formula can be accessed as a consistently-named method of that object. This technique was used in the first example program to define a single governing equation method, `ode`, for use throughout the problem domain and without the need to explicitly use discretization formulas.

These methods should be defined in the `define_interpolants` method within the `Volume` class definition:

```
    # Defining this method gives users an opportunity to define their own
    # mathematical operators, for later use in describing governing equations.
    def define_interpolants(self):

        # Operator for estimating first derivative at left boundary of volume
        self.dx_left = self.default_interpolant(-1.0, 1, 1)

        # Operator for estimating function value at the volume center
        self.interpolate = self.default_interpolant(0.0, 0, 2)

        # Operator for estimating first derivative at right boundary of volume
        self.dx_right = self.default_interpolant(1.0, 1, 1)
```

Here, three user-defined members of the `Volume` object are created. These new members store functions (functions are objects, too) that compute the first spatial derivatives at the right and left boundaries, and the interpolated function value at the center of the corresponding volume.

In this way, each `Volume` object has methods named `dx_right`, `dx_left`, and `interpolate`, which have consistent meanings with respect to the corresponding spatial regions. Other methods may use these routines without detailed knowledge of the underlying numerical details.

The `define_interpolant` method built into the `pygdh.Volume` class creates and returns an `Interpolant` object, which behaves according to the provided parameter values. While an object of this class contains a method named `value` which actually computes the value of interest, it is unwieldy to write expression such as `vol.dx_left.value(y)`. Instead, the `Interpolant` objects allow a special notation, in which the values of interest can be computed by appending the function call or indexing operators to the Interpolant object name, as in `vol.dx_left(y)` or `vol.dx_left[y]`. The former has been used the the `ode` method.

The `define_interpolant` method requires three parameter values. The first of these is the normalized relative position in the corresponding volume at which the value is to be computed. .. This requires some explanation, which is left to the next subsection.

In one dimension, the second parameter value gives the order of the derivative to be computed. The interpolated solution function value is `0`, the first spatial derivative is `1`, and so on.

The third parameter value specifies the accuracy of the interpolation formula. For those familiar with the derivation of difference formulas, this is the exponent of the highest-order term in the Taylor series expansion that is used in the coefficient calculations. This value should be at least as large as the order of the derivative to be computed. For the first example, it is safe to take this number to be `2` for the interpolated function value, and `1` for a first spatial derivative.

3.5.2 Suggested exercises

1. Solve the original governing equation, but for $1 < x < 2$, and with the boundary conditions

$$\begin{aligned} y(1) &= 0 \\ y(2) &= 1 \end{aligned}$$

Do you get what you expect?

2. Modify the original example code to solve

$$\begin{aligned} \frac{d^2y}{dx^2} &= -\alpha^2 y + \beta x \\ y(0) &= 0 \\ y(1) &= 2 \end{aligned}$$

for $\alpha = \beta = 1$ and $0 \leq x \leq 1$. Hint: As a rough approximation, use

$$\int_a^b \beta x \, dx \approx \beta x_{\text{center}}(b-a),$$

where, in this one-dimensional spatial problem, x_{center} can be obtained as the `coordinate` member of a `Volume` object is a scalar giving the position of the grid point associated with the `Volume`.

The results should agree well with the analytical solution

$$y(x) = \frac{\sin(\alpha x)}{\sin(\alpha)} + \frac{\beta x}{\alpha^2}$$

3. Modify the original example code to solve

$$\begin{aligned} x^2\frac{d^2y}{dx^2} + 2x\frac{dy}{dx} - 2y &= 0 \\ y(1) &= 1 \\ y(2) &= 1 \end{aligned}$$

for $1 \leq x \leq 2$. Note the change in the spatial domain.

Hint: Use rough approximations such as

$$\int_a^b x\frac{dy}{dx} \, dx \approx x_{\text{center}} \int_a^b \frac{dy}{dx} \, dx.$$

The results should agree well with the analytical solution

$$y(x) = \frac{3x}{7} + \frac{4}{7x^2}.$$

3.6 Solving a PDE with a flux boundary condition

A time-dependent PDE may be solved by combining the techniques used previously to describe ODEs in time and space.

The example program is included as `pygdh/examples/PDE/PDE.py` within the PyGDH archive. It begins with

```
import pygdh
import numpy
import math
```

where the `math` module contains common mathematical functions.

This example will solve the one-dimensional heat equation,

$$\frac{\partial y}{\partial t} = \alpha \frac{\partial^2 y}{\partial x^2}$$

with the boundary conditions

$$y(0) = 0$$
$$\alpha \frac{\partial y}{\partial x}(1) = 1.$$

3.6.1 Discretization

This section will discuss the discretization of a PDE with a solution that varies in both space and time, using the one-dimensional heat equation as an example.

Using the backward difference formula introduced in the previous chapter, the differential equation

$$\frac{\partial y}{\partial t}(t, x) = g(t, x, y(t, x))$$

is approximated by

$$\frac{y(t_0 + \Delta t, x) - y(t_0, x)}{\Delta t} \approx g(t_0 + \Delta t, x, y(t_0 + \Delta t, x)).$$

As in the first example problem, we now obtain the finite-volume form by moving all nonzero terms to one side and integrating both sides over a one-dimensional volume ranging from a to b gives

$$\int_a^b \frac{\partial y}{\partial t}\, dx - \alpha \left[\frac{\partial y}{\partial x}\right]_a^b = 0.$$

Using the average value of the integrand, this is

$$\left\langle \frac{\partial y}{\partial t} \right\rangle (b - a) - \alpha \left[\frac{\partial y}{\partial x}\right]_a^b = 0.$$

As an approximation, the average value of the time derivative is now taken as the value of the time derivative at the center of the volume:

$$\left(\frac{\partial y}{\partial t}\right)_{\text{center}} (b - a) - \alpha \left[\frac{\partial y}{\partial x}\right]_a^b \approx 0,$$

which is the same thing as the time derivative of the value at the center of the volume:

$$\frac{\partial y_{\text{center}}}{\partial t}(b-a) - \alpha \left[\frac{\partial y}{\partial x}\right]_a^b \approx 0.$$

Discretization in time by the first-order backward difference scheme then gives

$$\frac{y\left(t_0+\Delta t, \frac{b+a}{2}\right) - y\left(t_0, \frac{b+a}{2}\right)}{\Delta t}(b-a) - \alpha\left(\frac{\partial y}{\partial x}(t_0+\Delta t, b) - \frac{\partial y}{\partial x}(t_0+\Delta t, a)\right) \approx 0.$$

As in the previous chapter, the `stored_solution_count` parameter to the `initialize_problem` method will be used to notify PyGDH that storage space must be provided for both the present solution and a past solution:

```
    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_problem' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, alpha, volume_count):

        # Save the argument value as an object member
        self.alpha = alpha

        # Create the Grid object representing the problem domain
        grid_obj = Domain('domain', alpha, volume_count)

        # Prepare to run simulation on new Grid object
        self.initialize_problem([grid_obj],stored_solution_count=2)
```

Due to the boundary condition in this example, there are two methods representing governing equations; this section will focus on the first, named `pde`, and discussion of `pde_boundary` will be left to a later section.

As in the previous examples, the Python method computes a single scalar value and can be written as a simple transcription of the discretized equation. It makes use of the same operators and object data members, along with the timestep size, which is available as the automatically-defined `Problem`-derived object member `self.timestep_size`:

```
    def pde(self, vol, residual):

        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]

        # FVM representation of governing equation
        residual[0] = ((vol.interpolate(y0)
                        - vol.interpolate(y_1))*vol.volume
                       - self.timestep_size * self.alpha
                       * (vol.dx_right(y0)
                          - vol.dx_left(y0)))
```

The local variables `y0` and `y_1` represent the current and past solutions. As in the previous chapter, they are used to reference a hierarchy of data structures. The definition of the local variable `y0` is equivalent to

```
y0 = ( ( ( ( self.grid ) [0] ) . field ) [0] ) [0]
```

Reading this from the innermost parentheses to the outermost parentheses, this definition is seen to be almost identical to that in the previous chapter, through step 4. That is, `self.grid[0].field[0]` is a two-dimensional array

of solution values at the present (0th) timestep on the single (0th) `Grid` describing the spatial problem domain. As before, the first array index indicates the independent variable (corresponding to the list of independent variable names in the `field_names` keyword parameter passed to `initialize_grid`), and the second index indicates the `Unit` associated with the position at which the selected variable has the stored value.

However, unlike in the previous chapter, only one index is given for this two-dimensional solution value array. When only one index is provided to a two-dimensional array, it specifies a value for the first index, and a one-dimensional section of this array (consisting of all elements with the provided index as their first index) is returned; in effect, a two-dimensional array may be accessed as a one-dimensional array of one-dimensional arrays. The index `0` given here corresponds to the only independent variable, `y`.

The resulting one-dimensional array can be accessed by a single index that is equivalent to the second integer index into the original two-dimensional array, so that an index into this array identifies a unique `Unit` object, although that is not explicitly done here, because the interpolation methods take one-dimensional arrays of this type as input. These methods are already aware of the spatial arrangement of the solution values in these arrays.

The integer label identifying a `Unit` object is automatically determined by PyGDH and may be accessed as the `number` member of a `Unit` object. In the other direction, the automatically-generated `unit_with_number` member of a `Grid` object is an array that stores the unit object associated with a given numerical label. In the simple case of a one-dimensional domain, the unit objects are simply numbered according to their spatial position. One may take `0` to correspond to the "left" end of the domain, with the smallest coordinate value, and the index `-1` as corresponding to the "right" end of the domain, with the largest coordinate value.

3.6.2 Boundary conditions

One of the boundary conditions (at the left boundary) is identical to that used in the first example problem, so the relevant entries in `declare_unknowns`, `set_boundary_values`, and `assign_equations` need no further explanation.

The condition at the right boundary is a flux condition, which applies to the derivative of the independent variable, rather than its value. The user-defined `set_boundary_values` method is only used at the boundaries at which the value is prescribed; other boundary conditions must be incorporated into governing equation methods.

For this boundary condition,

$$\alpha \frac{\partial y}{\partial x}(1) = 1$$

a second method representing the same governing equation is introduced for use with the corresponding boundary volume only. The boundary condition can be substituted directly into the discretized equation obtained earlier:

$$\frac{y\left(t_0 + \Delta t, \frac{b+a}{2}\right) - y\left(t_0, \frac{b+a}{2}\right)}{\Delta t}(b-a) - \alpha\left(\frac{1}{\alpha} - \frac{\partial y}{\partial x}(t_0 + \Delta t, a)\right) \approx 0$$

$$\frac{y\left(t_0 + \Delta t, \frac{b+a}{2}\right) - y\left(t_0, \frac{b+a}{2}\right)}{\Delta t}(b-a) - 1 + \alpha\frac{\partial y}{\partial x}(t_0 + \Delta t, a) \approx 0$$

The Python representation is then given by:

```
    def pde_boundary(self, vol, residual):

        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]
```

(continues on next page)

(continued from previous page)

```
        # FVM representation of governing equation, incorporating boundary
        # condition
        residual[0] = ((vol.interpolate(y0)
                        - vol.interpolate(y_1))*vol.volume
                       + self.timestep_size
                       * (-1.0 + self.alpha * vol.dx_left(y0)))
```

As the boundary condition at the "right" side of the domain gives the value of the derivative of the solution variable, rather than prescribing its value, the value of the solution variable at this boundary is unknown and must be calculated by PyGDH. For this reason, the corresponding `unknown` value is set to `True` in the `declare_unknowns` method.

Finally, this new method must be noted in `equation_scalar_counts`, and the boundary equation must be associated with the volume at the right boundary. As with the solution arrays, the elements of the `equations` structure are sequences with elements corresponding to `Volume` objects, numbered as in the `unit_with_number` members of the `Grid` objects.

```
    # This is a required method that notifies PyGDH about the number of scalar
    # values returned by each governing equation method.
    def set_equation_scalar_counts(self):

        # This is a dictionary associating governing equation methods with
        # the number of scalar values that they return
        self.equation_scalar_counts = {self.pde: 1,
                                       self.pde_boundary: 1}
```

Initial condition

Since the example problem is time-dependent, the initial condition must be provided. This must be done in a user-defined method named `set_initial_conditions` within the definition of the `Problem`-derived class. This method must have `self` as its only parameter.

The initial condition should be a continuous function of position and must be consistent with the boundary conditions; the numerical initial condition defined in `set_initial_conditions` must be consistent with the boundary conditions as expressed in `set_boundary_conditions`.

The `set_initial_conditions` methods should set the values for the present solution, with index `0` in the `field` deque. It should set the values of the dependent variables at all locations at which they are unknown; while there is no need to set values that are also set by `set_boundary_conditions` it causes no harm.

In the present example,

```
    # This sets the initial values of the independent variables on the region
    # represented by an object of this class.
    def set_initial_conditions(self):

        # Defined for clarity
        y = self.grid[0].field[0][0]

        # Set the value of the independent variable on each volume
        for vol in self.grid[0].unit_with_number:
            y[vol.number] = (
                vol.coordinate / self.grid[0].alpha
                - 0.5*math.sin(0.5*math.pi*vol.coordinate))
```

a local variable `y` is defined for clarity. The for loop in this method iterates over the `Volume` objects in `unit_with_number`, setting the initial dependent variable value for each. The `number` members of the `Volume` objects are used to find the appropriate positions in the solution value arrays.

A known solution should be provided as the initial condition. An initial condition based on a single spatial eigenfunction gives the analytical solution the particularly simple form:

$$y(t,x) = \frac{x}{\alpha^2} + \frac{1}{2}\exp\left[-\left(\frac{\pi\alpha}{2}\right)^2 t\right]\sin\left(\frac{\pi}{2}x\right)$$

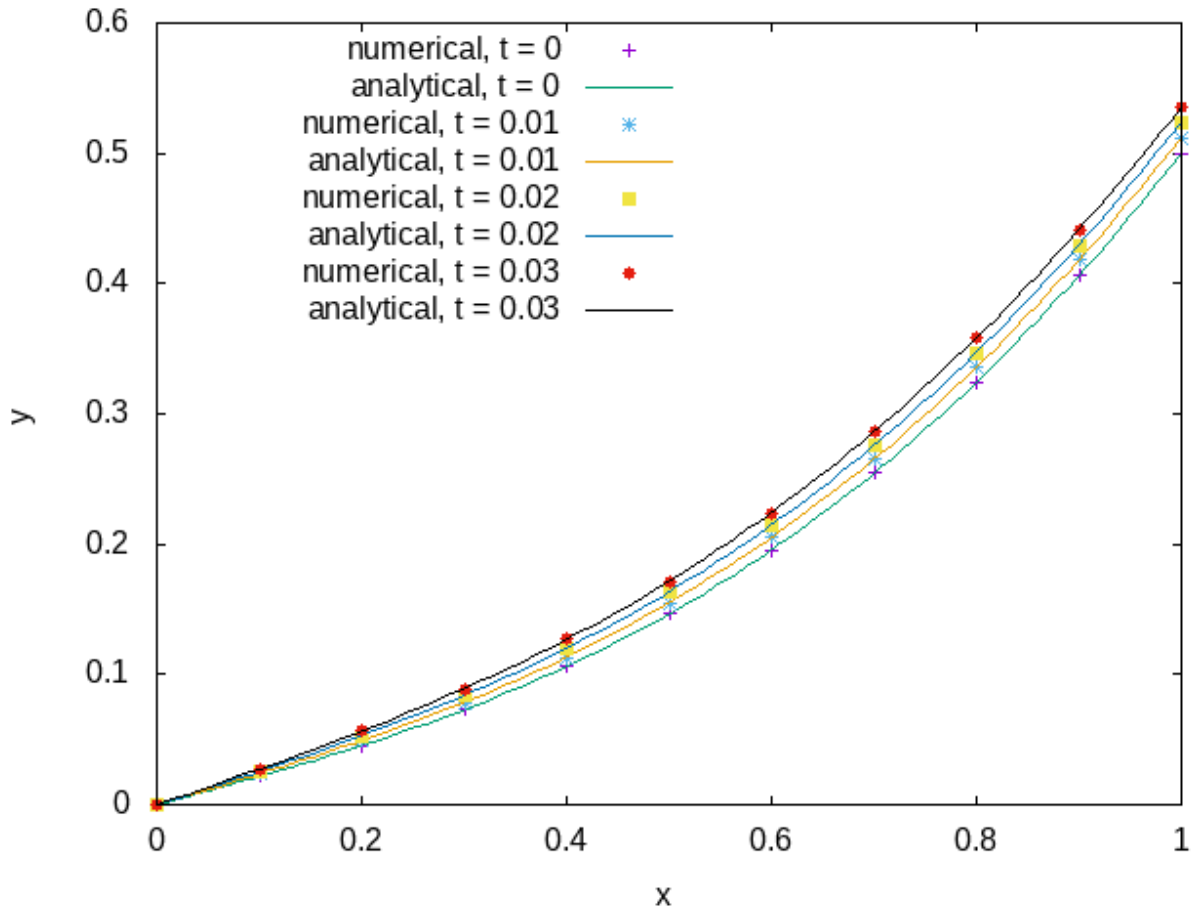
This is evaluated for $t = 0$, and the result is translated into the Python representation used above. This example makes use of the `Domain` object member `alpha`, as well as entities from the `math` module, and the `coordinate` member of the `Volume` objects, which indicate their coordinate position in the spatial domain.

3.6.3 Solving a time-dependent problem

As in the previous chapter, the time-dependent solver method is used:

```
# Create an object instance with 11 volumes and setting alpha = 1.0
problem = PDE(1.0, 11)

number_of_timesteps = 3
timestep_size = 0.01

# The names of all output files will begin with this string
filename_root = 'PDE'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

# Calculate solution and write output to file
problem.solve_time_dependent_system(filename_root, output_types,
                                    timestep_count=number_of_timesteps,
                                    timestep_size=timestep_size)
```

The results are plotted against the analytical solution at a few timesteps:



This output was produced with the following GNUPLOT script:

```
set term png
set output 'PDE.png'
set key left top
set xlabel 'x'
set ylabel 'y'
plot alpha=1.0, y(t,x)=x/alpha**2-0.5*exp(-(pi*alpha*0.5)**2*t)*sin(0.5*pi*x), 'PDE_
↪domain.gnuplot' using 1:2 index 0 title 'numerical, t = 0' with points, y(0,x)␣
↪title 'analytical, t = 0', 'PDE_domain.gnuplot' using 1:2 index 1 title 'numerical,␣
↪t = 0.01' with points, y(0.01,x) title 'analytical, t = 0.01', 'PDE_domain.gnuplot'␣
↪using 1:2 index 2 title 'numerical, t = 0.02' with points, y(0.02,x) title
↪'analytical, t = 0.02', 'PDE_domain.gnuplot' using 1:2 index 3 title 'numerical, t␣
↪= 0.03' with points, y(0.03,x) title 'analytical, t = 0.03'
set output
```

3.6.4 Suggested exercises

1. Change the number of timesteps and timestep size. Do the numerical results remain consistent with the analytical solution?
2. Suppose that `self` references a `Problem` object. What does `self.grid[1]` mean? How about `self.grid[0].field[-1]`, `self.grid[1].field[0][1]`, `self.grid[1].field[0][1][20]`, and `self.grid[1].field[0][1,20]`? Why is it wrong to write `self.grid[2].field[0,1,20]`?

3. Change the flux boundary condition to

$$\alpha \frac{\partial y}{\partial x}(1) = 10.$$

Use the initial condition

$$y(x) = \frac{10x}{\alpha} + \sin\left(\frac{\pi}{2}x\right).$$

Your results should match the analytical solution

$$y(t, x) = \frac{10x}{\alpha} + \exp\left[-\left(\frac{\pi\alpha}{2}\right)^2 t\right] \sin\left(\frac{\pi}{2}x\right).$$

3.7 Specifying numerical output

This tutorial has used only GNUPLOT-formatted output, specified by including the `'GNUPLOT'` string in the `output_types` argument to `solve_time_independent_system` and `solve_time_dependent_system`. However, other output formats are available, and output files can produced in multiple formats simultaneously–this is why the `output_types` argument is a list. Currently, the other supported output types are 'CSV' (comma-separated values), 'PICKLE' (a Python-specific format) and 'HDF5' (Hierarchical Data Format, version 5).

The human-readable text file formats (currently 'GNUPLOT' and 'CSV') are suitable for relatively simple problems, and CSV is particularly suitable for use with spreadsheet applications.

The 'PICKLE' and 'HDF5' formats are not human-readable; they store solutions for later access by PyGDH. In particular, this is useful for saving solutions from which simulations can be restarted, and for doing sophisticated postprocessing of numerical solutions. These will be discussed in the upcoming chapter on postprocessing.

3.7.1 Output file names

If a single `Grid` object is used in a program, the output file names are have the root specified by the `filename_root` argument to `solve_time_independent_system` and `solve_time_dependent_system`, followed by the descriptive `Grid` label, and an extension determined by the file format (`.gnuplot` for `'GNUPLOT'`, `.csv` for `'CSV'`, `.pkl` for `PICKLE`, and `.h5` for `'HDF5'`).

3.7.2 GNUPLOT format

In the `'GNUPLOT'` format, the results for a single timestep occupy contiguous non-blank lines. Results for separate timesteps are separated by two blank lines, so that the `index` keyword in a GNUPLOT `plot` or `splot` statement may be used to select results from specific timesteps. The first timestep in the file, associated with the GNUPLOT index `0`, reflects the initial condition.

Each non-blank line is either a comment, beginning with `#`, or consists of a sequence of numbers, separated by spaces. For a `Grid` with more than one `Volume`, the lines of data begin with the `coordinate` value(s) of a `Volume` in the `Grid`, and are followed by the solution field values in the order given by the `field_names` parameter given at the `Grid` object creation. These are followed by additional output field values specified by the user. For a two-dimensional `Grid`, single blank lines are used to separate data along separate grid lines so that GNUPLOT recognizes the grid layout as such.

For a `Grid` with a single `Volume`, the lines of data consist of the timestep number followed by the simulation time, and then the solution values in the same order as in the previous case. No blank lines are left between results from successive timesteps.

Comment lines are included, and indicate the meanings of the various columns, along with the timestep index and simulation time for time-dependent problems.

3.7.3 CSV format

There is no standard "comma-separated values" format, but PyGDH makes use of the `csv` module in the standard Python library. The output files appear to work well with common spreadsheet applications. This format is very similar to the GNUPLOT format, except that numbers are separated by commas, blank lines are not placed between successive grid lines in the two-dimensional case, and only a single separator line is left between results for different timesteps (for `Grid` objects with more than one `Volume`). The separator line contains the timestep number and simulation time in the first two columns; other values occupy the third column onward.

3.7.4 HDF format

The Hierarchical Data Format allows multiple datasets to be stored in a single file much as files are stored on a hard disk. HDF5 output requires installation of the HDF5 library, and the h5py Python interface to the HDF5 library.

HDF5 formatted files are named by appending the extension 'h5' to the `filename_root` parameter to `solve_time_independent_system` and `solve_time_dependent_system`.

