Adding a new model to NPSGD

Thomas Dimson
Natural Phenomenon Simulation Group (NPSG)
David R. Cheriton School of Computer Science
University of Waterloo, Canada

January 25, 2011

Contents

1 Introduction

NPSGD was designed to to make it easy to add new models. Within a NPSGD, a "model" is a Python class that inherits from a specific base class. This class often is a wrapper for an existing implementation of a model in another language such as C++ or Matlab. By editing these model wrappers, NPSGD will create a create the web interface with the corresponding parameters. It is up to the module to perform the work of running the model and giving meaningful output to the user.

2 Quick Start

The Python model wrappers of NPSGD are located in the `models/` subdirectory an extension of `.py`. To quickly implement a model wrapper, simply copy `example.py` to a new filename called `my\_model.py`. Edit this file and assign the `short\_name` class variable to `my\_model`. Your model will now be accessible to all of the NPSGD daemons. In particular, you will be able to access `my_model` via the `npsgd_web` daemon at `http://npsgdserver:8000/models/my_model`.

Please note that you will have to make sure that the model is available on all servers running NPSGD daemons (queue, web, and workers) for NPSGD to function correctly.

3 Models

In NPSGD, a "model" is actually a Python class that inherits, from the base class of `ModelTask`. This gives the user the flexibility to be language-agnostic in terms of model implementation, with a quick Python wrapper as a model runner.

All model wrappers (with a .py extension) are placed in the `models/` subdirectory. Periodically, the **queue**, **web**, and **worker** daemons will scan this directory for conforming (or newly timestamped) classes and load them into memory. From there, the model will be immediately accessible from the web, and therefore from the worker and queue. By default, the scanner propagation time is 120 seconds. If the wrapper file is ever removed, the model will continue to be served in memory so that older requests can complete.

Model wrappers are loaded with an associated "version" - a MD5 hash of the file that was used to declare the model wrapper. All requests in NPSGD are keyed by both the model name and this version value. When a requests comes in from the web page, a hidden parameter is sent telling the `npsgd_web` what version of the model the user was looking at. This allows for you to update the models on disk without destroying requests that are are already in memory (since they will execute under the old version).

The model versioning also allows worker machines to have a subset of the models available to the queue and web machines. When a worker polls the queue, it gives the queue a list of supported model-version pairs. The queue will only return requests that were made with models available in this list.

4 Helper classes

Rather than inheriting from `ModelTask` directly, NPSGD provides two helper classes that can greatly speedup creation of models. These are `StandaloneTask` and `MatlabTask`.

4.1 StandaloneTask

Models inheriting from `StandaloneTask` are wrappers for standalone executables that communicate using command line arguments (e.g., specifying the path to specific data input). When such a model executes the pre-requisites are prepared in the working directory and a subprocess is spawned using the executable and command line parameters.

4.2 MatlabTask

`MatlabTask` aids in connecting models that were programmed in Matlab. Essentially, this allows the user to access web parameters *directly* in Matlab code, bypassing the need worry about technical details such as spawning subprocesses to run the model.

5 NPSGD Pipeline

The purpose of NPSGD is to perform all the administrative work needed to deliver data to your model implementation. It is useful to understand exactly the processes that occurs in order for this to happen, though the model implementor will not have to touch this pipeline at all.

1. A model that is implemented in `models/example.py` is noticed and loaded by all NPSGD daemons.
2. Online web user visits the web interface for the model, typically at `http://npsgdserver:8000/models/example`. The user specifies all the parameters present in the model and clicks submit.
3. Web daemon **npsgd_web** submits the request to the queue.
4. Queue daemon, **npsgd_queue** sends an email to the user with a confirmation code for the request.
5. User receives email, clicks the confirmation code link which is typically `http://npsgdserver:8000/confirm_submission/code`. The web daemon submits the confirmation code to the queue daemon at this time.
6. Queue daemon waits for a worker to poll. When a worker polls with this model available, queue will hand off the request.
7. Worker spawns the model class with the parameters that were specified in the web interface. Model proceeds with model pipeline documented in Section 6.
8. Model pipeline completes, sends an e-mail to the user and then tells the queue that everything has completed successfully.

6 Model Pipeline

The model pipeline is the section that a model implementor will actually have to create in the model. Typically, these follow a strict order that is specified in `ModelTask` class under the method `run`:

1. A working directory is created for the model (usually `/var/tmp/npsgd/unique\_id`.
2. Execution is setup in the `prepareExecution` method. This is generally done by converting parameters into a form (e.g., a file in the working directory) that the model can work with.
3. The model is run under the `runModel` method. A typical model run includes calling the subprocess with specified parameters and waiting for the result.
4. Graphs are prepared under the `prepareGraphs` method using the data that was outputted from runModel.
5. PDF document is created in `generatePDF`, through `getAttachments` using PDFLatex.
6. E-mail is sent using `sendResultsEmail`.

If any of these stages fail, a message will be communicated back to the queue telling it that it failed to execute the task. The queue will recycle the request a fixed number of times (in case the error is transient) and will send an error e-mail eventually if it continues to fail.

Note that the pipeline can be greatly simplified by using the helper classes outlined in Section 4.

7 Model Implementation

This section is a basic guide about specifying a model to run.

7.1 Defining a class

A model must inherit from `ModelTask`, or one of the specialized helper classes specified in Section 4. The model must be saved under `models/my\_model.py`. For example, a skeleton model could take the form of

```
from npsgd.model_task import ModelTask
from npsgd.model_parameters import *

class MyModel(ModelTask):
    short_name = "my_model"
    full_name  = "My Model"
    subtitle   = "Super example model"
```

7.2 Parameters

All model parameters are specified in the class structure itself, in a list with the name `parameters`. As an example to start, we could specify an integer representing the number of samples to take into account during a simulation, with a default of 1000:

```
...
class MyModel(ModelTask):
    ...
    parameters = [
        ...
        IntegerParameter('nSamples', description="Number of
            samples to specify",
             default=1000)
        ...
    ]
```

7.2.1 Parameter Options

All parameters support a variety of different options in order to modify its behaviour. The first value passed into the parameter is always the `name`, which is used as a reference through NPSGD. The rest of the options are specified using keyword arguments. Each parameter type outlined in Section 7.2.2 has a superset of these options:

Option	Description
name	Unique identifier used to reference the parameter through NPSGD. These generally should not have spaces, and by convention are camel cased.
description	As it says, used to describe the parameter. The string specified here will appear in a number of places, including the output LaTeX document, and the HTML page.
default	Used to specify the default value of a parameter.
hidden	Boolean used to toggle whether the parameter is hidden or not. Currently, this only affects the output HTML. Hidden parameters will simply be passed along with their default value. This is useful for subclassing models
helpText	Used to give "helpful hints" when a user is specifying the model. This currently appears as balloon text on the HTML output page.

7.2.2 Parameter Types

Currently, NPSGD supports the following parameter types:

- `StringParameter`: Basic string input. The only extra option is `units`, which represents the units of input.
- `FloatParameter`: Basic float input. Float parameters can be used to specify a specific range of inputs by specifying the `rangeStart`, `rangeEnd` and `step` inputs. If both `rangeStart` and

`rangeEnd` are specified, the float parameter will function as a slider. If only `rangeStart` or `rangeEnd` are specified, the input will be clamped to values accordingly. Additionally, floats take a `units` parameter that can be used to specify the units of the float (*e.g.,* nm, cm)

- `IntegerParameter`: Basic integer input. Integers have exactly the same options as `FloatParameter`, but will be verified to ensure the number is an integer.
- `RangeParameter`: Range parameters are used for specifying a range of floating point inputs (*e.g.,* 400-2500nm). Options for `RangeParameter` match `FloatParameter`, but must have all of `rangeEnd`, `rangeEnd` and `step` specified. The output and default values of a range must be specified as a pair of floats, *e.g.,* `default=(1,5)`, representing the range of choices.
- `SelectParameter`: Select parameters are used to clamp input to a specific set of options, something like a combo box. Select parameters have an option called `options`, which is a list of valid inputs that the select box can take (*e.g.,* "Strong", "Weak").
- `BooleanParameter`: Boolean parameters are similar to the `SelectParamater` type, but now are clamped to true/false. These parameters will display as text boxes in HTML and have no additional options.

All parameter types are declared in `npsgd/model\_parameters.py`. Custom parameter types could be based off these examples and declared in any file that is accessible to the model.

7.2.3 Attachments

By default, NPSGD includes only the PDF created via LaTeX, outlined in Section 7.2.5 as an attachment. If your model creates more output (such as data files, graphs, pictures, etc.) then you may want to add additional attachments. In NPSGD, this is performed by adding a class variable by the name of `attachments` consisting of a list of all additional attachments within the working directory to include. For example,

```
...
class MyModel(ModelTask):
    ...
    attachments = [picture.jpg, data.txt]
    ...
```

The above code listing would include two e-mail attachments (`picture.jpg` and `data.txt`) along with the usual `results.pdf`.

If you want more flexibility in specifying attachments, consider overriding the `getAttachments` method on `ModelTask`.

7.2.4 Graphical Output

After an executable has run, usually a task will create graphs out of the output In NPSGD, this can be accomplished by saving files in the working directory by overriding the `prepareGraphs` method in a model. Python has an excellent library called `matplotlib` (`http://matplotlib.sourceforge.net/`) which creates graphs that are on-par with Matlab's using commands that are almost identical to Matlab plot syntax. For example,

```
...
import os
import matplotlib
matplotlib.use("Agg") #suppress graphical user interface
import matplotlib.pyplot as plt
...
class MyModel(ModelTask):
    ...
    attachments = [plot.png]
    ...
    def prepareGraphs(self):
        x = [1,2,3,4,5]
        y = [10,5,2,6,7]
        plt.clf() #clear previous plot
        plt.plot(x,y)
        plt.xlabel("X")
        plt.ylabel("Y")
        plt.title("Demo Plot")
        plt.savefig(os.path.join(self.workingDirectory, "plot.
            png"))
    ...
```

Note that the `prepareGraphs` method is completely optional - if your model does not have graphical output, or creates the output inside the executable then you may include the relevant files using the attachment mechanism outlined in Section 7.2.3.