The HDF5 library and format offer many advantages over the simple text file output formats. However, the use of these files will be transparent to most users of PyGDH, as long as PyGDH is used to store and retrieve results from these files. Also, since exact numerical results are stored in this format, as opposed to the text file formats in which human-readable representations are stored, PyGDH offers the option of continuing a time-dependent simulation from results associated with any timestep in an HDF5 file.

The HDF5 files are not in human-readable form. There are a number of utilities that one may use to extract data from an HDF5 file, but as PyGDH is capable of handling storage and retrieval of its results, most users will not need to use additional tools.

3.7.5 PICKLE format

The PICKLE format is a Python-specific format, and like the HDF5 output option, stores data in a way that is convenient for further processing. However, it is much slower at certain operations, and is platform-specific. This format is meant to provide the ability for sophisticated postprocessing when a user is unable to use the HDF5 library. From the perspective of a PyGDH user, PICKLE files are used by PyGDH exactly as HDF5 files are used, but HDF5 files are access preferentially if both are available.

HDF5 formatted files are named by appending the extension 'pkl' to the `filename_root` parameter to `solve_time_independent_system` and `solve_time_dependent_system`.

3.7.6 Adding additional output fields: an example

The solution field values are always saved in the output files, but it may be desirable to have additional output variables. For example, a solid mechanics simulation may use displacement as the solution variable, but stress values, which may be computed from displacement, may also be of interest. Here, we discuss an example program that produces additional output. The program is included in the PyGDH archive as `pygdh/examples/output/output.py`.

Consider a one-dimensional system, a spherically-symmetric spherical body experiencing linearly-elastic deformation. The solution variable will be the displacement u.

Since this system is spherically symmetric, the momentum equation has only one non-zero component, associated with the radial direction. Under the assumption of negligible inertia, this gives an equation for mechanical equilibrium:

$$\frac{\partial t_r}{\partial r} + \frac{2}{r}(t_r - t_t) = 0$$

where t_r is the normal stress on a radially-oriented surface, and t_t is the normal stress on any surface perpendicular to the radial direction. All other stress components are zero. The boundary conditions are

$$u(0) = 0$$
$$t_r(1) = 0$$

In order to obtain a discretized form by the finite-volume method, one may integrate over the volume of the sphere contained between radial distances r_0 and r_1. However, since the governing equation is independent of angle, this is equivalent to integrating over radial position only, with an additional factor of r^2. But one may multiply the rearranged equation by any factor, so except at $r = 0$, one may simply integrate both sides of the previous equation in r to obtain

$$t_r(r_1) - t_r(r_0) + 2\left\langle \frac{t_r - t_t}{r} \right\rangle (r_1 - r_0) = 0$$

The angle brackets indicate an average over the interval–for simplicity, this will be approximated by the value of the enclosed quantity at the midpoint between r_0 and r_1. No equation will be needed at the origin, because the solution value at the origin will be given.

The stress components are given by

$$t_r = \lambda(tr\mathbf{E}) + 2\mu\frac{\partial u}{\partial r}$$
$$t_t = \lambda(tr\mathbf{E}) + 2\mu\frac{u}{r}$$
$$tr\mathbf{E} = \frac{\partial u}{\partial r} + 2\frac{u}{r}$$
$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$$
$$\mu = \frac{E}{2(1+\nu)}$$

Although u will be used as the solution variable, it is clearer to work in terms of the stress components, and it is desirable to include the stress components in the output as well.

PyGDH requires that additional output variables be arrays with one element for every `Volume` of the associated `Grid` object. This ensures that any additional output data structures have the same structure as the solution field arrays and will fit neatly into the human-readable text formats.

Declaring global variables

While variables used to hold intermediate values do not necessarily have to be output variables, they should persist throughout the solution process, so that their values can be used and modified by multiple methods. Variables of this sort should be defined in the `define_field_variables` method of a `Grid`-derived object. These methods are called once per `Grid` object for an entire simulation, before any numerical solutions are obtained. By defining these methods in `Grid`-derived classes, these variables are naturally associated with a specific `Grid`.

From the discretized equation, one can see that the interpolated solution field values will be needed at the boundaries of each `Volume`, and at the centers; however, for consistency with the solution field output, the desired stress component output values correspond to the `Volume coordinate` locations. For `Volume` objects on the edges of the domain, the `coordinate` locations are associated with the outer boundaries, but for `Volume` objects on the interior, the `coordinate` locations are associated with the `Volume` centers. None of the data structures of intermediate calculations are appropriate, so additional data structures will be created specifically for output. For this example, the `define_field_variables` method is written as:

```
    # Variables that are used by multiple methods and/or additional output
    # variables should be defined here. This routine is called once before
    # any numerical solutions are generated.
    def define_field_variables(self):

        # 'numpy.empty' creates an NumPy array of the specified size, with
        # uninitialized element values. The specified sizes here are consistent
        # with the solution arrays.
        self.tr_r0 = numpy.empty(self.unit_count)
        self.tr_center = numpy.empty(self.unit_count)
        self.tr_r1 = numpy.empty(self.unit_count)

        self.tt_r0 = numpy.empty(self.unit_count)
        self.tt_center = numpy.empty(self.unit_count)
        self.tt_r1 = numpy.empty(self.unit_count)

        self.tr_output = numpy.empty(self.unit_count)
        self.tt_output = numpy.empty(self.unit_count)
```

The first six variables defined above will be used to hold the results of intermediate calculations. The last two variables are used to temporarily store additional output values.

Calculation of global variable values

Intermediate calculations can be performed in a `calculate_global_values` method (defined in a `Problem`-derived class) which PyGDH calls before evaluating the discretized equation methods over the problem domain. This is the appropriate place to make calculations and store intermediate results, especially when they are used in the discretized equation methods of more than one `Volume`. Here, the intermediate values (stress components, in this case) are calculated in a `calculate_global_values` method and stored in the newly-defined data structures:

```
    # This optional method provides an opportunity to calculate and store
    # values in the global variables, before the equation methods are
    # evaluated (on each timestep, for a time-dependent problem).
    def calculate_global_values(self):

        # Defined for efficiency and clarity
        u = self.grid[0].field[0][0]

        for vol in self.grid[0].unit_with_number:

            ## Calculate stress components at left edge of volume

            if vol.number == 0:
                # Since the value of u is given at the origin, the stress
                # components at the origin are not needed in the calculations.
                # However, it may be desirable to include them in the output.
                #
                # Limiting forms of the stress components are needed at the
                # origin because they depend on u / r.  Unfortunately,
                # obtaining this ahead of time usually requires some
                # analytical result.
                u__r = vol.dx_left(u)
            else:
                u__r = (vol.interpolate_left(u)
```

(continues on next page)

(continued from previous page)

```
                    / (vol.coordinate + vol.left))

            trE = vol.dx_left(u) + 2*u__r

            self.grid[0].tr_r0[vol.number] = (self.Lame_lambda*trE
                                              + 2*self.mu*vol.dx_left(u))
            self.grid[0].tt_r0[vol.number] = (self.Lame_lambda*trE
                                              + 2*self.mu*u__r)

            ## Calculate stress components at center of volume

            self.grid[0].tr_center[vol.number] = (
                self.Lame_lambda*trE
                + 2*self.mu*vol.dx_center(u))
            if vol.number == 0:
                self.grid[0].tt_center[vol.number] = (
                    self.grid[0].tr_center[vol.number])
            else:
                self.grid[0].tt_center[vol.number] = (
                    self.Lame_lambda*trE
                    + 2*self.mu*vol.interpolate(u)
                    / (vol.coordinate + vol.center))

            ## Calculate stress components at right edge of volume

            trE = (vol.dx_right(u)
                   + 2*vol.interpolate_right(u)
                   / (vol.coordinate + vol.right))

            self.grid[0].tr_r1[vol.number] = (
                self.Lame_lambda*trE
                + 2*self.mu*vol.dx_right(u))
            self.grid[0].tt_r1[vol.number] = (
                self.Lame_lambda*trE
                + 2*self.mu*vol.interpolate_right(u)
                / (vol.coordinate + vol.right))
```

These methods are called before the governing equation methods on every iteration of the nonlinear solver.

Alternatively, global values may store the results of limited postprocessing performed on numerical solutions. This may be done by defining a method named `process_solution`, within a `Problem`-derived class. A method of this name is automatically called once after a solution is obtained (for a given timestep, in a time-dependent problem). In this example, although the stress component values needed for output are a subset of those calculated as intermediate values, the `calculate_global_values` does not copy them to the output data structures because this would be done on every iteration of the iterative solver used by PyGDH, even though only the final iteration for a given timestep produces the desired values. Instead, this copying is done in a `process_solution` routine, called once after a solution is obtained:

```
    # This routine is called once after a solution is obtained
    # (on each timestep in a time-dependent problem)
    def process_solution(self):

        grid0 = self.grid[0]

        grid0.tr_output[0] = grid0.tr_r0[0]
```

(continues on next page)

(continued from previous page)

```
        grid0.tt_output[0] = grid0.tt_r0[0]

        for vol_number in range(1,grid0.unit_count-1):
            grid0.tr_output[vol_number] = grid0.tr_center[vol_number]
            grid0.tt_output[vol_number] = grid0.tt_center[vol_number]

        grid0.tr_output[-1] = grid0.tr_r1[-1]
        grid0.tt_output[-1] = grid0.tt_r1[-1]
```

Declaring additional output variables

Additional output fields are specified in the `output_fields` member of a `Grid` object. This member is a dictionary, and is empty by default (however, the solution field values are automatically written to output). The `output_fields` dictionaries may be modified (or overwritten) in a user-defined method named `set_output_fields`, defined in each `Grid`-derived class. This method must take only `self` as an argument.

Entries in an `output_fields` dictionary must consist of descriptive strings as keys and NumPy arrays as values, and as mentioned in the previous paragraph, the arrays must have one element for every `Volume` in the `Grid`, to ensure consistency with the solution output written into the same files. PyGDH includes all of the specified arrays when writing to the output files after every timestep of a time-dependent process, or after the solution is found for a time-independent problem.

In this example, the last two variables defined in `define_field_variables` are marked as output variables. The descriptive keys in the `output_fields` dictionary are used to describe the data in output files.

```
    # This is an optional method for specifying additional output variables
    def set_output_fields(self):

        # This is a dictionary, with pairs of descriptive strings and arrays
        # with the same shape as the field variable arrays
        self.output_fields = {'radial_stress':self.tr_output,
                              'tangential_tress':self.tt_output}
```

The `scipy.optimize.fsolve` solver used by PyGDH is an iterative solver, which means that the discretized equation methods over the domain are evaluated many times in order to refine the numerical solution. The `calculate_global_values` method is called once at the start of each iterative step, and the `process_solution` method is called after the final iterative step has been made and an acceptable numerical solution has been obtained. The solver then evaluates the equations one last time to ensure that the error is within the desired tolerance. For this reason, the `calculate_global_values` method is called one final time with the numerical solution to be returned, and so calculations that it performs are consistent with the returned solution, and suitable for use as output.

Final comments

The rest of the program is similar to programs from previous chapters. As in the earlier `PDE` example, one of the boundary conditions must be introduce in `set_boundary_values`, while the other must be introduced through a separate governing equation method for the `Volume` on the corresponding boundary.

The additional output fields are associated with the solution output for the same `Grid` object. For HDF5 files, this is simply an implementation detail, but for the CSV and GNUPLOT formats, one output file is created for each `Grid`, and each output field occupies a different column at a given timestep (each row corresponds to a position in the domain, with the `Volume` position indicated by the first column, or columns in two dimensions). Since the additional output fields are specified in the `output_fields` dictionary, and the order in which entries are placed into dictionaries does

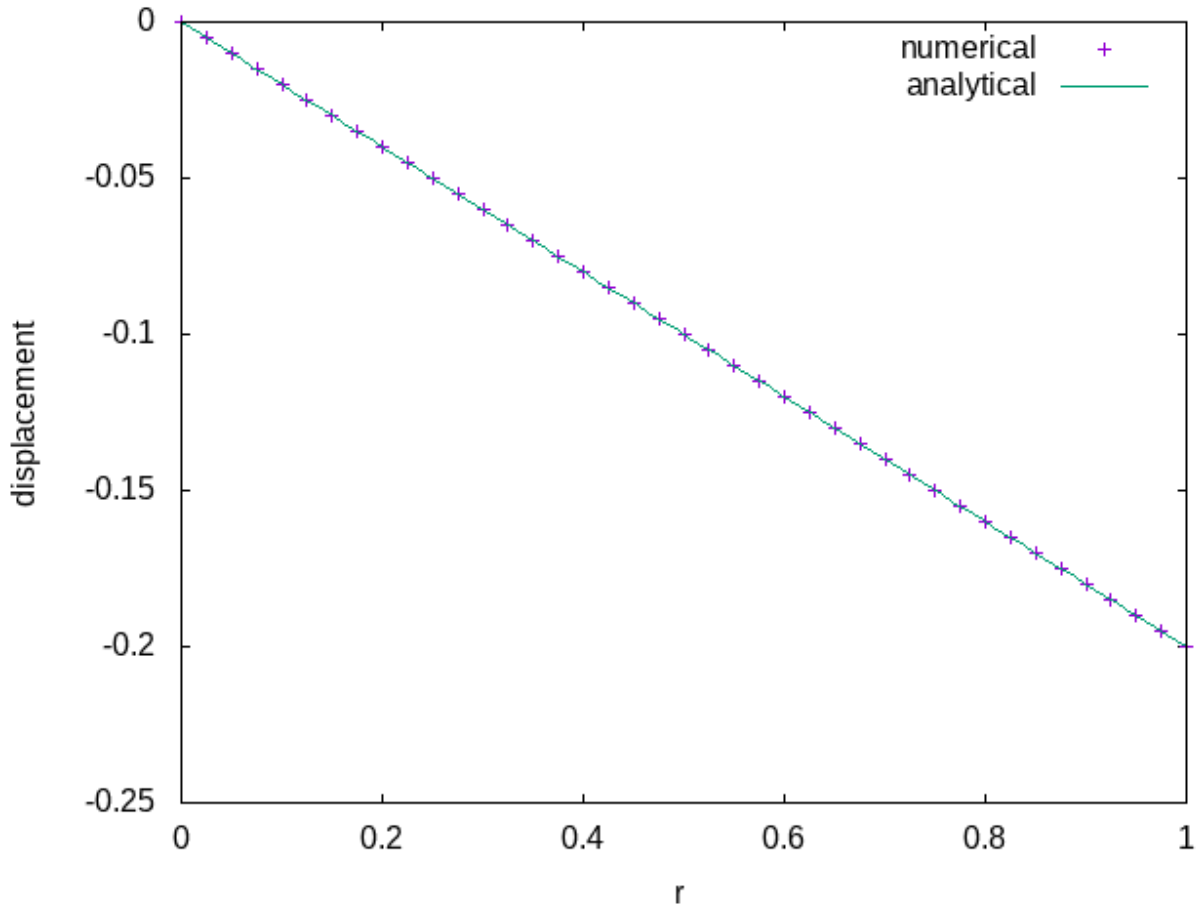
not necessarily indicate the order in which they are stored internally, header lines or comment lines are automatically added to the CSV and GNUPLOT text files so that the meaning of each column is clearly defined, even if the order in which they are stored by PyGDH is not evident. The ordering of columns will depend on the Python implementation used. One should always check the first lines of the output files for this information.

As it turns out, the analytical solution is particularly simple–u is linear in r:

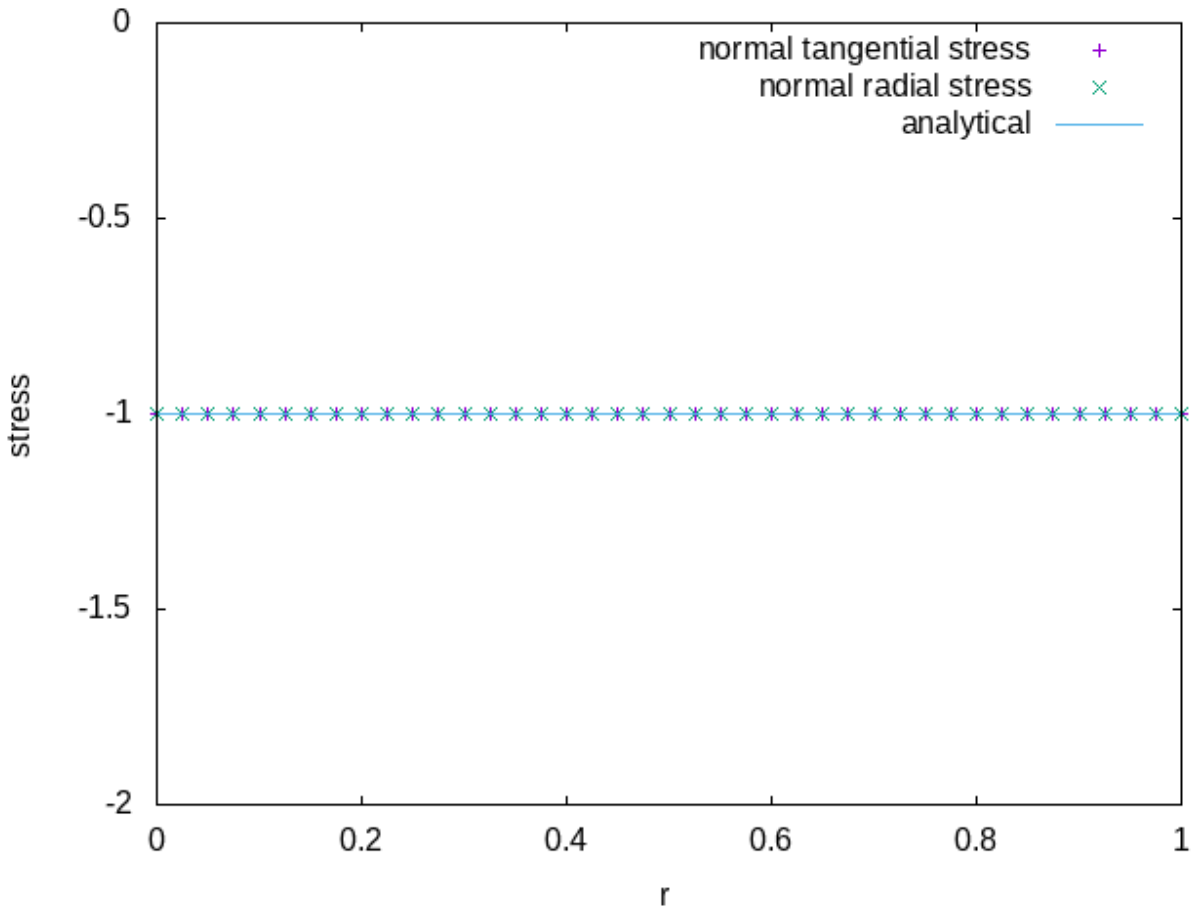
$$u(r) = -\frac{p}{3\lambda + 2\mu} r$$

where p is the pressure at the surface.

This is in agreement with the numerical result, taking $p = \lambda = \mu = 1$ for convenience:



As a consequence, the stress components are constants (and identical to each other, as they must be at the origin, due to symmetry), equal in magnitude to the pressure at the surface:



The GNUPLOT script used to generate these plots is:

```
set term png
set output 'output_u.png'
set xlabel 'r'
set ylabel 'displacement'
plot 'output_sphere.gnuplot' using 1:2 title 'numerical', -0.2*x title 'analytical'
set output 'output_stress.png'
set ylabel 'stress'
plot [][-2:0] 'output_sphere.gnuplot' using 1:3 title 'normal tangential stress',
↪'output_sphere.gnuplot' using 1:4 title 'normal radial stress', -1 title 'analytical
↪'
```

3.7.7 Suggested exercises

1. Modify the example program to produce a CSV output file as well, and import into a spreadsheet program and produce a plot. Are the results consistent with the GNUPLOT output?
2. Modify the example program to produce HDF5 output file as well, and use a HDF5 browser (for example, hdfview) to look through its contents.

3.8 A nonlinear problem and validation testing

PyGDH was originally created for the purpose of solving nonlinear systems, and uses a nonlinear solver from NumPy by default. The approach used to solve a nonlinear problem with PyGDH is the same as that used with the linear problems from the previous examples; the discretized equation methods may be linear or nonlinear in the solution variable values. The first half of this chapter will use PyGDH to solve a nonlinear problem.

The second half of this chapter will discuss numerical validation. Validation of solutions should be considered a necessary step in solving problems because errors can arise throughout the solution process–the original problem formulation, the discretized equations, and the computer implementation (including the underlying libraries like PyGDH) are all possible sources of error.

3.8.1 A nonlinear problem

This example extends the previous example to large (and nonlinear) displacements within a spherically-symmetric domain (ignoring any inconsistency in using Hooke's law in conjunction with large-deformation mechanics). The code is included in the PyGDH archive as `pygdh/examples/nonlinear.py`. Since displacements may be large, one must be careful to specify whether a function of position (actually, a function associated with a particular material "particle") is written in terms of the present, displaced position of the material particle, or the initial, undisturbed position of the particle. The first of these is the "spatial", or "Eulerian", description, and the second is the "material", or "Lagrangian", description.

PyGDH does not directly support calculations on domains that vary in shape or size over the course of a calculation; the coordinate values of all grid lines cannot be changed. However, it is still possible to solve problems on domains that are not fixed, simply by formulating the problems appropriately–all that is necessary is to map the changing domain to a fixed domain. In this case, one way to achieve this is to transform equations written in the spatial description to equations written in the material description. The resulting equations are then solved as functions of position on the original domain.

It turns out that performing this transformation for the stress balance gives an equation with the same form:

$$\frac{\partial t_r}{\partial r} + \frac{2}{r}(t_r - t_t) = 0$$

where the symbol r now refers to the initial radial position of the material particle in question, and the stress components take a more complicated form:

$$t_r = \left(1 + \frac{u}{r}\right)^2 \left\{ \lambda(\mathrm{tr}\mathbf{E^a}) + \mu \left(1 - \frac{1}{\left(1 + \frac{\partial u}{\partial r}\right)^2} \right) \right\}$$

$$t_t = \left(1 + \frac{u}{r}\right)^2 \lambda(\mathrm{tr}\mathbf{E^a}) + \left(1 + \frac{\partial u}{\partial r}\right) \mu \left(1 + \frac{u}{r} - \frac{1}{1 + \frac{u}{r}} \right)$$

$$\mathrm{tr}\mathbf{E^a} = \frac{3}{2} - \frac{1}{2\left(1 + \frac{\partial u}{\partial r}\right)^2} - \frac{1}{\left(1 + \frac{u}{r}\right)^2}$$

Under the assumptions that $|u/r| << 1$ and $|\frac{\partial u}{\partial r}| << 1$, one may show that the small-deformation equations may be recovered.

The discretized equation retains the same overall form as in the previous example:

$$t_r(r_1) - t_r(r_0) + 2\left\langle \frac{t_r - t_t}{r} \right\rangle (r_1 - r_0) = 0$$

where again, the angle brackets indicate an average over the interval–for simplicity, this will be approximated by the value of the enclosed quantity at the midpoint between r_0 and r_1. No equation will be needed at the origin, because the solution value at the origin will be given as before.

The implementation will be similar to the small-deformation implementation, with two significant additions. First, the `calculate_global_values` method is modified to include the large deformation calculations described above:

```
    # This optional method provides an opportunity to calculate and store
    # values in the global variables, before the equation methods are
    # evaluated (on each timestep, for a time-dependent
    def calculate_global_values(self):

        # Defined for efficiency and clarity
        u = self.grid[0].field[0][0]

        for vol in self.grid[0].unit_with_number:

            ## Calculate stress components at left edge of volume

            if vol.number == 0:
                # Since the value of u is given at the origin, the stress
                # components at the origin are not needed in the calculations.
                # However, it may be desirable to include them in the output.
                #
                # Limiting forms of the stress components are needed at the
                # origin because they depend on u / r.  Unfortunately,
                # obtaining this ahead of time usually requires some
                # analytical result.
                u__r = vol.dx_left(u)
            else:
                u__r = (vol.interpolate_left(u)
                        / (vol.coordinate + vol.left))

            trE = (1.5 - 0.5/(1.0 + vol.dx_left(u))**2
                   - 1.0 / (1.0 + u__r)**2)

            self.grid[0].tr_r0[vol.number] = (1.0 + u__r)**2*(
                self.Lame_lambda*trE
                + self.mu*(1.0
                           - 1.0 / (1.0 + vol.dx_left(u))**2))

            self.grid[0].tt_r0[vol.number] = ((1.0 + u__r)**2
                                              *self.Lame_lambda*trE
                                              + (1.0 + vol.dx_left(u))
                                              *self.mu*(1.0 + u__r
                                                        - 1.0 / (1.0 + u__r)))

            ## Calculate stress components at center of volume

            if vol.number == 0:
                u__r = vol.dx_left(u)
            else:
                u__r = (vol.interpolate(u)
                        / (vol.coordinate + vol.center))

            trE = (1.5 - 0.5/(1.0 + vol.dx_center(u))**2
                   - 1.0 / (1.0 + u__r)**2)

            self.grid[0].tr_center[vol.number] = (
                    (1.0 + u__r)**2*(
                        self.Lame_lambda*trE
```

(continues on next page)

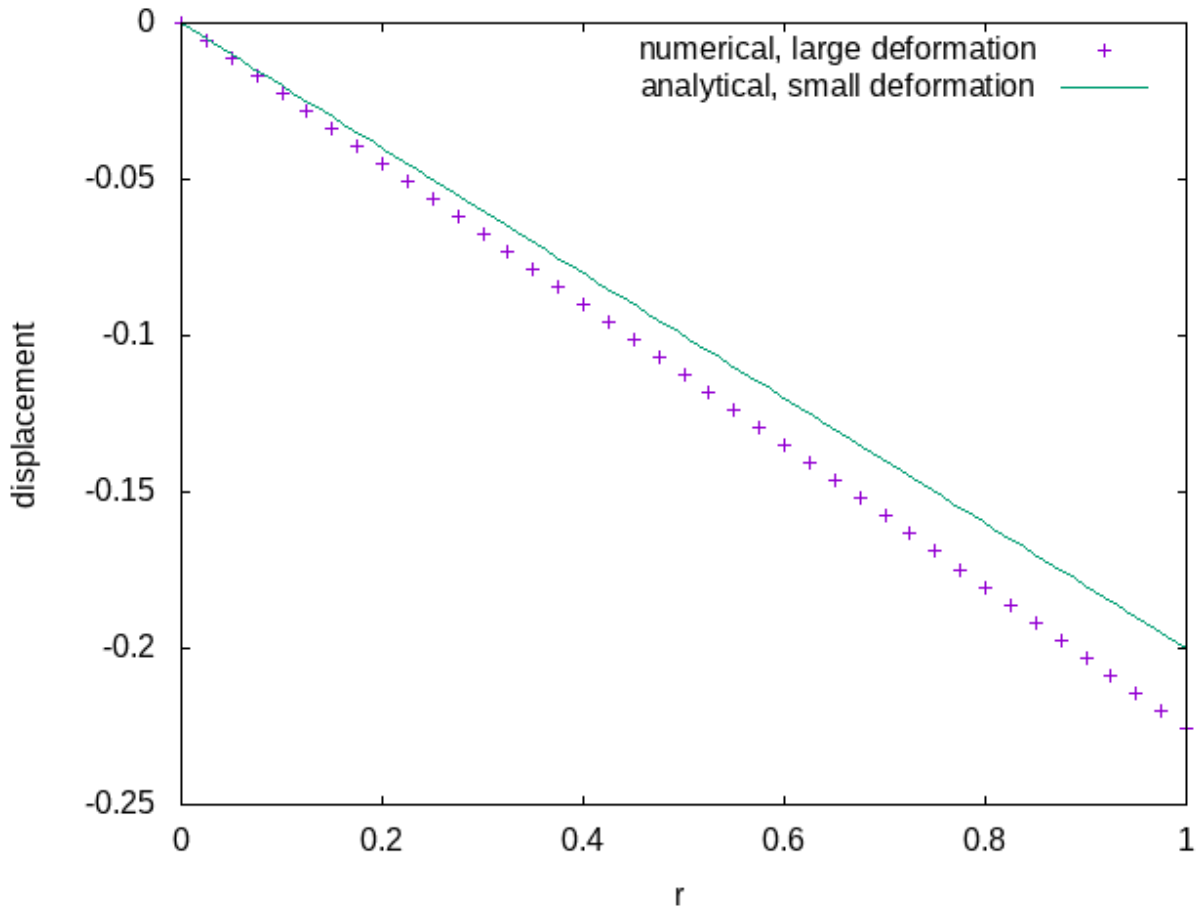
(continued from previous page)

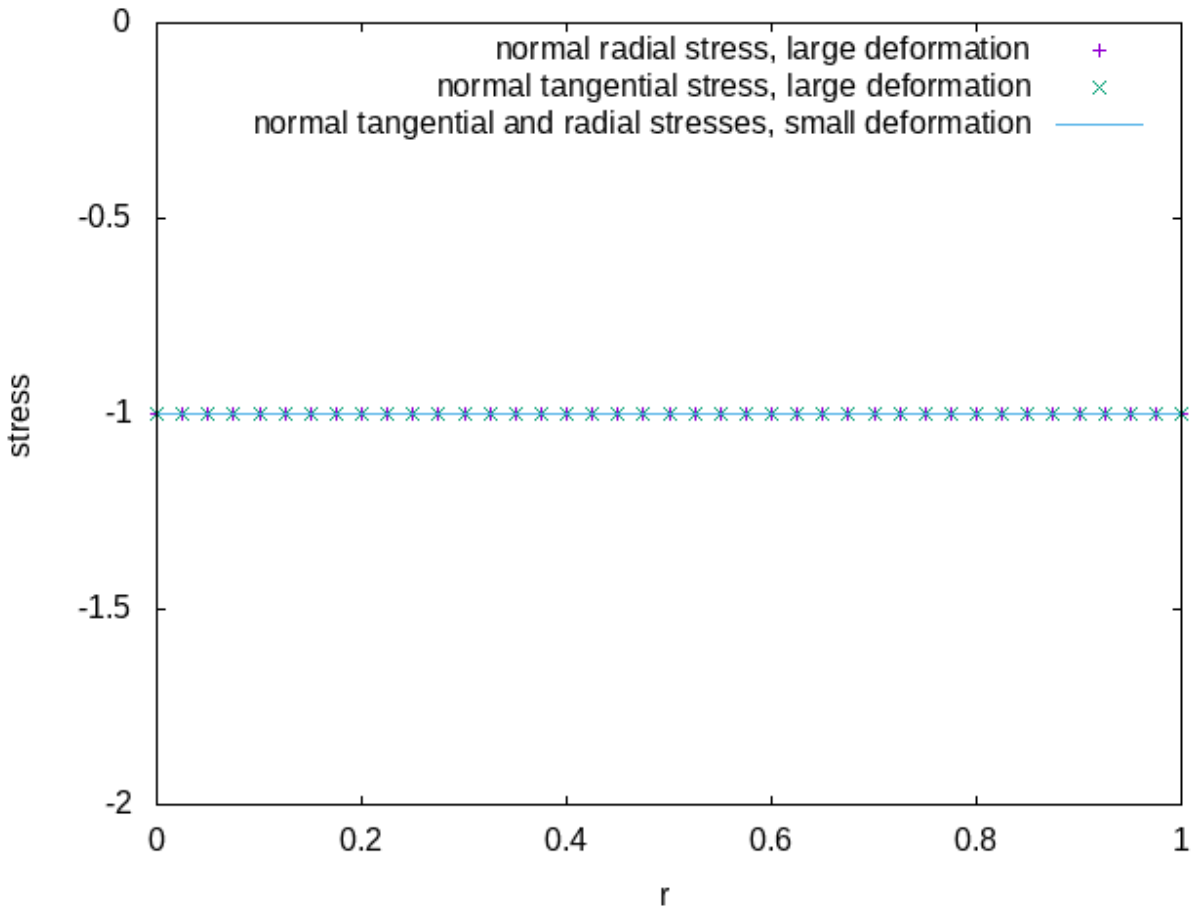
```
                    + self.mu*(
                        1.0 - 1.0 / (
                            1.0 + vol.dx_center(u))**2)))

        self.grid[0].tt_center[vol.number] = ((1.0 + u__r)**2
                                              *self.Lame_lambda*trE
                                              + (1.0 + vol.dx_center(u))
                                              *self.mu*(1.0 + u__r
                                                        - 1.0
                                                        / (1.0 + u__r)))

        ## Calculate stress components at right edge of volume

        u__r = (vol.interpolate_right(u)
                / (vol.coordinate + vol.right))

        trE = (1.5 - 0.5/(1.0 + vol.dx_right(u))**2
               - 1.0 / (1.0 + u__r)**2)

        self.grid[0].tr_r1[vol.number] = (
                    (1.0 + u__r)**2*(
                        self.Lame_lambda*trE
                        + self.mu*(
                            1.0 - 1.0 / (
                                1.0 + vol.dx_right(u))**2)))

        self.grid[0].tt_r1[vol.number] = ((1.0 + u__r)**2
                                          *self.Lame_lambda*trE
                                          + (1.0 + vol.dx_right(u))
                                          *self.mu*(1.0 + u__r
                                                    - 1.0 / (1.0 + u__r)))
```

Second, a method named `set_initial_guess` is defined in the `Grid`-derived class in order to aid the nonlinear solver. PyGDH uses an iterative solver provided by the SciPy package. In general, iterative solvers of this type are not guaranteed to find the solution to a system of nonlinear algebraic equations, such as those produced by PyGDH. Providing a suitable initial guess of the solution can be critical to success of the solver routine. The `set_initial_guess` method gives the user an opportunity to set the "initial" solution data (associated with position `0` in the `field` deque) to an initial guess:

In this case, the solution to the linear, small-deformation problem turns out to be sufficiently close to the solution for the large-deformation problem to enable convergence of the iterative solver.

The displacement differs somewhat from the small-deformation solution, but the stress components are almost the same:


numerical, large deformation
analytical, small deformation
displacement
r
0
-0.05
-0.1
-0.15
-0.2
-0.25
0
0.2
0.4
0.6
0.8
1



3.8.2 Validation

Finding a numerical solution to the discretized equations, as expressed to FVM builder, does not guarantee that the mathematical problem that one intended to solve has indeed been solved. Discretization schemes may be unacceptably inaccurate, and discretized equations and computer programs can contain errors; this is why results must be checked.

In the previous examples, the numerical results were plotted against analytical solutions. However, analytical solutions are frequently unavailable, especially for nonlinear problems–this is why numerical techniques are often needed. Sometimes, a limiting case of a difficult problem will yield an analytical solution for comparison with numerical results, but errors in the solution to the more general problem may go undetected.

One might instead perform additional numerical calculations to check a numerical solution. For example, one might check how well the numerical results satisfy other discretized forms of the original problem. While this cannot guarantee that the solution is free of error, it is helpful for detecting errors in many situations.

The methods for customizing output, as introduced in the previous chapter, can be used for this validation process. Array variables holding one element per `Volume` in a `Grid`, representing the error for some field one that `Volume`, are defined in the `define_field_variables` method. These variables are then marked for output in the `set_output_fields` method. Finally, the `process_solution` method is defined. This method is automatically called after a solution is computed; the computed solution is used to evaluate an alternative set of discretized equations, and the results are stored in the arrays created for error output. The results are automatically saved to the output files along with the solution field variable values.

One can obtain an alternative set of discretized equations in any number of ways. One might revisit the finite volume formulation, making more or less accurate approximations in various terms of the governing equations. Or one could

use a different approach to discretization. For example, the "finite difference method" approximates an equation with values at a finite set of points. Derivatives in time and space are approximated by "finite difference" formulas that incorporate information from neighboring points. Although the finite volume method also makes use of finite difference formulas, solution values in the finite volume method represent the behavior of functions throughout volumes, and integration over volumes can lead to very different discrete forms of the governing equations than would be obtained by the finite difference method.

Discretized equations for the finite difference method are typically obtained by replacing spatial derivatives in the differential equation with finite difference formulas that approximate these spatial derivatives. Time discretization is performed in the same way as with the finite volume method.

The alternative set of discretized equations can be expressed to PyGDH in a similar way to the original set of discretized equations. As before, PyGDH provides methods that approximate spatial derivatives. Time discretization is the responsibility of the user, although PyGDH manages storage of past solutions needed in time-differencing formulas.

Although it is tempting to use intermediate values computed by `calculate_global_values` in this validation process, these intermediate calculations may also contain errors. For this reason, it is best to recompute any intermediate values needed for the alternative discretized equations, with a completely different body of code that is not copied from the finite volume method implementation. While this doesn't eliminate the possibility of error in the intermediate calculations, it is unlikely that the same minor errors will be made in both cases.

The terms of the discretized equation are rearranged as in the finite volume equations used to obtain the solution, with all nonzero terms collected on one side. The `process_solution` implementation, including the copying of computed stresses to the output arrays is:

```
    # This routine is called once after a solution is obtained
    # (on each timestep in a time-dependent problem)
    def process_solution(self):

        grid0 = self.grid[0]

        grid0.tr_output[0] = grid0.tr_r0[0]
        grid0.tt_output[0] = grid0.tt_r0[0]

        for vol_number in range(1,grid0.unit_count-1):
            grid0.tr_output[vol_number] = grid0.tr_center[vol_number]
            grid0.tt_output[vol_number] = grid0.tt_center[vol_number]

        grid0.tr_output[-1] = grid0.tr_r1[-1]
        grid0.tt_output[-1] = grid0.tt_r1[-1]

        tt = numpy.empty(grid0.unit_count)
        tr = numpy.empty(grid0.unit_count)
        u = grid0.field[0][0]
        for vol in grid0.unit_with_number:
            if vol.number == 0:
                u__r = vol.dx_center(u)
            else:
                u__r = u[vol.number] / (vol.coordinate + vol.center)
            trE = (1.5 - 0.5/(1.0 + vol.dx_center(u))**2
                   - 1.0/(1.0 + u__r)**2)
            tt[vol.number] = (
                (1.0 + u__r)**2
                *(self.Lame_lambda*trE
                  + self.mu
                  *(1.0 - 1.0 / (1.0 + vol.dx_center(u))**2)))
            tr[vol.number] = (
```

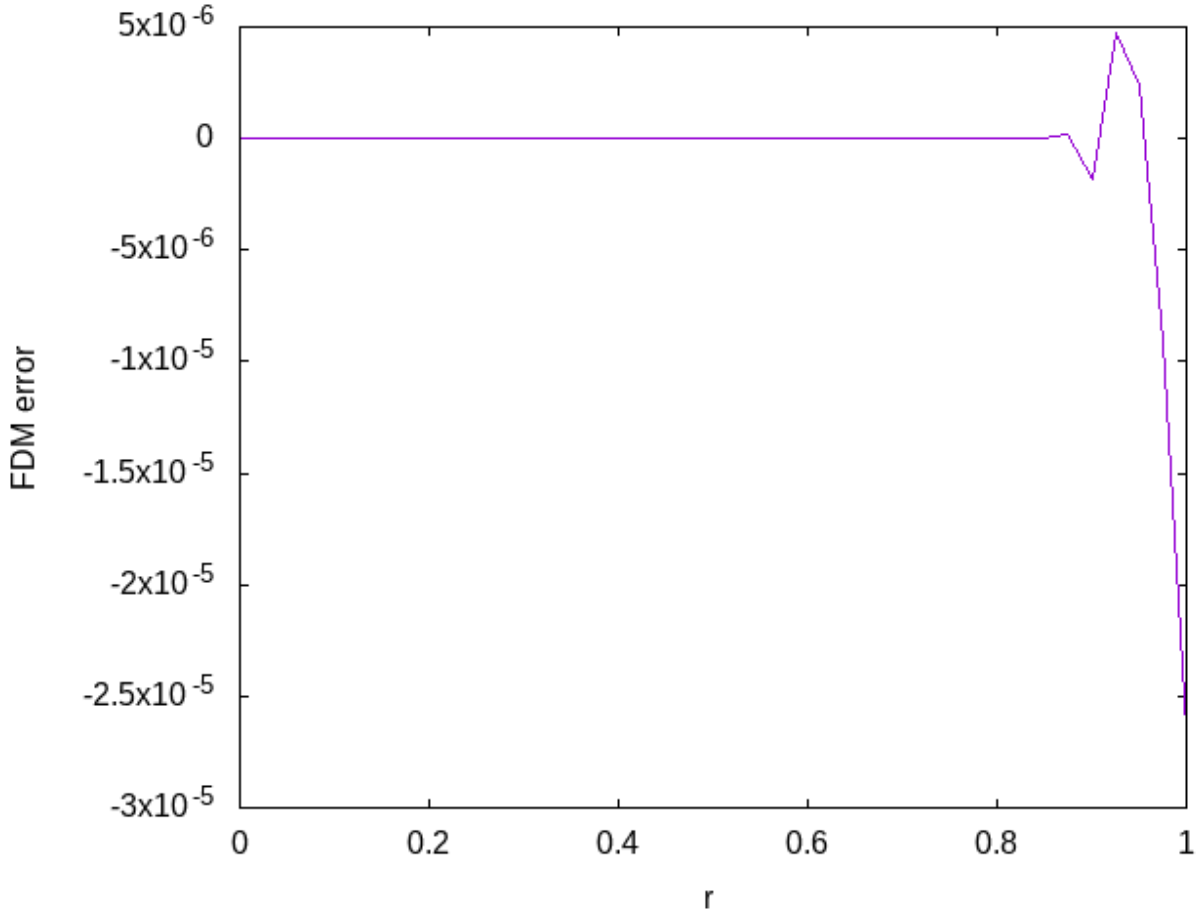
(continues on next page)

(continued from previous page)

```
            (1.0 + u__r)**2*self.Lame_lambda*trE
            + (1.0 + vol.dx_center(u)) * self.mu
            *(1.0 + u__r - 1.0 / (1.0 + u__r)))

        tol = 1e-9

        # No equation was solved in the volume next to the origin
        grid0.validation_residual[0] = 0.0
        for vol_number in range(1,grid0.unit_count):
            vol = grid0.unit_with_number[vol_number]
            grid0.validation_residual[vol_number] = (vol.dx_center(tt)
                - 2.0*(tt[vol.number] - tr[vol.number])
                / (vol.coordinate + vol.center))
        for vol_number in range(1,grid0.unit_count):
            assert grid0.validation_residual[vol_number]**2 < tol
```

On a `Volume` for which the value of the solution field is prescribed, there is no reason to evaluate the alternative form of the governing equation–the numerical method has no freedom to modify the value associated with the present `Volume`, and no equation was ever evaluated on this `Volume`. In this example, the value of the displacement at the origin is prescribed, so the error at the `Volume` at the origin is simply set to zero.

The error in satisfying the governing equation, as discretized by the finite difference method, when using the numerical solution computed by the finite volume method, is plotted below:



A detailed analysis of the expected error magnitude is rather involved, but errors computed by this method should be

much smaller than the contribution from the smallest term in the discretized equation. Where terms cancel, as in this particular case, terms contained within these terms should be examined. In this case, the stresses have the magnitude of the given pressure, and the radial distances have a maximum value of 1, so it is clear that the computed errors are reassuringly small.

3.8.3 Suggested exercises

1. Perform a similar validation for the original ODE example.

3.9 Solving systems of equations

In the previous chapters, each `Volume` was associated with a single governing equation and a single solution variable. PyGDH may also be used to solve systems of coupled governing equations on a spatial domain. Typically, this requires only that additional discretized equation methods be supplied, and that the additional solution fields, equations, and boundary conditions be declared. The example code is included in the PyGDH archive as `pygdh/examples/system/system.py`.

3.9.1 Diffusion with reaction

The one-dimensional form of Fick's second law will be used to describe the diffusion of two species in the dilute limit. These two species can react in a homogeneous reaction with a rate equation that is first-order in both concentrations:

$$\frac{\partial c_1}{\partial t} = D_1 \frac{\partial^2 c_1}{\partial x^2} - k c_1 c_2$$
$$\frac{\partial c_2}{\partial t} = D_2 \frac{\partial^2 c_2}{\partial x^2} - k c_1 c_2$$

with zero fluxes at one end of the domain and equal nonzero fluxes at the other end:

$$-D_1 \frac{\partial c_1}{\partial x}(0) = -D_2 \frac{\partial c_2}{\partial x}(0) = 0$$
$$D_1 \frac{\partial c_1}{\partial x}(1) = D_2 \frac{\partial c_2}{\partial x}(1) = N$$

and parabolic initial conditions that are consistent with the boundary conditions:

$$c_1(x) = 1 + \frac{N(D_1 - D_2)}{6 D_1 D_2} + \frac{N}{2 D_1} x^2$$
$$c_2(x) = 1 + \frac{N}{2 D_2} x^2$$

The constant terms are chosen so that the domain initially contains equal amounts of species 1 and 2.

3.9.2 Problem domain

Since there are now two unknown solution variables, both must be named in the `'field_names'` entry in the grid description. The variable named `'c1'` is the element with index 0 in this list, and the variable named `'c2'` has index 1. These same associations of solution variables and indices will be recognized by PyGDH throughout the rest of the program.

```
    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_grid' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, name, unit_count):

        self.initialize_grid(name,
                             coordinate_values = {
                                 'x' :  numpy.linspace(0.0, 1.0, unit_count)},
                             coordinate_order=['x'],
                             field_names=['c1','c2'],
                             # Each unit in 'Domain' will be described by a
                             # 'Volume' object
                             unit_classes=[Volume for i in range(unit_count)],
                             stored_solution_count=2)
```

3.9.3 Discretized equations

Rearranging each equation into the form required by PyGDH (collecting all nonzero terms on one side), and integrating each equation over a volume with lower boundary $x = a$ and upper boundary $x = b$, one obtains

$$\frac{\partial \langle c_1 \rangle}{\partial t}(b-a) - D_1\left(\frac{\partial c_1}{\partial x}(b) - \frac{\partial c_1}{\partial x}(a)\right) + k\langle c_1 c_2 \rangle (b-a) = 0$$
$$\frac{\partial \langle c_2 \rangle}{\partial t}(b-a) - D_2\left(\frac{\partial c_2}{\partial x}(b) - \frac{\partial c_2}{\partial x}(a)\right) + k\langle c_1 c_2 \rangle (b-a) = 0$$

where the angle brackets indicate that the value used is an average over the volume. In the corresponding methods, a first-order implicit time-differencing scheme will be used.

One can represent these equations with two discretized equation methods, one for each governing equation, that compute and store scalar values. Alternatively, one may define a single method that computes and stores a vector with two components, each a scalar value. This is a more efficient approach when the two governing equations rely on some of the same intermediate calculations. The choice is unimportant in this example, but the latter case is used for illustration:

```
    # This describes a "vector-valued" equation, storing two scalar values
    def pdes(self, vol, residuals):

        c1 = self.grid[0].field[0][0]
        c2 = self.grid[0].field[0][1]

        c1_1 = self.grid[0].field[-1][0]
        c2_1 = self.grid[0].field[-1][1]

        residuals[0] = ((c1[vol.number] - c1_1[vol.number]) * vol.volume
                        * self.inverse_timestep_size
                        - self.D_1*(vol.dx_right(c1) - vol.dx_left(c1))
                        + self.k*vol.interpolate(c1)
                        * vol.interpolate(c2)*vol.volume)
        residuals[1] = ((c2[vol.number] - c2_1[vol.number]) * vol.volume
                        * self.inverse_timestep_size
                        - self.D_2*(vol.dx_right(c2) - vol.dx_left(c2))
                        + self.k*vol.interpolate(c1)
                        * vol.interpolate(c2)*vol.volume)
```

Note that the indices of the `self.grid[0].field[0]` two-dimensional arrays identify the solution variable of interest, and that their numerical labels are consistent with their order in the `'field_names'` entry of `initialize_grid`.

Similar methods are needed to incorporate the flux boundary conditions. For illustration, one method will represent both equations at one boundary, and two methods will be used to represent each equation at the other boundary.

3.9.4 Equations and unknowns

As described in the `declare_unknowns` method,

```
    # This method is required. It informs PyGDH of the unknown quantities for
    # for which it must solve.
    def declare_unknowns(self):

        # "Left" boundary
        vol = self.grid[0].unit_with_number[0]
        self.unknown[0][0,vol.number] = True
        self.unknown[0][1,vol.number] = True

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            vol = self.grid[0].unit_with_number[vol_number]
            self.unknown[0][0,vol.number] = True
            self.unknown[0][1,vol.number] = True

        # "Right" boundary, value to be copied from "Left" boundary of Right
        # grid
        vol = self.grid[0].unit_with_number[-1]
        self.unknown[0][0,vol.number] = True
        self.unknown[0][1,vol.number] = True
```

the governing equations must be evaluated for all `Volume` objects, as the boundary conditions are all flux conditions, so there are no given boundary values and each `Volume` is associated with two unknown field values. The `unknown_field` index corresponds to the position of each solution variable name in the `'field_names'` parameter of `initialize_grid`.

The choice of representations for the governing equations is reflected in `assign_equations`:

```
    def assign_equations(self):

        domain_equations = self.equations[0]
        # "Left" boundary
        domain_equations[0] = [self.no_flux_pdes]

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            domain_equations[vol_number] = [self.pdes]

        # "Right" boundary
        domain_equations[-1] = [self.c1_inlet_pde, self.c2_inlet_pde]
```

The number of scalars associated with each method is noted in `equation_scalar_counts`, and the elements of `equations[0]` specify the methods that PyGDH must run in order to evaluate the required governing equations.

At the no-flux boundary and in the interior of the domain, the solution behavior is governed by discretized equation methods that each store two scalars, one for each of the unknown values associated with each `Volume`.

At the "inlet" boundary, two discretized equation methods that each store one scalar are provided, again one for each of the unknown values associated with each `Volume`.

3.9.5 Initial conditions

The initial conditions for each solution variable are entered in a straightforward way. As before, the last index provided in each of the local variable definitions corresponds to the position of each solution variable name in the `'field_names'` entry of the grid description given at initialization.

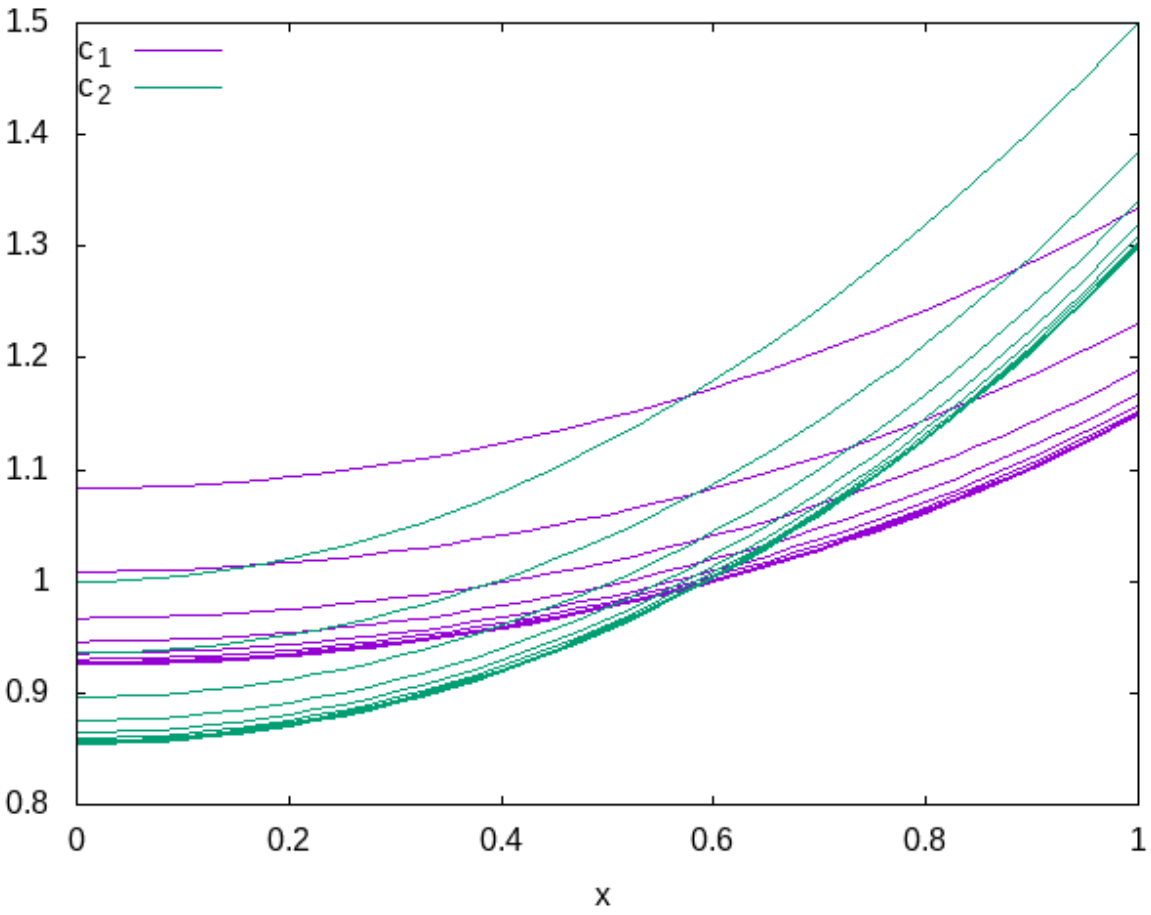
```
    # This specifies the initial conditions for this time-dependent
    # problem
    def set_initial_conditions(self):
        c1 = self.grid[0].field[0][0]
        c2 = self.grid[0].field[0][1]
        for vol in self.grid[0].unit_with_number:
            x = vol.coordinate
            c1[vol.number] = (
                1.0 + self.N*(self.D_1 - self.D_2)/(6.*self.D_1*self.D_2)
                + 0.5*self.N*x**2/self.D_1)
            c2[vol.number] = 1.0 + 0.5*self.N*x**2/self.D_2
```

3.9.6 Solution

The solution is then obtained with the parameter values `N = 1.0`, `D_1 = 1.0`, `D_2 = 2.0`, and `k = 1.0`:

```
# Create an object instance
volume_count = 41
grid_obj = Domain('domain',volume_count)

N = 1.
D_1 = 2.
D_2 = 1.
k = 1.
problem = System([grid_obj], N, D_1, D_2, k)

# The names of all output files will begin with this string
filename_root = 'system'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

timestep_size = 0.5
timesteps = 10

# Calculate solution and write output to file
problem.solve_time_dependent_system(filename_root, output_types,
                                    timestep_count=timesteps,
                                    timestep_size=timestep_size)
```

The results are plotted for ten timesteps, with both curves decreasing over time toward a steady solution:



3.9.7 Validation

The validation process is similar to that discussed in the nonlinear PDE example. As before, array variables (that can hold an error measurement for every `Volume`) are defined once for the entire program, and marked for inclusion in the output files:

These arrays are then filled with error estimates:

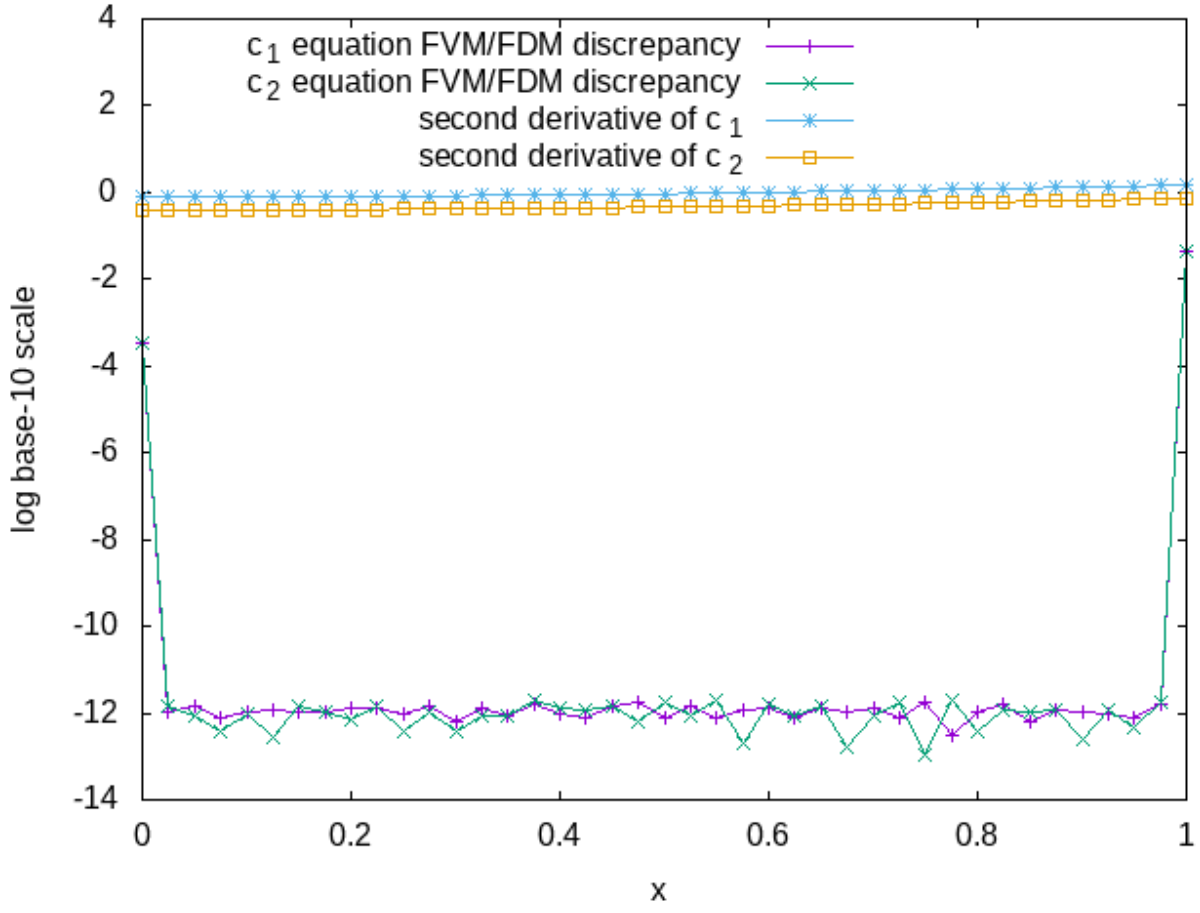
```
    # This routine is called once after a solution is obtained
    # (on each timestep in a time-dependent problem)
    def process_solution(self):

        grid0 = self.grid[0]

        c1 = grid0.field[0][0]
        c2 = grid0.field[0][1]
        c1_1 = grid0.field[-1][0]
        c2_1 = grid0.field[-1][1]

        for vol_number in range(grid0.unit_count):
            vol = grid0.unit_with_number[vol_number]

            grid0.d2c1__dx2[vol_number] = vol.d2x_center(c1)
```

(continues on next page)

(continued from previous page)

```
            grid0.d2c2__dx2[vol_number] = vol.d2x_center(c2)

            grid0.c1_fdm_residual[vol_number] = (
                (c1[vol_number] - c1_1[vol_number])
                * self.inverse_timestep_size
                - self.D_1*grid0.d2c1__dx2[vol_number]
                + self.k*c1[vol_number] * c2[vol_number])

            grid0.c2_fdm_residual[vol_number] = (
                (c2[vol_number] - c2_1[vol_number])
                * self.inverse_timestep_size
                - self.D_2*grid0.d2c2__dx2[vol_number]
                + self.k*c1[vol_number] * c2[vol_number])
```

The base-10 logarithm of the error magnitude (relative to a finite-difference form of the discretized equations) is plotted below for the solution at final timestep:



Since the system is clearly approaching a steady state, the time derivative terms are expected to approach zero, so their magnitudes relative to the discrepancies are not of concern.

From the computed solution, it is apparent that the reaction terms are on the order of 1, and therefore much larger than the discrepancies. The magnitudes of the Laplacian terms are harder to estimate, so the numerical approximations have been included in the program output. These too are significantly larger than the discrepancies.

The discrepancies at the boundaries are much larger than those in the interior; the solution values here are strongly

linked to the alternative methods used in the boundary `Volume` elements, and the influence of neighboring solution values comes only from one direction. Nevertheless, these discrepancies are not of concern, as they are still much smaller that the values of the terms in the discretized equation, and one can demonstrate that their magnitudes decrease as the number of `Volumes` is increased. In this case, the discretized equation error scales is known to scale linearly with the grid spacing, so doubling the number of `Volumes` should halve the discrepancies, and this is indeed observed for the relatively large errors at the boundaries.

Both plots were produced with the following GNUPLOT script:

```
set term png enhanced
set output 'system.png'
set key left top
set xlabel 'x'
plot 'system_domain.gnuplot' using 1:2 title 'c_1' with lines, 'system_domain.gnuplot
↪' using 1:3 title 'c_2' with lines
set output 'system_log_validation.png'
set ylabel 'log base-10 scale'
plot [][-14:4] 'system_domain.gnuplot' using 1:(log10(abs($5))) index 10 title 'c_1␣
↪equation FVM/FDM discrepancy' with linespoints, 'system_domain.gnuplot' using␣
↪1:(log10(abs($4))) index 10 title 'c_2 equation FVM/FDM discrepancy' with␣
↪linespoints, 'system_domain.gnuplot' using 1:(log10(abs($7))) index 10 title
↪'second derivative of c_1' with linespoints, 'system_domain.gnuplot' using␣
↪1:(log10(abs($6))) index 10 title 'second derivative of c_2' with linespoints
```

3.9.8 Suggested exercises

1. Confirm that the FVM/FDM discrepancy magnitudes at the boundaries decrease linearly with the number of domain elements.

3.10 Postprocessing

Sometimes, the standard human-readable output formats are too limiting–every dataset contains one entry for each `Volume` in a `Grid`, identified by the `coordinate` value(s) of the `Volume`. For this reason, PyGDH includes postprocessing capabilities, with which users can write programs to extract and manipulate data stored in HDF5 or PICKLE files.

The postprocessing programs take stored solutions and restore them into the same data structures that were used in the original solver programs. This strategy ensures that postprocessing with PyGDH follows naturally from writing solver programs; few new concepts are required. Also, it provides a simple mechanism for restarting time-dependent simulations from stored results.

The existing GNUPLOT or CSV output facilities remain available to postprocessing programs, but users may generate human-readable results in their own formats; the postprocessing facilities allow users to work with the numerical solutions in a more flexible way than one can by defining a `process_solution` method and setting additional output fields.

3.10.1 Program organization

In each of the examples given in previous chapters, the problem class definition is followed by statements that create an object of that class and direct that object to find solutions. However, postprocessing should be performed by a separate program, because there is no need to recompute the solution when only the postprocessing step is modified. Since the PyGDH postprocessing strategy is to read saved solutions into the same data structures with which they were originally

computed, the data structures must be defined in both the original solver program and in the corresponding postprocessing program. Rather than maintaining two similar source code files, it is recommended that users put almost all of the source code into a common file (in this example, `pygdh/examples/postprocessing/PDE_body.py`). Small additional files (in this example, `pygdh/examples/postprocessing/PDE_solve.py` and `pygdh/examples/postprocessing/PDE_postprocess.py`) can then access this common source code and use it to either compute numerical solutions or to postprocess saved solutions. This strategy will also be useful when making use of the Cython compiler, as described in an *upcoming chapter*.

As an illustration, the program presented in the previous chapter can be split into two source code files. The first, called `pygdh/examples/postprocessing/PDE_body.py`, defines the entire mathematical problem except for some high-level parameter values:

```
import pygdh
import numpy
import math

# Objects of this class are the units from which the spatial domain will be
# constructed.
class Volume(pygdh.Volume):

    # Defining this method gives users an opportunity to define their own
    # mathematical operators, for later use in describing governing equations.
    def define_interpolants(self):

        # Operator for estimating first derivative at left boundary of volume
        self.dx_left = self.default_interpolant(-1.0, 1, 1)

        # Operator for estimating function value at the volume center
        self.interpolate = self.default_interpolant(0.0, 0, 2)

        # Operator for estimating first derivative at right boundary of volume
        self.dx_right = self.default_interpolant(1.0, 1, 1)

# Objects of this class describe the spatial domains on which problems will be
# solved.
class Domain(pygdh.Grid):

    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_grid' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, name, alpha, volume_count):

        # Save the argument value to an object member
        self.alpha = alpha

        self.initialize_grid(name,
                             # The independent variable will be named 'y'
                             field_names=['y'],
                             # The dependent variable will be named 'x',
                             # and 'y' will be computed at the specified number
                             # of equally-spaced values of 'x'
                             coordinate_values={
                                 'x': numpy.linspace(0.0, 1.0, volume_count)},
                             # Each unit in 'Domain' will be described by a
                             # 'Volume' object
                             unit_classes=[Volume for i in range(volume_count)],
                             stored_solution_count=2)
```

(continues on next page)

(continued from previous page)

```
# Objects of this class bring the spatial domain definitions together with
# equations, boundary conditions, and for time-dependent problems, initial
# conditions. These objects can then be directed to compute numerical
# solutions.
class PDE(pygdh.Problem):

    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_problem' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, alpha, volume_count):

        # Save the argument value as an object member
        self.alpha = alpha

        # Create the Grid object representing the problem domain
        grid_obj = Domain('domain', alpha, volume_count)

        # Prepare to run simulation on new Grid object
        self.initialize_problem([grid_obj],stored_solution_count=2)

    # This sets the initial values of the independent variables on the region
    # represented by an object of this class.
    def set_initial_conditions(self):

        # Defined for clarity
        y = self.grid[0].field[0][0]

        # Set the value of the independent variable on each volume
        for vol in self.grid[0].unit_with_number:
            y[vol.number] = (
                vol.coordinate / self.grid[0].alpha
                - 0.5*math.sin(0.5*math.pi*vol.coordinate))

    # This method is required. It informs PyGDH of the unknown quantities for
    # for which it must solve.
    def declare_unknowns(self):

        # "Left" boundary
        vol = self.grid[0].unit_with_number[0]
        self.unknown[0][0,vol.number] = False

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            vol = self.grid[0].unit_with_number[vol_number]
            self.unknown[0][0,vol.number] = True

        # "Right" boundary
        vol = self.grid[0].unit_with_number[-1]
        self.unknown[0][0,vol.number] = True

    # These methods represent the governing equation and are named and defined
    # by the user. These must accept three particular parameters and must store
    # results in the data structure associated with the last parameter.

    def pde(self, vol, residual):
```

(continues on next page)

(continued from previous page)

```
        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]

        # FVM representation of governing equation
        residual[0] = ((vol.interpolate(y0)
                        - vol.interpolate(y_1))*vol.volume
                       - self.timestep_size * self.alpha
                       * (vol.dx_right(y0)
                          - vol.dx_left(y0)))

    def pde_boundary(self, vol, residual):

        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]

        # FVM representation of governing equation, incorporating boundary
        # condition
        residual[0] = ((vol.interpolate(y0)
                        - vol.interpolate(y_1))*vol.volume
                       + self.timestep_size
                       * (-1.0 + self.alpha * vol.dx_left(y0)))

    # This method is used to define boundary conditions in which the solution
    # value is specified. This must be consistent with the contents of the
    # 'declare_unknowns' methods defined in the corresponding 'Grid' objects.
    def set_boundary_values(self):

        # Defined for clarity; there is only one 'Grid' object, with index 0,
        # and one solution field 'y', with index 0
        y = self.grid[0].field[0][0]

        # "Left" boundary
        y[0] = 0.0

    # This is a required method that notifies PyGDH about the number of scalar
    # values returned by each governing equation method.
    def set_equation_scalar_counts(self):

        # This is a dictionary associating governing equation methods with
        # the number of scalar values that they return
        self.equation_scalar_counts = {self.pde: 1,
                                       self.pde_boundary: 1}

    # This is a required method that notifies PyGDH about the methods that
    # describe governing equations, and indicates where in the domain the
    # equations apply. This must be consistent with the 'declare_unknowns'
    # methods of the corresponding 'Grid' objects.
    def assign_equations(self):

        # Defined for clarity, equations to be defined on the only Grid,
```

(continues on next page)

(continued from previous page)

```
        # with index 0
        domain_equations = self.equations[0]

        # "Left" boundary
        domain_equations[0] = []

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            domain_equations[vol_number] = [self.pde]

        # "Right" boundary
        domain_equations[-1] = [self.pde_boundary]
```

The second, called `pygdh/examples/postprocessing/PDE_solve.py` merely imports the common source code, creates a `pygdh.Problem`-derived object, defines parameter values, and tells the object to solve the mathematical problem:

```
import PDE_body

# Create an object instance with 11 volumes and setting alpha = 1.0
problem = PDE_body.PDE(1.0, 11)

number_of_timesteps = 3
timestep_size = 0.01

# The names of all output files will begin with this string
filename_root = 'PDE_postprocess'

# One output file will be created for each format given here
output_types = ['HDF5']

# Calculate solution and write output to file
problem.solve_time_dependent_system(filename_root, output_types,
                                    timestep_count=number_of_timesteps,
                                    timestep_size=timestep_size)
```

Note that one must now indicate that the `PDE` class is in the imported `PDE_body` module, and that the `output_types` list has been changed to include the `'HDF5'` option. The resulting data file will be used by the postprocessing program.

Another benefit of this arrangement is that the small files like `PDE_solve.py` can serve as records of the parameters used to obtain a stored solution.

3.10.2 An example using matplotlib

Postprocessing programs can make use of any capabilities available through Python. This examples makes use of the matplotlib library, which allows sophisticated plots to be created from within the Python environment.

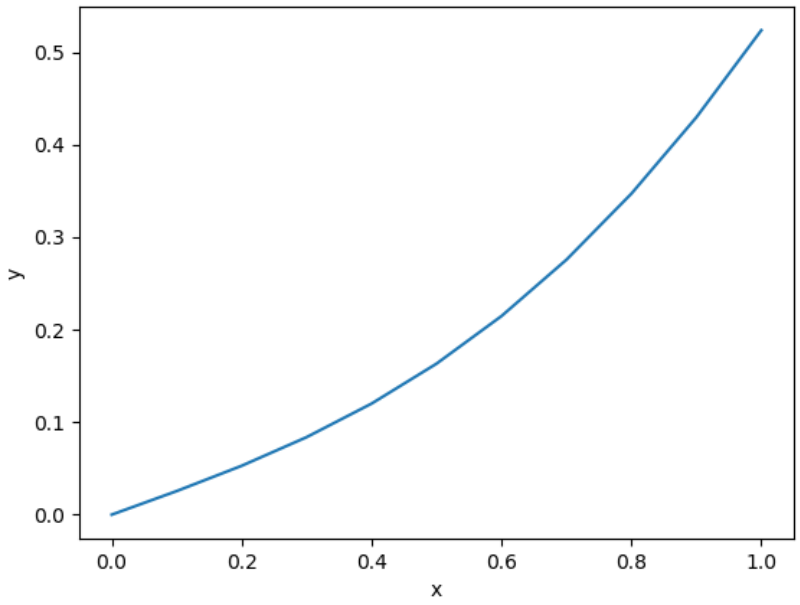
As in `pygdh/examples/postprocessing/PDE_solve.py`, the postprocessing program, `pygdh/examples/postprocessing/PDE_postprocess.py` begins by creating a `pygdh.Problem`-derived object, which in turn creates a `pygdh.Grid`-derived object of a size that is compatible with that used to obtain the stored solutions. The `PDE` object is then instructed to load the HDF5 or PICKLE file containing saved solutions, and then asked to restore its internal state to that immediately after the solution for the second timestep was computed.

```
# Create an object instance with 11 volumes and setting alpha = 1.0
problem = PDE_body.PDE(1.0, 11)

# Load the saved solution file
filename_root = 'PDE_postprocess'
problem.load_solutions(filename_root)

problem.restore_from_timestep(2)
```

The specified solution is then accessible just as it was when it was first computed by the solver. Creating a simple plot with matplotlib can involve little more than specifying the data to be plotted:

```
# Plot with matplotlib
import matplotlib.pyplot as plt

plt.plot(problem.grid[0].coordinates_list[0],problem.grid[0].field[0][0])
plt.xlabel('x')
plt.ylabel('y')
plt.savefig(filename_root + '.png')
plt.show()
```

This produces the following image:

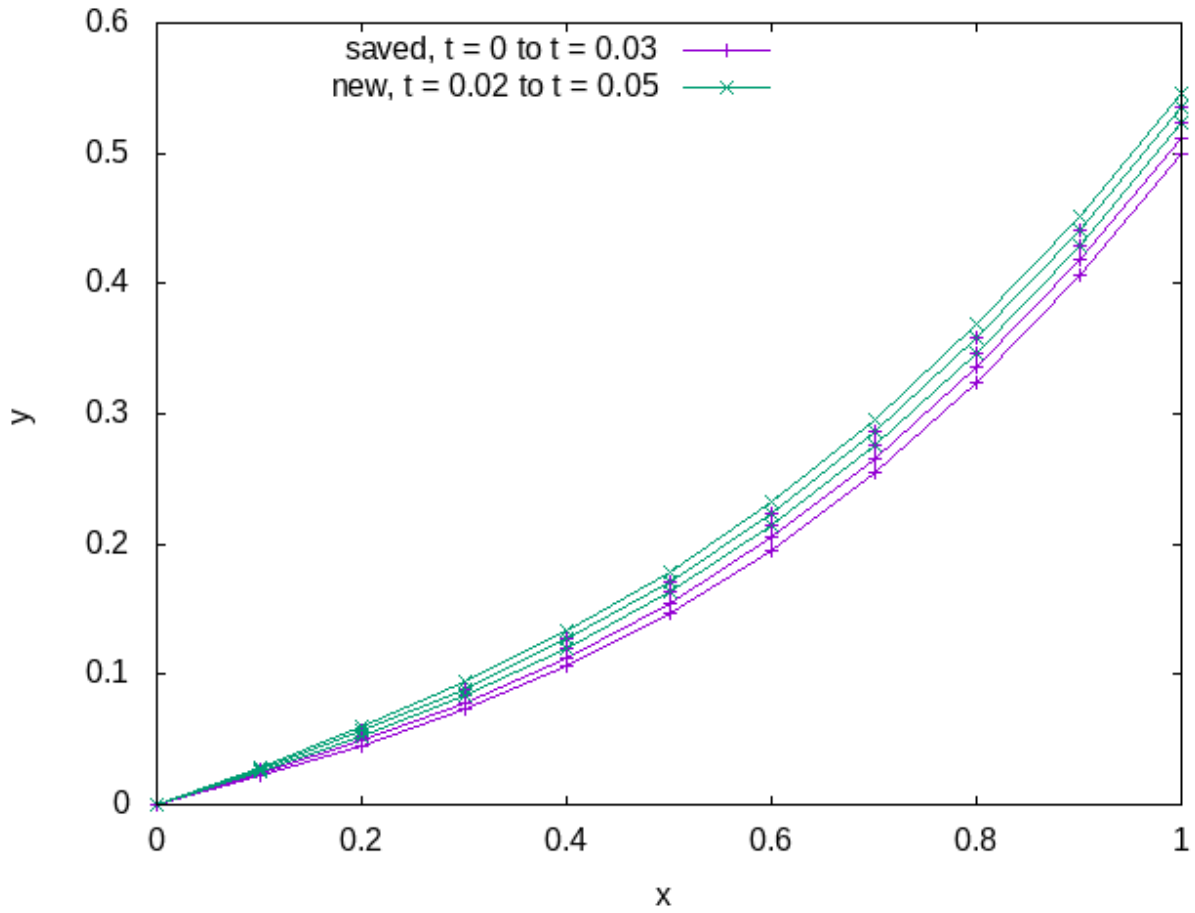


3.10.3 Restarting time-dependent simulations

Since postprocessing in PyGDH involves restoring `pygdh.Problem`-derived objects to their states following the calculation of saved numerical solutions, restarting time-dependent simulations from any saved solution is a simple matter. Restoring an object to a saved state restores not only the "present" solution, but the "past" solutions that are also needed to compute future solutions according to the timestepping scheme used. The following program computes three additional timesteps for the PDE example:

```
import PDE_body

# Create an object instance with 11 volumes and setting alpha = 1.0
problem = PDE_body.PDE(1.0, 11)

number_of_timesteps = 2
timestep_size = 0.01

saved_solution_filename_root = 'PDE_postprocess'

# Load the saved solution file
problem.load_solutions(saved_solution_filename_root)

problem.restore_from_timestep(2)

restart_solution_filename_root = 'PDE_restart'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

# Calculate solution and write output to file
problem.solve_time_dependent_system(restart_solution_filename_root,
                                    output_types,
                                    timestep_count=number_of_timesteps,
                                    timestep_size=timestep_size)
```

PyGDH knows to restart the simulation from the specified timestep because a saved solution file has been loaded. The saved and new solutions are shown in the following figure:



Below is the GNUPLOT script used to generate this figure:

```
set term png
set output 'PDE_restart.png'
set key left top
set xlabel 'x'
set ylabel 'y'
plot 'PDE_domain.gnuplot' index 0:3 title 'saved, t = 0 to t = 0.03' with linespoints,
↪ 'PDE_restart_domain.gnuplot' title 'new, t = 0.02 to t = 0.05' with linespoints
set output
```

HDF5 output files are automatically used in preference to PICKLE files if both are available because the HDF5 format can be accessed much more efficiently. However, the same capabilities are available through the PICKLE format. This is an example using files with slightly different names, so that the HDF5 file generated earlier is not used in preference to the PICKLE file.

```
import PDE_body

# Create an object instance with 11 volumes and setting alpha = 1.0
problem = PDE_body.PDE(1.0, 11)

number_of_timesteps = 3
timestep_size = 0.01

# The names of all output files will begin with this string
```

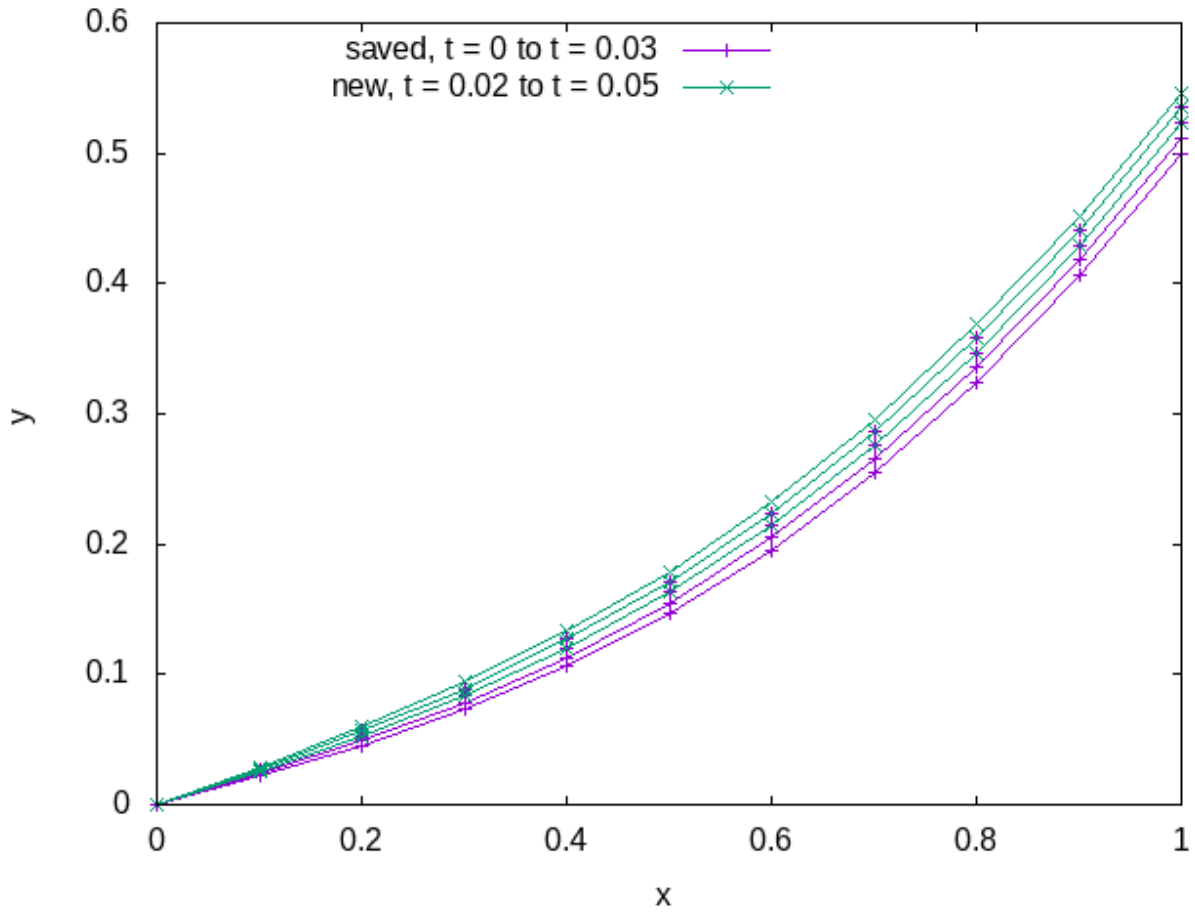
(continues on next page)

(continued from previous page)

```
filename_root = 'PDE_pickle_postprocess'

# One output file will be created for each format given here
output_types = ['PICKLE']

# Calculate solution and write output to file
problem.solve_time_dependent_system(filename_root, output_types,
                                    timestep_count=number_of_timesteps,
                                    timestep_size=timestep_size)
```

```
import PDE_body

# Create an object instance with 11 volumes and setting alpha = 1.0
problem = PDE_body.PDE(1.0, 11)

number_of_timesteps = 2
timestep_size = 0.01

saved_solution_filename_root = 'PDE_pickle_postprocess'

# Load the saved solution file
problem.load_solutions(saved_solution_filename_root)

problem.restore_from_timestep(2)

restart_solution_filename_root = 'PDE_pickle_restart'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

# Calculate solution and write output to file
problem.solve_time_dependent_system(restart_solution_filename_root,
                                    output_types,
                                    timestep_count=number_of_timesteps,
                                    timestep_size=timestep_size)
```

```
set term png
set output 'PDE_pickle_restart.png'
set key left top
set xlabel 'x'
set ylabel 'y'
plot 'PDE_domain.gnuplot' index 0:3 title 'saved, t = 0 to t = 0.03' with linespoints,
↪ 'PDE_pickle_restart_domain.gnuplot' title 'new, t = 0.02 to t = 0.05' with␣
↪linespoints
set output
```



3.10.4 Suggested exercises

1. Revisit the large-deformation solid mechanics problem

1. Split the source code into two files, one containing a program that solves the mathematical problem by making use of the source code imported from the second file, which contains source code that will be needed by the postprocessing program as well.
2. The output was originally written in terms of the material positions, the positions of material "particles" before displacement. Write a postprocessing program that reads in the saved solution and writes out the same data in written in terms of the spatial positions, the displaced particle positions.
3. Modify the postprocessing program to produce a plot of spatial positions as a function of material positions.

3.11 Coupled domains

This chapter demonstrates a problem for which two `Grid` objects are created, with separate equations simultaneously solved over each `Grid` and with coupling between the solutions.

Employing multiple `Grid` objects, as introduced in this chapter, can be useful in many other situations. For example, as discussed in an earlier chapter, a `Grid` object containing a single spatial unit can be used to solve time-dependent ODEs. With multiple grids, one can solve problems that couple PDEs and time-dependent ODEs.

This example revisits the problem of the PDE with a flux boundary condition. The same problem will be solved in this chapter, but the domain will be divided into two halves, each represented by separate `Grid` objects, in order to demonstrate how one may couple the solutions on each domain.

3.11.1 Spatial domains

For convenience, the left and right spatial domains will be represented by objects belong to the same user-defined `Domain` class. This is desirable because these objects share many similarities. For more flexibility, one could define two different classes, perhaps both derived from an intermediate user-defined class.

The `Domain` class in this example is very similar to that of the earlier PDE example, but the associated spatial coordinate values are now provided through a parameter to `__init__`, as they will differ between the left and right spatial domains.

```
# Objects of this class describe the spatial domains on which problems will be
# solved.
class Domain(pygdh.Grid):

    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_grid' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, name, unit_count, x_coordinates):

        self.initialize_grid(name,
                             # The independent variable will be named 'y'
                             field_names=['y'],
                             # The dependent variable will be named 'x',
                             # and 'y' will be computed at the positions
                             # specified by the x_coordinates parameter
                             coordinate_values={'x': x_coordinates},
                             # Each unit in 'Domain' will be described by a
                             # 'Volume' object
                             unit_classes=[Volume for i in range(unit_count)],
                             stored_solution_count=2)
```

The `__init__` method of the `Problem`-derived class now creates two `Grid` objects. A list naming these objects is passed to `initialize_problem`. The order in which they are listed implicitly associates each with a numerical label; here, `'left_domain'` is associated with the index `0`, and `'right_domain'` is associated with the index `1`.

```
    def __init__(self, alpha_l, unit_count1, alpha_r, unit_count2):

        # Save the argument value as an object member
        self.alpha_l = alpha_l
        self.alpha_r = alpha_r

        # The left and right domains are both objects of the 'Domain' class,
        # but they must be initialized to represent different regions
        grid_obj1 = Domain('left_domain',unit_count1,
                           numpy.linspace(0., 0.5, unit_count1))
        grid_obj2 = Domain('right_domain',unit_count2,
                           numpy.linspace(0.5, 1., unit_count2))
```

(continues on next page)

(continued from previous page)

```
        # Prepare to run simulation using two Grid objects, numbered in this
        # order
        self.initialize_problem([grid_obj1,grid_obj2],stored_solution_count=2)
```

3.11.2 Discretized equations, part 1

Most of the `Volume` objects are associated with the same discretized equation used in the original example, although separate `Grid` objects may associated with separate thermal diffusivities, and one must be careful to reference the solution field values of the appropriate `Grid` by providing the appropriate index to the `grid` array:

```
    def left_grid_pde(self, vol, residuals):

        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]

        residuals[0] = ((vol.interpolate(y0) - vol.interpolate(y_1))*vol.volume
                        - self.timestep_size * self.alpha_l
                        * (vol.dx_right(y0) - vol.dx_left(y0)))
```

```
    def right_grid_pde(self, vol, residuals):

        # Present solution values
        y0 = self.grid[1].field[0][0]

        # Past solution values
        y_1 = self.grid[1].field[-1][0]

        residuals[0] = ((vol.interpolate(y0) - vol.interpolate(y_1))*vol.volume
                        - self.timestep_size * self.alpha_r
                        * (vol.dx_right(y0) - vol.dx_left(y0)))
```

3.11.3 Boundary conditions

Similarly, the equation at the right boundary of the right domain, at which there is a flux boundary condition, is the same as that in the original example, except with the appropriate value of the thermal diffusivity and using the index corresponding to the second `Grid`:

```
    def right_grid_right_boundary_pde(self, vol, residuals):

        # Present solution values
        y0 = self.grid[1].field[0][0]

        # Past solution values
        y_1 = self.grid[1].field[-1][0]

        residuals[0] = ((vol.interpolate(y0) - vol.interpolate(y_1))*vol.volume
```

(continues on next page)

(continued from previous page)

```
                        + self.timestep_size
                        * (-1.0 + self.alpha_r * vol.dx_left(y0)))
```

The value of the dependent variable at the left boundary of the left domain is given in the problem formulation, and so is treated in the same way as in the original example.

However, the interface between the two domains requires particular attention. The two domains are continuous, and as described in the earlier chapter on discretization, the dependent variable values associated with the `Volume` objects at the ends of each domain are "located on" the domain boundaries. The dependent variable value of the rightmost `Volume` of the left domain should therefore be identical to that on the leftmost `Volume` of the right domain. This value should be determined by the combined behavior within the "half-volumes" on either side of the interface.

The shared interface can be implemented in many ways. In this example, the governing equation will be solved on the rightmost `Volume` of the left domain (while still accounting for the behavior on both sides of the interface), and the value at the adjacent boundary of the right domain will be "given" by the solution on the left side of the interface. As usual, the boundary values will be set through `set_boundary_values`

```
    def set_boundary_values(self):

        ## Boundary with the prescribed value is on the left side of the left
        ## domain
        y_left = self.grid[0].field[0][0]

        # Left boundary
        y_left[0] = 0.

        ## In the right domain, the value at left boundary is taken from the
        ## adjacent left domain
        y_right = self.grid[1].field[0][0]

        # Left boundary
        y_right[0] = y_left[-1]
```

This arrangement is consistent with the `declare_unknowns` method that is common to both `Domain` objects, and which is identical to that in the original example

```
    # This method is required. It informs PyGDH of the unknown quantities for
    # for which it must solve.
    def declare_unknowns(self):

        for grid_number in range(2):
            # left boundary
            vol = self.grid[grid_number].unit_with_number[0]
            self.unknown[grid_number][0,vol.number] = False

            # Interior volumes
            for vol_number in range(1, self.grid[grid_number].unit_count-1):
                vol = self.grid[grid_number].unit_with_number[vol_number]
                self.unknown[grid_number][0,vol.number] = True

            # right boundary
            vol = self.grid[grid_number].unit_with_number[-1]
            self.unknown[grid_number][0,vol.number] = True
```

3.11.4 Discretized equations, part 2

As discussed in the previous section, as implemented, the "known" solution value on the rightmost `Volume` of the left `Grid` is copied from the computed value of the leftmost `Volume` of the right `Grid`. The `Volume` on the right `Grid` is associated with a discretized equation that has contributions from both sides of the interface.

In order to obtain this discretized equation, we retrace our steps in the earlier formulation on a single `Grid`, but recognizing this time that the parameter α may take different values α_l and α_r on the left and right sides of the interface. The left edge of the left `Volume` will be located at position a, the interface at b, and the right edge of the right `Volume` at c.

Integrating the governing equation over the entire region gives

$$\int_a^c \frac{\partial y}{\partial t}\,dx - \alpha_l \left[\frac{\partial y}{\partial x}\right]_a^b - \alpha_r \left[\frac{\partial y}{\partial x}\right]_b^c = 0.$$

Using the average value of the integrand, this is

$$\left\langle \frac{\partial y}{\partial t} \right\rangle (c-a) - \alpha_l \left[\frac{\partial y}{\partial x}\right]_a^b - \alpha_r \left[\frac{\partial y}{\partial x}\right]_b^c = 0.$$

As an approximation, the average value of the time derivative is now taken as the value of the time derivative at the interface:

$$\frac{\partial y_b}{\partial t}(c-a) - \alpha_l \left[\frac{\partial y}{\partial x}\right]_a^b - \alpha_r \left[\frac{\partial y}{\partial x}\right]_b^c \approx 0.$$

Discretization in time by the first-order backward difference scheme then gives

$$\frac{y\left(t_0+\Delta t, b\right) - y\left(t_0, b\right)}{\Delta t}(c-a) - \alpha_l \left[\frac{\partial y}{\partial x}\right]_a^b - \alpha_r \left[\frac{\partial y}{\partial x}\right]_b^c \approx 0.$$

In the physical interpretation of the heat equation, the product of α and the spatial derivative the negative of the flux, which is continuous in the present case. Therefore, the terms corresponding to positions immediately adjacent to the interface should cancel in principle, and we can conclude that

$$\frac{y\left(t_0+\Delta t, b\right) - y\left(t_0, b\right)}{\Delta t}(c-a) - \alpha_r \frac{\partial y}{\partial x}(t_0+\Delta t, c) + \alpha_l \frac{\partial y}{\partial x}(t_0+\Delta t, a) \approx 0.$$

Note that this suggests that the method implementing this equation must use information taken from both neighboring `Grid` objects. This is implemented as follows:

```
    def interface_pde(self, vol, residuals):

        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]

        right_y0 = self.grid[1].field[0][0]
        right_vol = self.grid[1].unit_with_number[0]

        residuals[0] = ((vol.interpolate(y0) - vol.interpolate(y_1))*(vol.volume +␣
↪right_vol.volume)
                        + self.timestep_size
                        * (-self.alpha_r*right_vol.dx_right(right_y0)
                           +self.alpha_l*vol.dx_left(y0)))
```

3.11.5 Assigning equations

The assignment of the governing equation methods to the `Volumes` of the two domains is straightforward, although one must be careful to use the appropriate numerical labels when referencing data structures

```
    def assign_equations(self):

        ## Left grid

        left_grid_equations = self.equations[0]

        # Left boundary
        left_grid_equations[0] = []

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            left_grid_equations[vol_number] = [self.left_grid_pde]

        # Right boundary
        left_grid_equations[-1] = [self.interface_pde]

        ## Right grid

        right_grid_equations = self.equations[1]

        # Left boundary, value to be copied from left grid
        right_grid_equations[0] = []

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            right_grid_equations[vol_number] = [self.right_grid_pde]

        # Right boundary
        right_grid_equations[-1] = [self.right_grid_right_boundary_pde]
```

3.11.6 Solution

Although the two thermal diffusivities can be specified independently, they will both be given the value used in the original example in order to check consistency with the original numerical solution:

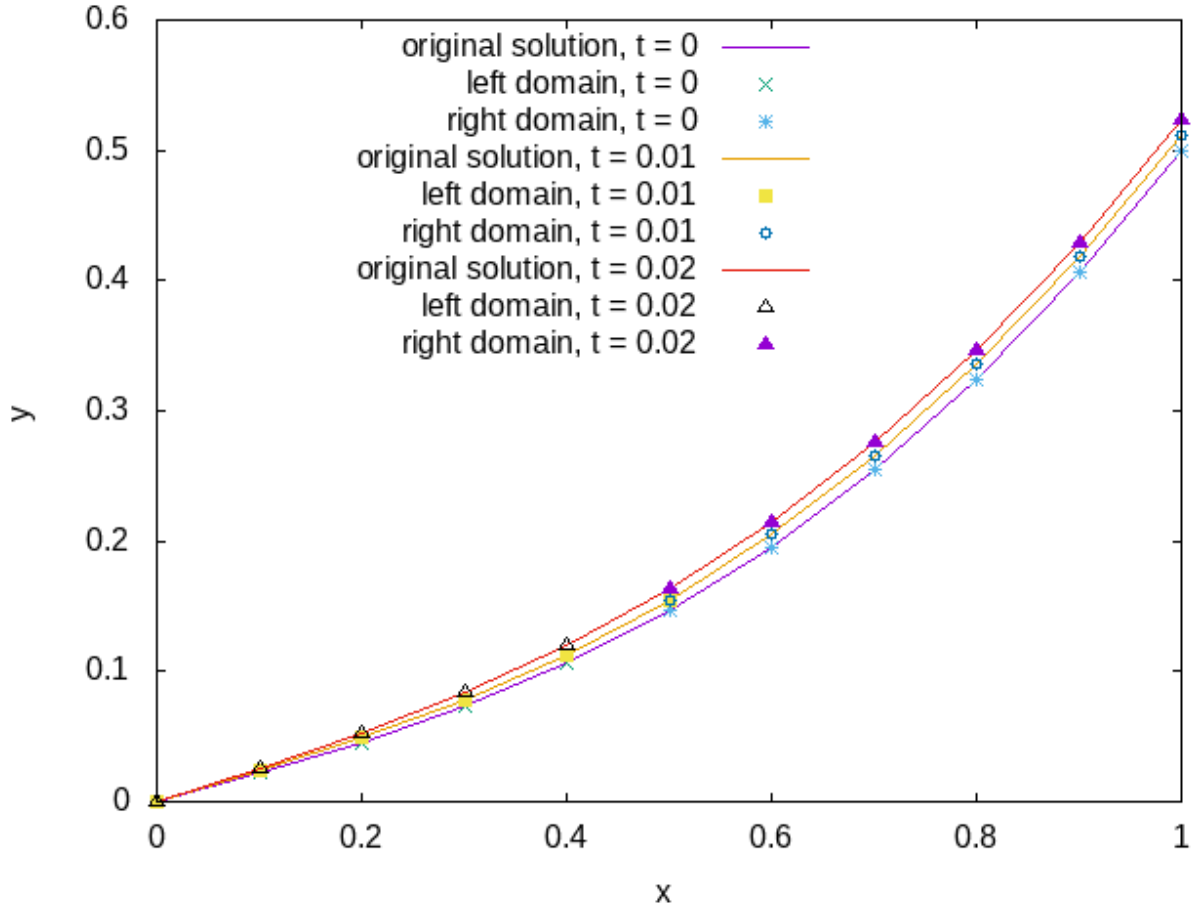
```
# Create an object instance, setting alpha_l = 1.0, alpha_r = 1.0
problem = MultipleGrids(1.0, 6, 1.0, 6)

number_of_timesteps = 3
timestep_size = 0.01

# The names of all output files will begin with this string
filename_root = 'multiple_grids'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

# Calculate solution and write output to file
problem.solve_time_dependent_system(filename_root, output_types,
```

(continues on next page)

(continued from previous page)

```
                                    timestep_count=number_of_timesteps,
                                    timestep_size=timestep_size)
```

The old and new solutions are plotted below at the first few timesteps:



This was generated with the following GNUPLOT script:

```
set term png
set output 'multiple_grids.png'
set key left top
set xlabel 'x'
set ylabel 'y'
plot '../PDE/PDE_domain.gnuplot' index 0 title 'original solution, t = 0' with lines,
↪'multiple_grids_left_domain.gnuplot' index 0  title 'left domain, t = 0' with␣
↪points, 'multiple_grids_right_domain.gnuplot' index 0  title 'right domain, t = 0'␣
↪with points, '../PDE/PDE_domain.gnuplot' index 1 title 'original solution, t = 0.01
↪' with lines, 'multiple_grids_left_domain.gnuplot' index 1  title 'left domain, t =␣
↪0.01' with points, 'multiple_grids_right_domain.gnuplot' index 1  title 'right␣
↪domain, t = 0.01' with points, '../PDE/PDE_domain.gnuplot' index 2 title 'original␣
↪solution, t = 0.02' with lines, 'multiple_grids_left_domain.gnuplot' index 2  title
↪'left domain, t = 0.02' with points, 'multiple_grids_right_domain.gnuplot' index 2 ␣
↪title 'right domain, t = 0.02' with points
set output
```

3.11.7 Exercises

1. Modify this program so that at the interface, the value on the left domain is copied from right domain. Due to the presence of `declare_unknowns` in the `Domain` definition, some large modifications are needed. Two suggested approaches are to:

a. pass an additional parameter to `__init__` to indicate which set of `unknown_field` values should be used for a given `Domain` object, or

b. remove `declare_unknowns` from `Domain` and create two new classes with `Domain` as a base class. Each of these classes will have a separate definition of `declare_unknowns`. The left and right spatial domains will now be associated with objects created from these two different classes (which share the same base class).

3.12 Coordinating the solution of multiple subproblems

When solving problems that might involve multiple equations, spatial domains, or timescales, one may choose to solve subproblems in a staggered fashion in order to reduce computational cost or improve code reusability. However, doing so will typically come at the cost of reduced overall accuracy, as solutions to equations that simultaneously hold in the mathematical sense are no longer solved simultaneously in the numerical sense.

`Coordinator` objects can be used to coordinate the staggered solution of different subproblems. We will return to the earlier example of the problem solved across coupled domains, and divide it into two subproblems that correspond to the two domains.

3.12.1 A common base class

The numerical problem to be solved takes the same form within all volumes in the interiors of the two domains, as they are not directly influenced by values at the domain boundaries. It description is identical to that used in solving the earlier problem in which solutions in the two regions are obtained simultaneously. We take advantage of this similarity and the object-oriented capabilities of the Python language by defining a base class that describes the common components of the problems:

```
# Objects of this class bring the spatial domain definitions together with
# equations, boundary conditions, and for time-dependent problems, initial
# conditions. These objects can then be directed to compute numerical
# solutions.
class BaseSubproblem(pygdh.Problem):

    def __init__(self, alpha, grid_obj):

        # Save the argument value as an object member
        self.alpha = alpha

        self.initialize_problem([grid_obj],stored_solution_count=2)

#### SPHINX SNIPPET START BOUNDARY SOURCE
    def add_boundary_source(self,boundary_source):

        self.boundary_source = boundary_source
#### SPHINX SNIPPET END BOUNDARY SOURCE

    # This sets the initial values of the independent variables on the region
    # represented by an object of this class.
```

(continues on next page)

(continued from previous page)

```
    def set_initial_conditions(self):

        # Defined for clarity
        y = self.grid[0].field[0][0]

        # Set the value of the independent variable on each volume
        for vol in self.grid[0].unit_with_number:
            y[vol.number] = (
                vol.coordinate / self.alpha
                - 0.5*math.sin(0.5*math.pi*vol.coordinate))

    def bulk_pde(self, vol, residuals):

        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]

        residuals[0] = ((vol.interpolate(y0) - vol.interpolate(y_1))*vol.volume
                        - self.timestep_size * self.alpha
                        * (vol.dx_right(y0) - vol.dx_left(y0)))
```

We note that the `Grid` objects are created separately and passed as parameters to `__init__` rather than being created within `__init__` as in the earlier implementation. This is done because the `Grid` objects are also passed to the `Coordinator` object in order to generate corresponding output files.

Each subproblem is then represented by a derived class that describes components that are specific to each subproblem. The discretized equations that correspond to the left and right ends of the combined domain are the same as those used in the earlier problem in which the two regions are solved simultaneously.

3.12.2 The interface

For direction on how to describe the interface between the two regions, we return to the development of the discretized equation that represents the boundary in the earlier coupled domain problem. It was determined that

$$\frac{y\left(t_0+\Delta t,b\right)-y\left(t_0,b\right)}{\Delta t}(c-a)-\alpha_l\left[\frac{\partial y}{\partial x}\right]_a^b-\alpha_r\left[\frac{\partial y}{\partial x}\right]_b^c\approx 0.$$

But for a region bounded by a and b,

$$\frac{y\left(t_0+\Delta t,b\right)-y\left(t_0,b\right)}{\Delta t}(b-a)-\alpha_l\left(\frac{\partial y}{\partial x}(t_0+\Delta t,b)-\frac{\partial y}{\partial x}(t_0+\Delta t,a)\right)\approx 0.$$

We would like to solve the last equation numerically. One can rewrite the first equation as

$$\frac{y\left(t_0+\Delta t,b\right)-y\left(t_0,b\right)}{\Delta t}(b-a)-\alpha_l\left[\frac{\partial y}{\partial x}\right]_a^b\frac{b-a}{c-a}-\alpha_r\left[\frac{\partial y}{\partial x}\right]_b^c\frac{b-a}{c-a}\approx 0.$$

Subtracting this from the second equation gives

$$-\alpha_l\left(\frac{\partial y}{\partial x}(t_0+\Delta t,b)-\frac{\partial y}{\partial x}(t_0+\Delta t,a)\right)+\alpha_l\left[\frac{\partial y}{\partial x}\right]_a^b\frac{b-a}{c-a}+\alpha_r\left[\frac{\partial y}{\partial x}\right]_b^c\frac{b-a}{c-a}\approx 0.$$

$$-\alpha_l\left[\frac{\partial y}{\partial x}\right]_a^b\frac{c-b}{c-a}+\alpha_r\left[\frac{\partial y}{\partial x}\right]_b^c\frac{b-a}{c-a}\approx 0.$$

$$-\frac{\alpha_l}{b-a}\left[\frac{\partial y}{\partial x}\right]_a^b \approx -\frac{\alpha_r}{c-b}\left[\frac{\partial y}{\partial x}\right]_b^c.$$

This suggests that to describe the region bounded by a and b, as a separate subproblem, one could solve

$$\frac{y\left(t_0+\Delta t, b\right)-y\left(t_0, b\right)}{\Delta t}(b-a)+\left\{-(b-a)\frac{\alpha_r}{c-b}\left[\frac{\partial y}{\partial x}\right]_b^c\right\} \approx 0,$$

or

$$\frac{y\left(t_0+\Delta t, b\right)-y\left(t_0, b\right)}{\Delta t}+\left\{-\frac{\alpha_r}{c-b}\left[\frac{\partial y}{\partial x}\right]_b^c\right\} \approx 0,$$

with the value of the term in curly brackets simply provided as a "boundary condition." Ideally, this would be taken from the domain on the right where a submodel solution is simultaneously obtained. However, for a staggered solution, we must settle for solutions that are not simultaneously obtained, but more likely obtained in the previous iteration.

In order to promote encapsulation and avoiding the need for the two subproblems to know the internal structure of each other, we will need a mechanism for passing this additional information to each subproblem. This is provided in part by the `add_boundary_source` methods that are used to pass methods describing the boundary terms in each subproblem to the other subproblem:

```
    def add_boundary_source(self,boundary_source):

        self.boundary_source = boundary_source
```

Unlike many of the methods encountered earlier, the `add_boundary_source` method is completely user-defined; its definition does not represent the override of methods defined by PyGDH. The ability to create arbitrary methods that can play important roles in the numerical implementations is another example of the benefit of implementing PyGDH as a Python package that leaves the Python interpreter completely exposed to users.

Following the above derivation, the derived classes that describe the full subproblems define equation methods that describe behavior adjacent to the interface and which incorporate boundary information from the other subproblem. Each subproblem also contains a method that makes this boundary information available to the other subproblem.

```
# Objects of this class bring the spatial domain definitions together with
# equations, boundary conditions, and for time-dependent problems, initial
# conditions. These objects can then be directed to compute numerical
# solutions.
class LeftSubproblem(BaseSubproblem):

#### SPHINX SNIPPET START DECLARE UNKNOWNS
    # This method is required. It informs PyGDH of the unknown quantities for
    # for which it must solve.
    def declare_unknowns(self):

        # left boundary
        vol = self.grid[0].unit_with_number[0]
        self.unknown[0][0,vol.number] = False

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            vol = self.grid[0].unit_with_number[vol_number]
            self.unknown[0][0,vol.number] = True

        # right boundary
        vol = self.grid[0].unit_with_number[-1]
```

(continues on next page)

(continued from previous page)

```
        self.unknown[0][0,vol.number] = True
#### SPHINX SNIPPET END DECLARE UNKNOWNS

    def set_equation_scalar_counts(self):

        # This is a dictionary associating governing equation methods with
        # the number of scalar values that they return
        self.equation_scalar_counts = {self.bulk_pde: 1,
                                       self.right_boundary_pde: 1}

    def right_boundary_pde(self, vol, residuals):

        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]

        residuals[0] = ((vol.interpolate(y0) - vol.interpolate(y_1))
                        + self.timestep_size * self.boundary_source())

    def right_boundary_term(self):
        vol = self.grid[0].unit_with_number[-1]
        y = self.grid[0].field[0][0]
        return -self.alpha/vol.volume*(vol.dx_right(y) - vol.dx_left(y))

#### SPHINX SNIPPET START ASSIGN EQUATIONS
    def assign_equations(self):

        grid_equations = self.equations[0]

        # Left boundary
        grid_equations[0] = []

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            grid_equations[vol_number] = [self.bulk_pde]

        # Right boundary
        grid_equations[-1] = [self.right_boundary_pde]
#### SPHINX SNIPPET END ASSIGN EQUATIONS

#### SPHINX SNIPPET START BC
    def set_boundary_values(self):

        ## Boundary with the prescribed value is on the left side of the left
        ## domain
        y_left = self.grid[0].field[0][0]

        # Left boundary
        y_left[0] = 0.
#### SPHINX SNIPPET END BC

# Objects of this class bring the spatial domain definitions together with
# equations, boundary conditions, and for time-dependent problems, initial
# conditions. These objects can then be directed to compute numerical
# solutions.
```

(continues on next page)

(continued from previous page)

```
class RightSubproblem(BaseSubproblem):

    # This method is required. It informs PyGDH of the unknown quantities for
    # for which it must solve.
    def declare_unknowns(self):

        # left boundary
        vol = self.grid[0].unit_with_number[0]
        self.unknown[0][0,vol.number] = True

        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            vol = self.grid[0].unit_with_number[vol_number]
            self.unknown[0][0,vol.number] = True

        # right boundary
        vol = self.grid[0].unit_with_number[-1]
        self.unknown[0][0,vol.number] = True

    def left_boundary_pde(self, vol, residuals):

        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]

        residuals[0] = ((vol.interpolate(y0) - vol.interpolate(y_1))
                        + self.timestep_size * self.boundary_source())

    def left_boundary_term(self):
        vol = self.grid[0].unit_with_number[0]
        y = self.grid[0].field[0][0]

        return -self.alpha/vol.volume*(vol.dx_right(y) - vol.dx_left(y))

#### SPHINX SNIPPET START RIGHT BOUNDARY PDE
    def right_boundary_pde(self, vol, residuals):

        # Present solution values
        y0 = self.grid[0].field[0][0]

        # Past solution values
        y_1 = self.grid[0].field[-1][0]

        residuals[0] = ((vol.interpolate(y0) - vol.interpolate(y_1))*vol.volume
                        + self.timestep_size
                        * (-1.0 + self.alpha * vol.dx_left(y0)))
#### SPHINX SNIPPET END RIGHT BOUNDARY PDE

    def assign_equations(self):

        grid_equations = self.equations[0]

        # Left boundary
        grid_equations[0] = [self.left_boundary_pde]
```

(continues on next page)

(continued from previous page)

```
        # Interior volumes
        for vol_number in range(1, self.grid[0].unit_count-1):
            grid_equations[vol_number] = [self.bulk_pde]

        # Right boundary
        grid_equations[-1] = [self.right_boundary_pde]

    def set_equation_scalar_counts(self):

        # This is a dictionary associating governing equation methods with
        # the number of scalar values that they return
        self.equation_scalar_counts = {self.left_boundary_pde: 1,
                                       self.bulk_pde: 1,
                                       self.right_boundary_pde: 1}
```

By passing only these methods between the subproblems, neither subproblem requires detailed information about the internal operation of the other subproblem, supporting the object-oriented strategy of encapsulation.

Note that the declaration of locations of unknown values and equation assignments indicate that in both subproblems, the values at the interface are to be solved. Since the solver steps are staggered, these values are not obtained simultaneously and are likely to be slightly inconsistent.

3.12.3 Differencing formulas

Unlike in the earlier implementation, each subproblem must operate without detailed knowledge of the other, and in particular, the derivatives at the interface must be determined using only information available within a single subdomain. However, the low-order derivative operators in volumes adjacent to boundaries might not vary from one end of a volume to the other due to presence of the interface. This can produce inaccurate behavior. In order to address this problem, higher-order derivative operators are generated at the volume edges away from the interface:

```
class LeftBoundaryVolume(pygdh.Volume):

    # Defining this method gives users an opportunity to define their own
    # mathematical operators, for later use in describing governing equations.
    def define_interpolants(self):

        # Operator for estimating first derivative at left boundary of volume
        #self.dx_left = self.default_interpolant(-1.0, 1, 1)
        self.dx_left = self.default_interpolant(-1.0, 1, 2)

        # Operator for estimating function value at the volume center
        self.interpolate = self.default_interpolant(0.0, 0, 2)

        # Operator for estimating first derivative at right boundary of volume
        self.dx_right = self.default_interpolant(1.0, 1, 1)

# Objects of this class are the units from which the spatial domain will be
# constructed.
class Volume(pygdh.Volume):

    # Defining this method gives users an opportunity to define their own
    # mathematical operators, for later use in describing governing equations.
    def define_interpolants(self):
```

(continues on next page)

(continued from previous page)

```
        # Operator for estimating first derivative at left boundary of volume
        self.dx_left = self.default_interpolant(-1.0, 1, 1)

        # Operator for estimating function value at the volume center
        self.interpolate = self.default_interpolant(0.0, 0, 2)

        # Operator for estimating first derivative at right boundary of volume
        self.dx_right = self.default_interpolant(1.0, 1, 1)

class RightBoundaryVolume(pygdh.Volume):

    # Defining this method gives users an opportunity to define their own
    # mathematical operators, for later use in describing governing equations.
    def define_interpolants(self):

        # Operator for estimating first derivative at left boundary of volume
        self.dx_left = self.default_interpolant(-1.0, 1, 1)

        # Operator for estimating function value at the volume center
        self.interpolate = self.default_interpolant(0.0, 0, 2)

        # Operator for estimating first derivative at right boundary of volume
        #self.dx_right = self.default_interpolant(1.0, 1, 1)
        self.dx_right = self.default_interpolant(1.0, 1, 2)
```

The derivative operators used throughout most of this Guide are sufficient for many applications, but PyGDH provides access to these details so that users have additional flexibility when needed, as in the present situation.

These specialized boundary `Volume` objects are incorporated into the `Grid` objects corresponding to the present subdomains:

```

# Objects of this class describe the spatial domains on which problems will be
# solved.
class Domain(pygdh.Grid):

    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_grid' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, name, unit_count, x_coordinates):

        unit_classes = [ LeftBoundaryVolume ]

        for i in range(1,unit_count-1):
            unit_classes.append(Volume)

        unit_classes.append(RightBoundaryVolume)

        self.initialize_grid(name,
                             # The independent variable will be named 'y'
                             field_names=['y'],
                             # The dependent variable will be named 'x',
                             # and 'y' will be computed at the positions
                             # specified by the x_coordinates parameter
                             coordinate_values={'x': x_coordinates},
                             # Each unit in 'Domain' will be described by a
```

(continues on next page)

(continued from previous page)

```
                                    # 'Volume' object
                                    unit_classes=unit_classes,
                                    stored_solution_count=2)
```

Finally, the `Grid` objects are created independently and passed to the separately created `Problem` objects, boundary information methods are exchanged, and a `Coordinator` object is created and informed of the `Grid` objects for which it must product corresponding output, as well as the `Problem` objects that are two be looped over in the specified order. The `Coordinator` object is then directed to run through the full problem by making alternating steps with the `Problem` objects.

```
unit_count = 6
left_domain = Domain('left_domain',unit_count,numpy.linspace(0.0,0.5,unit_count))
right_domain = Domain('right_domain',unit_count,numpy.linspace(0.5,1.0,unit_count))

# Create an object instance, setting alpha in both regions to 1.0
alpha = 1.0
left_subproblem = LeftSubproblem(alpha,left_domain)
right_subproblem = RightSubproblem(alpha,right_domain)

left_subproblem.add_boundary_source(right_subproblem.left_boundary_term)
right_subproblem.add_boundary_source(left_subproblem.right_boundary_term)

coordinator = pygdh.Coordinator()
# Output will be automatically generated for Grids declared to Coordinator here
coordinator.initialize_coordinator([left_domain,right_domain],[right_subproblem,left_
↪subproblem],stored_solution_count=2)

number_of_timesteps = 3
timestep_size = 0.01

# The names of all output files will begin with this string
filename_root = 'coordinated'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

# Calculate solution and write output to file
coordinator.solve_time_dependent_system(filename_root, output_types,
                                        timestep_count=number_of_timesteps,
                                        timestep_size=timestep_size,handle_exception_
↪in_solver=False)
```

The output, generated from

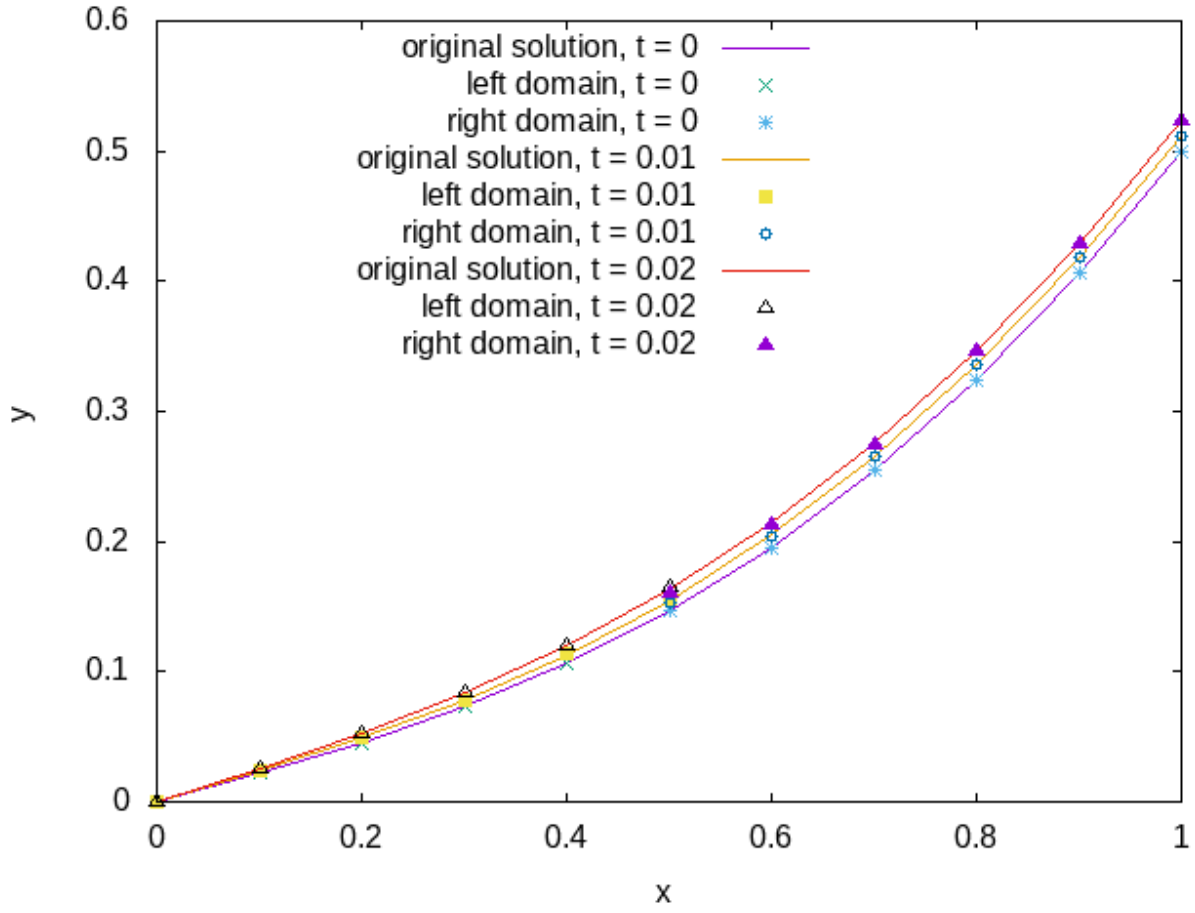
```
set term png
set output 'coordinated.png'
set key left top
set xlabel 'x'
set ylabel 'y'
plot '../PDE/PDE_domain.gnuplot' index 0 title 'original solution, t = 0' with lines,
↪'coordinated_left_domain.gnuplot' index 0  title 'left domain, t = 0' with points,
↪'coordinated_right_domain.gnuplot' index 0  title 'right domain, t = 0' with points,
↪ '../PDE/PDE_domain.gnuplot' index 1 title 'original solution, t = 0.01' with lines,
↪ 'coordinated_left_domain.gnuplot' index 1  title 'left domain, t = 0.01' with␣
↪points, 'coordinated_right_domain.gnuplot' index 1  title 'right domain, t = 0.01'␣
↪with points, '../PDE/PDE_domain.gnuplot' index 2 title 'original solution, t = 0.02
↪' with lines, 'coordinated_left_domain.gnuplot' index 2  title 'left domain, t = 0.
↪02' with points, 'coordinated_right_domain.gnuplot' index 2  title 'right domain, t␣
↪= 0.02' with points
```

(continues on next page)

(continued from previous page)

```
set output
```

shows results that are close to those produced by the earlier implementation in which the subdomain calculations are performed simultaneously. The slight mismatch near the interface is a reflection of the error incurred by obtaining alternating subproblem solutions that are slightly inconsistent with each other.



3.13 Two-dimensional spatial domains

Programs using PyGDH to solve problems with two-dimensional spatial domains have a similar structure to those used to solve one-dimensional problems, but a few extensions are necessary.

Please note that in comparison to the one-dimensional case, even seemingly small problem domains in higher dimensions may require fairly large amounts of computing time. This is one reason that support for three-dimensional spatial domains has not yet been implemented.

3.13.1 Initialization

The number of domain dimensions is inferred from the parameters passed to `initialize_grid`. As in the one-dimensional case, the `'coordinate_values'` parameter is itself a dictionary with one sequence of values for each spatial coordinate. These values give the locations of grid lines along the specified coordinate. As this suggests,

these coordinate value sequences define a grid of rectangles (potentially of varying sizes), formed by the intersection of grid lines, along which the value of only one coordinate changes.

The sequences of coordinate values must be specified in a strictly monotonically increasing order.

The `'coordinate_order'` parameter is a list of the "keys" from the `'coordinate_values'` dictionary, indicating the order in which numerical coordinates are to be understood (this must be specified explicitly, because the ordering of entries in a dictionary is not guaranteed to be the order in which they are created). For example, the parameters appearing in:

```
    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_grid' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, name, unit_count):

        self.initialize_grid(name,
                             coordinate_values = {
                                 'x': numpy.linspace(0., 1., unit_count),
                                 'y': numpy.linspace(0., 1., unit_count)},
                             coordinate_order=['x','y'],
                             field_names=['T'],
                             # Each unit in 'Domain' will be described by a
                             # 'Volume' object
                             unit_classes=[ [ Volume
                                              for j in range(unit_count) ]
                                            for i in range(unit_count)])
```

indicate that the first number in a pair of coordinates corresponds to a coordinate named `'x'`, and that the second number corresponds to a coordinate named `'y'`. Note that the same names do not have to be used for corresponding Python variables, but inconsistency is likely to be confusing.

The `'coordinate_order'` list also serves to associate numerical labels with the different coordinate directions. These numerical labels are used as indices in arrays for which a numerical index is used to indicate direction, such as when describing interpolants.

3.13.2 Grid layout overview

PyGDH is only capable of solving problems on spatial domains that can be mapped to rectangular grids. These grids may have irregularly-spaced grid lines, and regions may be excluded from the grid when defining the spatial domain if the remaining volumes have boundaries that coincide with grid lines.

While the `Volume` objects in two-dimensional domains are assigned integer labels, these labels are not necessarily related in a simple way to positions within the domain, as is the case in one dimension. Many of the underlying data structures, such as the solution value arrays in `fields`, use these integer labels, but it is never necessary for the user to know the actual label values.

In two dimensions, the sequences given in the `'coordinate_values'` entry in a grid description dictionary give the coordinates of the grid lines–on each grid line, the value of the respective coordinate is fixed. Each intersection of grid lines is associated with the position of a `Volume`. In other words, taking the first coordinate from the `'coordinate_values'` entry associated with the first element in the `'coordinate_order'` list, and taking the second coordinate from the entry associated with the second element in the `'coordinate_order'` list, gives a coordinate pair that is associated with a `Volume`. This coordinate pair is stored as the `coordinate` member of the same `Volume`.

The integer indices into the `'coordinate_values'` sequences which correspond to the `coordinate` values of a `Volume` object are, when taken in order, the "grid indices" of the `Volume`. These are stored as the `grid_indices`

member of the same `Volume`, which is a useful label for referencing `Volume` objects.

3.13.3 Defining domain boundaries

In the one-dimensional case, there were only `Volume` elements associated with domain boundaries, one at each end. In the two-dimensional case, a rectangular domain has four boundaries, and excluding portions of a grid creates additional boundaries.

Since assignments of values and equations are commonly performed on collections of `Volume` objects, PyGDH requires, for two-dimensional problems, that users define collections of these objects in order to simplify the rest of the problem description. This is done by defining a method named `define_unit_set_with_name` in a class derived from `pygdh.Grid`. This method takes no arguments besides `self`, and must define a dictionary for each `Grid` in `self.grid` that contains more than one `Volume`. Each "key-value" entry in this dictionary should have a descriptive name (in the form of a string) as a "key" for the collection of `Volume` objects represented by the "value".

These collections of `Volume` objects must be defined as Python set objects, which correspond to mathematical sets. PyGDH assists the user in constructing the required set objects by specifying locations of their `Volume` objects.

In the one-dimensional case, relative positions of `Volume` objects within the domain were directly related to the integer labels, so identifying the `Volume` objects by label was a simple matter. In two dimensions, `Volume` objects are most easily referenced by their locations, expressed as grid indices. The `boundary_unit_set_grown_from_edge_index` method of the `Grid` objects returns a set of `Volume` objects on the same boundary edge, given the grid indices of one of the `Volume` objects on that boundary.

The following method definition groups the boundaries of the rectangular domain into sets with descriptive, user-defined names:

3.13.4 Declaring unknowns and assigning equations

The process of notifying `Volume` objects about their associated unknown solution values is essentially the same as in the one-dimensional problems, but here, the user is encouraged to use the set definitions from the previous section:

```
    # This method is required. It informs PyGDH of the unknown quantities for
    # for which it must solve.
    def declare_unknowns(self):

        # Indicate that equation is to be solved on all volumes, then overwrite
        # for volumes on boundaries.  This is inefficient, but only done once,
        # so the cost is negligible.
        for vol in self.grid[0].unit_with_number:
            self.unknown[0][0,vol.number] = True

        ## Solution values are prescribed on all boundaries

        for vol in self.grid[0].unit_set_with_name['left']:
            self.unknown[0][0,vol.number] = False

        for vol in self.grid[0].unit_set_with_name['bottom']:
            self.unknown[0][0,vol.number] = False

        for vol in self.grid[0].unit_set_with_name['top']:
            self.unknown[0][0,vol.number] = False

        for vol in self.grid[0].unit_set_with_name['right']:
            self.unknown[0][0,vol.number] = False
```

Note that the `for` statements iterate directly over the `Volume` objects contained in the set objects. It is not necessary for the user to know how the set objects store the `Volume` objects, only that the `for` loops are guaranteed to visit all of the stored objects. .. The `volumes` member of the `Grid` objects stored in `self.grid` is a `list` containing all `Volume` objects in the domain.

Similarly, the set definitions should be used when assigning equations to `Volume` objects.

3.13.5 Discretization in two dimensions

In Cartesian coordinates, the Laplace equation in two dimensions is written as:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

Performing a double integral over the two-dimensional "volume" (an area) within the boundaries of a `Volume` gives, for this "well-behaved" equation:

$$\begin{aligned}
\int_{x_0}^{x_1} \int_{y_0}^{y_1} \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) dy\, dx &= 0 \\
\int_{y_0}^{y_1} \int_{x_0}^{x_1} \frac{\partial^2 T}{\partial x^2}\, dx\, dy + \int_{x_0}^{x_1} \int_{y_0}^{y_1} \frac{\partial^2 T}{\partial y^2}\, dy\, dx &= 0 \\
\int_{y_0}^{y_1} \left[\frac{\partial T}{\partial x} \right]_{x_0}^{x_1} dy + \int_{x_0}^{x_1} \left[\frac{\partial T}{\partial y} \right]_{y_0}^{y_1} dx &= 0 \\
\left[\frac{\partial T}{\partial x} \left(x, \frac{y_1 + y_0}{2} \right) \right]_{x_0}^{x_1} (y_1 - y_0) + \left[\frac{\partial T}{\partial y} \left(\frac{x_1 + x_0}{2}, y \right) \right]_{y_0}^{y_1} (x_1 - x_0) &\approx 0
\end{aligned}$$

In the notation of PyGDH, this can be written as:

```
def pde(self, vol, residual):

    # Defined for clarity
    T = self.grid[0].field[0][0]

    residual[0] = ((vol.dx_right(T) - vol.dx_left(T))*vol.length[1]
                   + (vol.dy_up(T) - vol.dy_down(T))*vol.length[0])
```

The `length` members of the `Volume` elements are sequences with as many elements as there are spatial dimensions in the domain. Each element gives the difference in the associated coordinate value from one side of the `Volume` to the other. The `'coordinate_order'` in the grid definition determines which integer index value corresponds to which coordinate variable.

3.13.6 Boundary values

For the present example, we will use the following boundary conditions:

$$\begin{aligned}
T(0, y) &= 0 \\
T(x, 0) &= 0 \\
T(1, y) &= \sin(\pi y) \\
T(x, 1) &= 0
\end{aligned}$$

The boundary values are defined by the same mechanism as before, but the user is again encouraged to make use of the set definitions:

```
    def set_boundary_values(self):

        # Defined for clarity
        T = self.grid[0].field[0][0]

        ## Prescribed value of zero on all but the 'right' boundary

        for vol in self.grid[0].unit_set_with_name['left']:
            T[vol.number] = 0

        for vol in self.grid[0].unit_set_with_name['bottom']:
            T[vol.number] = 0

        for vol in self.grid[0].unit_set_with_name['top']:
            T[vol.number] = 0

        # This boundary condition is consistent with the other conditions at the
        # corners, as it should be.
        for vol in self.grid[0].unit_set_with_name['right']:
            T[vol.number] = math.sin(math.pi*vol.coordinate[1])
```

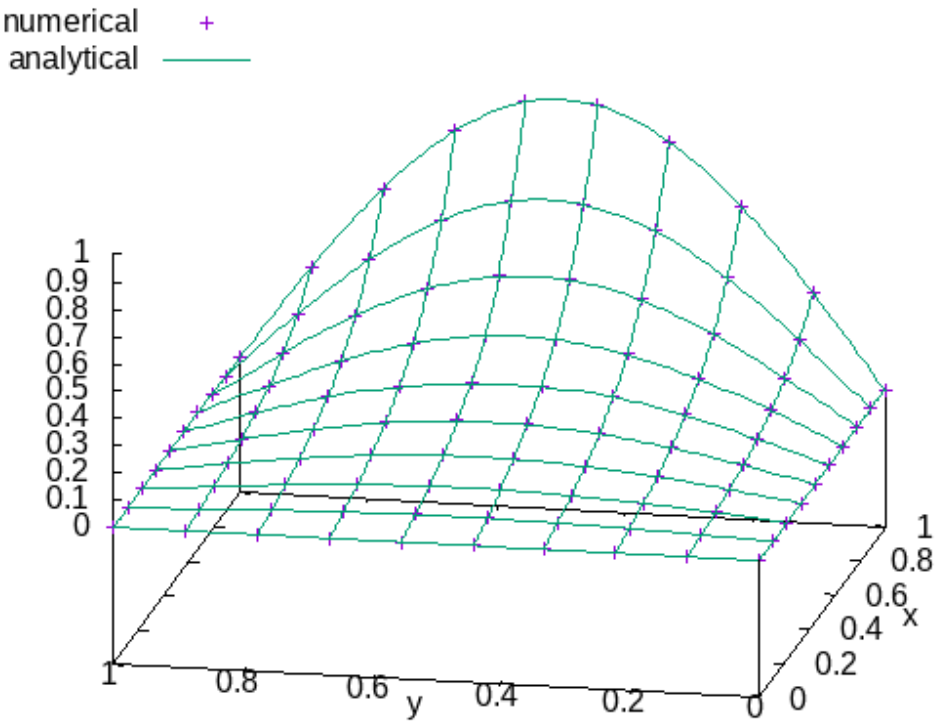
3.13.7 Solution

Just as in the one-dimensional time-independent case, an instance of the user's class is created, and directed to obtain a solution:

```
problem = Laplace_Equation(10)

# The names of all output files will begin with this string
filename_root = 'two_dimensions'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

problem.solve_time_independent_system(filename_root, output_types)
```

The results are plotted below against the analytical solution,

$$T(x, y) = \frac{\sinh(\pi x)}{\sinh \pi} \sin(\pi y)$$



This output was produced with the following GNUPLOT script:

```
set term png
set output 'two_dimensions.png'
set key left top
set view 67,281
set xlabel 'x'
set ylabel 'y'
splot 'two_dimensions_domain.gnuplot' title 'numerical', sin(pi*y)*sinh(pi*x)/
↪sinh(pi) title 'analytical'
set output
```

3.13.8 Suggested exercises

1. Solve Poisson's equation

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = -4\pi\rho$$

for $\rho = 1$ and boundary conditions

$$\begin{aligned}\phi(0, y) &= 0 \\ \phi(x, 0) &= -2\pi x^2 + 1 \\ \phi(1, y) &= \sin(\pi y) - 2\pi \\ \phi(x, 1) &= -2\pi x^2 + 1\end{aligned}$$

and compare the numerical result with the analytical solution.

3.14 A more complicated two-dimensional problem domain

PyGDH can solve equations on spatial domains with more complicated shapes. This chapter will make use of much of the material in the previous chapter, with small modifications.

3.14.1 Defining the domain shape

PyGDH is limited to solving equations on domains bounded by grid lines along which only one spatial coordinate varies. While the `'coordinate_values'` parameter to `initialize_grid` defines the maximum size of the domain, parts of the grid can be excluded from the problem domain. These excluded parts must also have boundaries along which only one spatial coordinate varies.

As in the previous chapter, each coordinate n-tuple (obtained by taking values from the `'coordinate_values'` entries in the order given in the `'coordinate_order'` entries) is associated with a region of the grid. In the interior of the grid, the boundaries between regions are located at coordinate values midway between the provided coordinate values. The first and last values of any coordinate sequence are interpreted as boundaries of the regions at the edge of the grid. Each region can also be referenced by its "grid indices", the n-tuple of indices into the `'coordinate_values'` sequences that give the coordinates associated with the region.

In the previous chapter, all of these regions are associated with `Volume` object. However, regions may be excluded from the problem domain, so that there is no associated `Volume` object. No equations are solved in these regions, and no degrees of freedom are associated with these regions. `Volume` objects adjacent to these excluded regions are then associated with the domain boundaries, for which boundary conditions must be supplied.

Regions may be excluded from the problem domain by simply entering `None` in place of a `Volume`-derived class in the `unit_classes` parameter to `initialize_grid`. In this example, this is done with nested `for` loops:

```
    # This method is automatically called immediately after an object of this
    # class is created. Users are free to modify this method definition as
    # desired, as long as the 'self.initialize_grid' method is called with
    # appropriate parameter values at some point in this method.
    def __init__(self, name, unit_count):

        # Each unit in 'Domain' will be described by a 'Volume' object
        unit_layout = [ [ Volume for j in range(unit_count) ]
                        for i in range(unit_count) ]

        # Put a hole into the domain
        for i in range(3,6):
            for j in range(3,6):
                unit_layout[i][j] = None

        self.initialize_grid(name,
```

(continues on next page)

(continued from previous page)

```
                                coordinate_values = {
                                    'x': numpy.linspace(0., 1., unit_count),
                                    'y': numpy.linspace(0., 1., unit_count)},
                                coordinate_order=['x','y'],
                                field_names=['T'],
                                unit_classes=unit_layout)
```

This excludes the regions associated with nine intersections formed by three grid lines in both directions, creating a "hole" in the interior of the grid. Actually, a slightly larger area is excluded from the problem domain, because the regions associated with the adjacent, newly-defined boundary `Volume` objects must be redefined. The `coordinate` member of a `Volume` object associated with a boundary is interpreted as a location on that boundary, while is interpreted as the center of a `Volume` object in the interior.

There are restrictions on excluding regions. Every `Volume` must have a neighboring `Volume` in each direction. This is a requirement in order to use even the linear interpolation routines, and under the interpretation of the `coordinate` of a `Volume` indicating a location on a boundary for boundary `Volume` objects, a `Volume` with no neighbors in one direction occupies no space. Also, the problem domain must be contiguous.

Please note that as a consequence of this approach to defining domain shapes, the boundaries formed by excluding regions have positions that are determined by the `'coordinate_values'` entries used to initialize `Grid` objects.

3.14.2 Defining domain boundaries

The boundaries defined in the previous chapter will be retained, and the newly-formed boundaries on the interior of the grid must be defined as well.

```
    def define_unit_set_with_name(self):

        # Descriptive names for boundaries of rectangular domain
        self.unit_set_with_name = {
            'left': self.boundary_unit_set_grown_from_edge_index((0, 1)),
            'bottom': self.boundary_unit_set_grown_from_edge_index((1, 0)),
            'top': self.boundary_unit_set_grown_from_edge_index((1, -1)),
            'right': self.boundary_unit_set_grown_from_edge_index((-1, 1)),
            'inner_left': self.boundary_unit_set_grown_from_edge_index((2,3)),
            'inner_bottom': self.boundary_unit_set_grown_from_edge_index((3,2)),
            'inner_top': self.boundary_unit_set_grown_from_edge_index((3,6)),
            'inner_right': self.boundary_unit_set_grown_from_edge_index((6,3))}

        boundaries = set()
        for unit_set in self.unit_set_with_name.values():
            boundaries = boundaries.union(unit_set)
        self.unit_set_with_name['boundaries'] = boundaries
        self.unit_set_with_name['all'] = set(self.unit_with_number)
        self.unit_set_with_name['interior'] = self.unit_set_with_name['all'].
↪difference(self.unit_set_with_name['boundaries'])
```

3.14.3 Declaring unknowns and assigning equations

The solution values will be prescribed on the "new" boundaries as well as on the "old" ones:

```
    # This method is required. It informs PyGDH of the unknown quantities for
    # for which it must solve.
    def declare_unknowns(self):

        # Indicate that equation is to be solved on all volumes, then overwrite
        # for volumes on boundaries.  This is inefficient, but only done once,
        # so the cost is negligible.
        for vol in self.grid[0].unit_with_number:
            self.unknown[0][0,vol.number] = True

        ## Solution values are prescribed on all boundaries

        for vol in self.grid[0].unit_set_with_name['boundaries']:
            self.unknown[0][0,vol.number] = False
        for vol in self.grid[0].unit_set_with_name['interior']:
            self.unknown[0][0,vol.number] = True

    def assign_equations(self):

        domain_equations = self.equations[0]

        # Solve the PDE for all interior ``Volume`` objects
        for vol in self.grid[0].unit_set_with_name['interior']:
            domain_equations[vol.number] = [self.pde]

        # Solution values are prescribed on all boundaries
        for vol in self.grid[0].unit_set_with_name['boundaries']:
            domain_equations[vol.number] = []
```

3.14.4 Discretized equation

Since PyGDH allows discretized equations to be written without details of the discretization formulas, the discretized equation method will be identical to that used in the previous chapter:

3.14.5 Boundary values

The boundary values are defined by the same mechanism as before, but the user is again encouraged to make use of the set definitions:

```
    def set_boundary_values(self):

        # Defined for clarity
        T = self.grid[0].field[0][0]

        ## Prescribed value of zero on all outside boundaries except for the
        ## 'right' boundary

        for vol in self.grid[0].unit_set_with_name['left']:
            T[vol.number] = 0

        for vol in self.grid[0].unit_set_with_name['bottom']:
            T[vol.number] = 0
```

(continues on next page)

(continued from previous page)

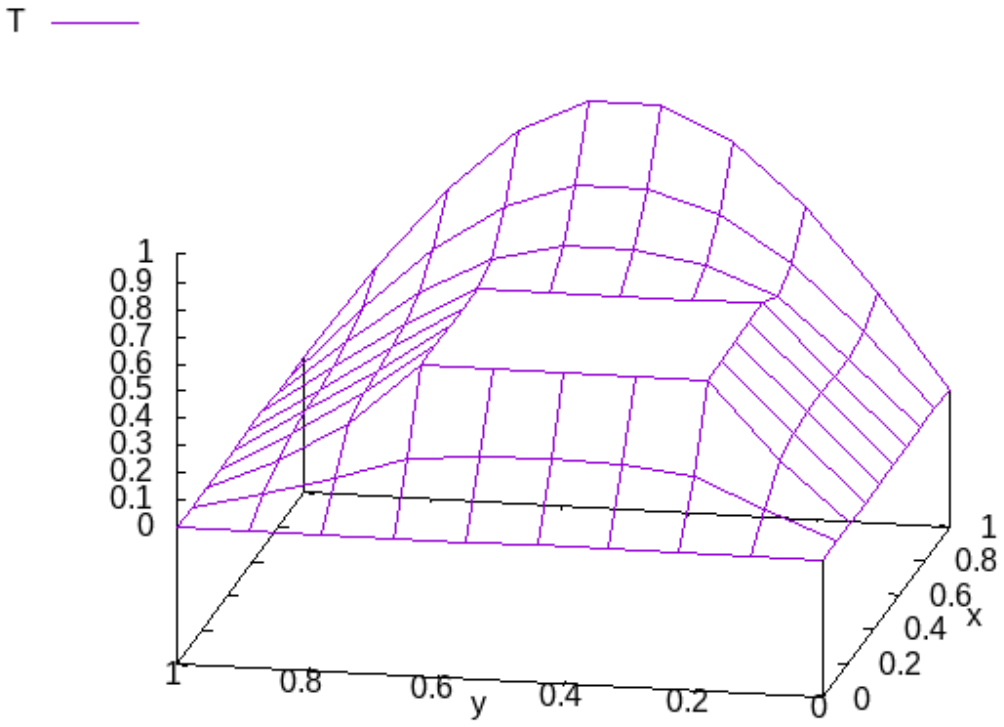
```
        for vol in self.grid[0].unit_set_with_name['top']:
            T[vol.number] = 0

        # This boundary condition is consistent with the other conditions at the
        # corners, as it should be.
        for vol in self.grid[0].unit_set_with_name['right']:
            T[vol.number] = math.sin(math.pi*vol.coordinate[1])

        ## Prescribed value of zero on all inner boundaries

        for vol in self.grid[0].unit_set_with_name['inner_left']:
            T[vol.number] = 0.5

        for vol in self.grid[0].unit_set_with_name['inner_bottom']:
            T[vol.number] = 0.5

        for vol in self.grid[0].unit_set_with_name['inner_top']:
            T[vol.number] = 0.5

        for vol in self.grid[0].unit_set_with_name['inner_right']:
            T[vol.number] = 0.5
```

3.14.6 Solution

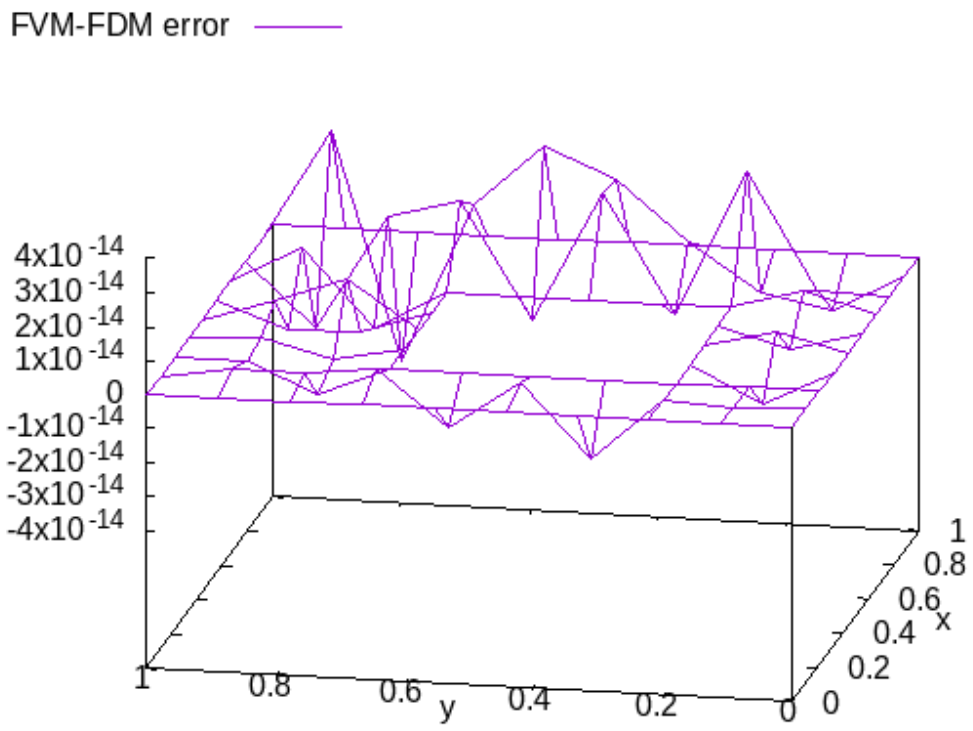
The solution is obtained as before.

```
problem = Laplace_Equation(10)

# The names of all output files will begin with this string
filename_root = 'multiply_connected'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

problem.solve_time_independent_system(filename_root, output_types)
```

The results are plotted below:



As before, a finite-difference approximate was used to validate the finite-volume solution, and the results are plotted below:



The errors are much smaller than the terms appearing in the equations.

Output was produced with the following GNUPLOT script:

```
set term png
set output 'multiply_connected.png'
set key left top
set view 67,281
set xlabel 'x'
set ylabel 'y'
splot 'multiply_connected_domain.gnuplot' using 1:2:3 title 'T' with lines
set output 'multiply_connected_validation.png'
splot 'multiply_connected_domain.gnuplot' using 1:2:4 title 'FVM-FDM error' with lines
set output
```

3.15 Compiling with Cython to increase speed

PyGDH was designed for ease-of-use and flexibility, with little emphasis on program speed, because many fast and powerful, but complex, software packages are already available. However, simple models often grow into more complicated models, so there is sometimes a need to make PyGDH programs run faster.

One important approach to increasing program speed is to compile programs to produce instructions in the computer's native language. The Cython compiler enables this for programs written in a language very similar to the Python language. Typically, a Cython program is first written and tested as a Python program, and then slight modifications

are made to provide the Cython compiler with additional information that allows it to produce a faster compiled program.

This additional information might tell the Cython compiler that particular variables will only refer to quantities of a specific type, such as integers, floating point numbers, or NumPy arrays. Hints like these help the compiler to optimize the resulting program.

3.15.1 Overview

Like in situations requiring *Postprocessing*, users wanting to use Cython should create one text file (called, for example, `body.py`) containing most of the Python code that describes the mathematical problem to be solved, and multiple text files that `import` this code and do high-level things such as telling PyGDH to test the problem formulation, solve the problem or to retrieve and postprocess previously computed solutions (these might be called `test.py`, `solve.py` and `postprocess.py`). By using such a strategy, data structures are defined only once in `body.py` but can be easily accessed from any of the higher-level Python source code.

During the debugging and testing phase, users will typically compute solutions on relatively small domains in order to rapidly obtain test results. In this phase, users can take advantage of the flexibility of a full Python environment, which allows users to interactively probe the operation of a running program. However, once this phase is complete, users may want to compute solutions on relatively large domains and program speed may become important. It is at this point that one might want to make use of the Cython compiler.

In order to do this, a text file containing Cython source code must be created. Typically, one starts by making a copy of `body.py` in our example, and naming the new file with a `pyx` extension (`body.pyx`), the extension for a Cython source code file. One then makes the desired changes to the new file to provide hints to the Cython compiler and then runs the Cython compiler to produce a compiled library file, which for the purposes of the Python `import` statement can be used just as the original `body.py` file. In fact, if both are present in the same directory, the compiled library file will be automatically used in preference to the original Python file. Typically, the `test.py`, `solve.py` and `postprocess.py` will run without any modifications.

PyGDH itself makes use of Cython; the PyGDH package contains both Python and Cython source code files, and during the default installation, the Cython source files are automatically compiled and installed alongside the Python source files if Cython and a C compiler are available.

3.15.2 An example

All of the examples in this tutorial are very small problems relative to the amount of computing power available to typical users. However, for illustration, we will revisit the problem in *Solving systems of equations* and increase the computational cost by increasing the grid size, decreasing the timestep size, and increasing the number of timesteps beyond what would be desirable in practical terms. We begin by copying the source code file and splitting it into a `system_solve.py` and a `system_body.py`, and then modify `system_solve.py` in order to increase the problem size:

```
import system_body

# Create an object instance
problem = system_body.System(81, 1., 1., 2., 1.)

# The names of all output files will begin with this string
filename_root = 'system'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

timestep_size = 2.5e-1
```

(continues on next page)

(continued from previous page)

```
timesteps = 20

# Calculate solution and write output to file
problem.solve_time_dependent_system(filename_root, output_types,
                                    timestep_count=timesteps,
                                    timestep_size=timestep_size)
```

Note that one must now indicate that the `System` class is in the imported `system_body` module.

Profiling

Profilers provide a detailed look at program performance, which can be extremely helpful when optimizing programs. It is a well-known programmer's rule of thumb that 90% of program running time is spent executing instructions representing 10% of its source code, so it is in the programmer's interest to spend most of their optimization effort on improving the performance of this critical 10%.

It is also frequently said that premature optimization is the "root of all evil," in part because the importance of any given piece of source code to the overall program running time is only fully known once the entire program is running correctly. So users are advised to ensure that their Python programs are producing correct results before transforming them into Cython programs.

The Python Standard Library provides profiling capabilities. In the present example, `system_solve.py` can be modified to find the 15 functions that take the most time:

```
import cProfile
import system_body
import sys

assert len(sys.argv) == 2

# Create an object instance
volume_count = 81
grid_obj = system_body.Domain('domain',volume_count)

N = 1.
D_1 = 2.
D_2 = 1.
k = 1.
problem = system_body.System([grid_obj], N, D_1, D_2, k)

# The names of all output files will begin with this string
filename_root = 'system'

# One output file will be created for each format given here
output_types = ['GNUPLOT']

timestep_size = 2.5e-1
timesteps = 20

# Calculate solution and write output to file
cProfile.run('problem.solve_time_dependent_system(filename_root, output_types,␣
↪timestep_count=timesteps, timestep_size=timestep_size)',sys.argv[1])
```

yielding:

```
Fri Jan 29 09:20:32 2021    system_uncompiled_profile.dat

         747 function calls in 3.978 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       20    3.897    0.195    3.897    0.195 {built-in method scipy.optimize._
↪minpack._hybrd}
        1    0.055    0.055    3.978    3.978 {method 'solve_time_dependent_system'␣
↪of 'pygdh.problem.Problem' objects}
       20    0.023    0.001    0.023    0.001 {method 'system_residuals' of 'pygdh.
↪problem.Problem' objects}
       20    0.001    0.000    3.922    0.196 minpack.py:169(_root_hybr)
       20    0.001    0.000    3.922    0.196 minpack.py:49(fsolve)
       20    0.000    0.000    0.024    0.001 minpack.py:25(_check_func)
       20    0.000    0.000    0.000    0.000 shape_base.py:25(atleast_1d)
       40    0.000    0.000    0.000    0.000 numerictypes.py:293(issubclass_)
      100    0.000    0.000    0.000    0.000 minpack.py:151(<genexpr>)
       20    0.000    0.000    0.000    0.000 numerictypes.py:365(issubdtype)
        1    0.000    0.000    3.978    3.978 {built-in method builtins.exec}
       20    0.000    0.000    0.000    0.000 {method 'flatten' of 'numpy.ndarray'␣
↪objects}
       40    0.000    0.000    0.000    0.000 {built-in method numpy.array}
       60    0.000    0.000    0.000    0.000 {built-in method builtins.issubclass}
       20    0.000    0.000    0.000    0.000 getlimits.py:365(__new__)
      100    0.000    0.000    0.000    0.000 {method 'get' of 'dict' objects}
       20    0.000    0.000    0.000    0.000 {method 'update' of 'dict' objects}
       20    0.000    0.000    0.000    0.000 numeric.py:469(asarray)
       20    0.000    0.000    0.000    0.000 {method 'pop' of 'dict' objects}
       40    0.000    0.000    0.000    0.000 fromnumeric.py:1785(shape)
       20    0.000    0.000    0.000    0.000 optimize.py:137(_check_unknown_options)
        1    0.000    0.000    0.000    0.000 _bootlocale.py:33(getpreferredencoding)
       40    0.000    0.000    0.000    0.000 {built-in method builtins.len}
       20    0.000    0.000    0.000    0.000 numeric.py:541(asanyarray)
       20    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        1    0.000    0.000    0.000    0.000 {built-in method _locale.nl_langinfo}
        1    0.000    0.000    3.978    3.978 <string>:1(<module>)
       20    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 codecs.py:186(__init__)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
```

The functions toward the top of this listing and which are in files written by the user (particularly `system_body.py` here) are the most important to optimize. Unsurprisingly, these are methods that perform mathematical calculations and which are repeatedly called.

Cython modifications

We now introduce Cython-specific program modifications, concentrating on the methods that are the most important to optimize. Typically, the compilation step involves its own debugging and testing phase. More details and deeper insights are available in the Cython documentation. In practice, not all Cython hints improve performance; some worsen performance, which may be a consequence of the small problem size. So alternating between profiling and adding Cython-specific changes is important.

Compilation with Cython is discussed in the Cython documentation. One method for using the Cython compiler is to use Python's Distutils mechanism. This involves creating another Python program, named `setup.py` by convention, that describes the compilation process and here is adapted from the program provided in the Cython documentation:

```
import numpy
from distutils.core import setup
from Cython.Build import cythonize
import sys
setup(
    include_dirs=[numpy.get_include()],
    ext_modules = cythonize("system_body.pyx",include_path=sys.path)
)
```

This is then run in the command-line environment:

```
python setup.py build_ext --inplace
```

This creates a compiled library file. Rerunning `system_profile.py`, this compiled library file is automatically used in preference to the `system_body.py` file. This produces:

```
Fri Jan 29 09:20:38 2021    system_compiled_profile.dat

         747 function calls in 3.864 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       20    3.796    0.190    3.796    0.190 {built-in method scipy.optimize._
↪minpack._hybrd}
        1    0.043    0.043    3.863    3.863 {method 'solve_time_dependent_system'␣
↪of 'pygdh.problem.Problem' objects}
       20    0.023    0.001    0.023    0.001 {method 'system_residuals' of 'pygdh.
↪problem.Problem' objects}
       20    0.001    0.000    3.820    0.191 minpack.py:169(_root_hybr)
       20    0.001    0.000    3.821    0.191 minpack.py:49(fsolve)
       20    0.000    0.000    0.023    0.001 minpack.py:25(_check_func)
       40    0.000    0.000    0.000    0.000 numerictypes.py:293(issubclass_)
       20    0.000    0.000    0.000    0.000 numerictypes.py:365(issubdtype)
      100    0.000    0.000    0.000    0.000 minpack.py:151(<genexpr>)
       20    0.000    0.000    0.000    0.000 shape_base.py:25(atleast_1d)
        1    0.000    0.000    3.864    3.864 {built-in method builtins.exec}
       20    0.000    0.000    0.000    0.000 {method 'flatten' of 'numpy.ndarray'␣
↪objects}
       20    0.000    0.000    0.000    0.000 getlimits.py:365(__new__)
       40    0.000    0.000    0.000    0.000 {built-in method numpy.array}
       60    0.000    0.000    0.000    0.000 {built-in method builtins.issubclass}
       20    0.000    0.000    0.000    0.000 {method 'update' of 'dict' objects}
      100    0.000    0.000    0.000    0.000 {method 'get' of 'dict' objects}
       40    0.000    0.000    0.000    0.000 fromnumeric.py:1785(shape)
       20    0.000    0.000    0.000    0.000 {method 'pop' of 'dict' objects}
       20    0.000    0.000    0.000    0.000 numeric.py:469(asarray)
       20    0.000    0.000    0.000    0.000 numeric.py:541(asanyarray)
       20    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
       20    0.000    0.000    0.000    0.000 optimize.py:137(_check_unknown_options)
       40    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 _bootlocale.py:33(getpreferredencoding)
       20    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
        1    0.000    0.000    0.000    0.000 {built-in method _locale.nl_langinfo}
```

(continues on next page)

(continued from previous page)

```
        1    0.000    0.000    3.863    3.863 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 codecs.py:186(__init__)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
```

The compiled methods are no longer visible to the profiler as separate methods, but the methods that call the compiled methods are still visible and the total running time has decreased. It should be noted that measured running times depend on the present state of the computer and will therefore vary if measured at different times.

Unfortunately, the improvement in program speed was fairly small in this example, although compilation can prove a substantial benefit in mathematically-intensive problems, in which the time spent calculating values is large compared to the overhead of controlling program flow.

Maintaining consistency between Python and Cython source code files

Even mature PyGDH programs, for which Cython versions have already been developed, might see further development. In this case, it is helpful to make changes in only one version of the source code files, rather than duplicating effort and potentially introducing mistakes in the process. Since further development of a model typically involves returning to the debugging and testing phase, it is reasonable to work with the Python version of the source code, but it is desirable to avoid redoing all of the work originally done to create a Cython version for the modified Python program.

This same challenge was encountered in the development of PyGDH itself; both Python and Cython versions of the source code are provided. Based on our experience, we suggest the following procedure (in a GNU/Linux or similar environment):

1. Produce a mature set of Python source code files.
2. Transform these into a set of Cython source code files, based on profiling information.
3. Use the `diff` utility to produce a "unified patch" file containing the changes in the Cython version relative to the Python version:

   ```
   diff -u body.py body.pyx > body.patch
   ```

4. Modify and test the Python version
5. Use the `patch` utility to apply the saved changes to the Python version, producing a new Cython version (and saving a backup):

   ```
   patch -b -o body.pyx body.py body.patch
   ```

6. Examine the new Cython version to ensure that the changes are reasonable.
7. Test the new Cython version, adding new modifications if desired.
8. Repeat from step 3 if further changes to the model are necessary.

3.15.3 Suggested exercises

1. Profile the two-dimensional Laplace equation solver, create a Cython version, and run it.

CHAPTER

FOUR

TEMPLATES FOR PROGRAMS USING PYGDH

The contents between angle brackets in the templates in the `pygdh/examples/templates` directory within the PyGDH archive can be replaced to create simple programs using PyGDH. Templates are provided for a time-dependent ODE, a spatially-dependent ODE, and a PDE with time and position as independent variables.

CHAPTER

FIVE

CITING PYGDH

Development and packaging of PyGDH required significant time and effort, and it is hoped that PyGDH will save significant time and effort for other researchers. This work appears in the Journal of Open Source Software. Please reference the published article when citing PyGDH.

CHAPTER

SIX

BACKGROUND AND ACKNOWLEDGEMENTS

This work was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Vehicle Technologies of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 under the Batteries for Advanced Transportation Technologies (BATT) Program and the Advanced Battery Materials Research (BMR) Program.

PyGDH was initially developed in the course of BATT-funded and BMR-funded research by Kenneth F. Higa under the guidance of Venkat Srinivasan, within the Energy Storage and Distributed Resources Department (now the Energy Storage and Distributed Resources Division) in the Environmental Energy Technologies Division (now the Energy Technologies Area) of Berkeley Lab (Lawrence Berkeley National Laboratory), a U.S. Department of Energy laboratory operated by the University of California. "Bringing Science Solutions to the World" is Berkeley Lab's mission.

PyGDH continues to receive updates from KFH under supervision of Vincent Battaglia at Berkeley Lab, supported through funding from the Vehicle Technologies Office (VTO) and Advanced Manufacturing Office (AMO) within the U.S. Department of Energy.

KFH thanks the developers of the Python programming language and its standard libraries, the developers of the NumPy, SciPy, h5py, and HDF5 libraries (and the underlying libraries), the developers of the Sphinx Python Documentation Generator, the GNUPLOT, matplotlib, the Cython compiler, LaTeX, and developers throughout the Open Source community in general.

KFH also thanks Ashlea Patterson, Sunil Mair, Anne Suzuki, Fabiola Lopez, Andrew Veenstra, Tiffany Ho and Fiona Stewart for improving the user experience by providing valuable feedback on early versions of the PyGDH documentation and software.

While great effort has been made to eliminate bugs, they undoubtedly exist. The PyGDH source code is open for all to examine. Users are encouraged to send bug reports and fixes to KFH <KHiga AT LBL DOT gov> for inclusion in the library.

In the spirit of scientific openness and in the interest of strengthening public confidence in the work of the scientific community, KFH respectfully encourages researchers receiving public funding to make their source code publicly available when publishing scientific results. It is hoped that PyGDH will encourage this by keeping problem-specific source code to a reasonable size. KFH thanks Greg Wilson of Software Carpentry for promoting openness in software development within the scientific community and providing the inspiration to release this work as open-source software.

Some of the concepts underlying PyGDH are described in the excellent introductory text:

Patankar, Suhas V. *Numerical Heat Transfer and Fluid Flow*, Hemisphere, 1980.

KFH expresses appreciation for the work of the Journal of Open Source Software and thanks reviewers Jamie J. Quinn and Wei Zhao for their detailed and helpful advice in improving this package, and editor Melissa Weber Mendonça for guiding this submission through the peer review process.

CHAPTER

SEVEN

COPYRIGHT NOTICE

CHAPTER

EIGHT

LICENSE