7.2.5 LaTeX Output

Output for all models is generally routed through `pdflatex`. Each model will **definitely** want to override the `latexBody` method. This method returns a string that is inserted into the LaTeX template available in `templates/latex/result\_template.tex`. The resultant tex file is then run through `pdflatex` in order to generate a file called `result.pdf`. This file, by default, is included in every e-mail that is sent out.

`ModelTask` has a method for creating a LaTeX table containing all the parameters that the user has specified. By including `self.latexParameterTable()` somewhere in the LaTeX output the output PDF will contain a very nicely formatted parameter table.

A complete example:

```
...
class MyModel(ModelTask):
    ...
    def latexBody(self):
        return r"""
        Hello there!

        \section{Main Section}
```

```
            This is the result of an example model run for NPSGD.
               Your
            parameters were:

            %s
""" % self.latexParameterTable()
```

It is highly recommended that you use Python docstrings (triple quoted strings) in order to specify output, as well as using the `r` prefix to the string (raw string mode, so you do not have to escape slashes).

7.2.6 Reading Parameter Values

When executing your script, preparing graphs and outputting LaTeX it is often necessary to have access to the parameters that the user has specified at the web interface. These are *automatically* delivered to the script using the names that you specified in for your parameters. By accessing `self.parametername.value` in any instance method, you will get access to the value that the user specifies. This is best illustrated by example:

```
...
class MyModel(ModelTask):
    ...
    parameters = [
        ...
        IntegerParameter('nSamples', description="Number of
            samples to specify",
             default=1000)
        ...
    ]

    def prepareGraphs(self):
        print self.nSamples.value #Will output the number of
            samples the
                                  #user specified
```

7.2.7 Helpers: StandaloneTask

Subclassing `StandaloneTask` automates the process of running a subprocess in order to execute a model on the command line. This is the most technical part of the process.

`StandaloneTask` specifies a method of `runModel` that simply executes a command as a Python subprocess, and stores the stderr/stdout of the subprocess in instance parameters `self.stdout` and `self.stderr`. The subprocess is executed within the model's working directory.

The model creator must specify one additional parameter, and one additional method for running. The class variable `executable` specifies the path to the executable we wish to run (typically the model executable, or something like `java` for a Java task). The instance method `executableParameters` returns a list of parameters for the executable along with values. The

parameters are specified as a Python list. A developer might take the values specified in the parameter list for the model wrapper to pass in values to the underlying script. This is best shown via example:

```
from npsgd.model_task import StandaloneTask
from npsgd.model_parameters import *

class LsModel(StandaloneTask):
    ...
    executable = "ls"
    ...
    def executableParameters(self):
        return [
            "-al",
            "/var/tmp"
        ]
```

Such a model will execute `ls -al /var/tmp` in a subprocess and return the results in `self.stdout`. Many more examples ship along with NPSGD.

7.2.8 Helpers: MatlabTask

Subclassing `MatlabTask` automates the process of spawning a Matlab subprocess, which can be a tricky and time consuming process. It also delivers the parameter values **directly** into the Matlab environment. The script will just "magically" have access to all the values of user input in variables that match the names specified in the model class. For example, if you declare a parameter named "myInt" and the user specifies a value of "5", your Matlab script will execute in an environment where the "myInt" variable is assigned 5.

NPSGD implements parameter delivery by creating a wrapper script on the fly that is fed into Matlab. This Matlab script has all the assignment statements for the parameters passed in and finishes with a call to the function that you specify, as described below.

A user of the `MatlabTask` helper need only specify one additional class parameter, namely `matlabScript` which gives the location of the script Matlab should execute. A full example of using `MatlabTask` is specified in Appendix A.

A Complete Example of a Matlab Task

A.1 NPSGD Code

```
from npsgd.matlab_task import MatlabTask
from npsgd.model_parameters import StringParameter, IntegerParameter
    , RangeParameter, FloatParameter

class ExampleModel(MatlabTask):
    short_name = 'example'
```

```
    full_name  = 'Example Model'
    subtitle   = 'A demo model'

    parameters = [
        StringParameter('test',      description="This a test string
            "),
        IntegerParameter('graphEnd', description="Graph end point"),
        RangeParameter('ranger',     description="Sample range
            parameter",\
                rangeStart=400, rangeEnd=700, step=1),
        FloatParameter('floater', description="Sample Float
            Parameter", rangeStart=10, rangeEnd=1000, step=1)
    ]

    matlabScript  = '/home/tdimson/public_html/npsg/npsgd/models/
        example/example.m'

    def latexBody(self):
        return r"""
            This is a test of including a figure.
            \begin{figure}
            \caption{A nice looking function}
            \includegraphics[width=5in]{test_figure}
            \end{figure}

            \newpage\appendix\section{Parameter List}
            %s
""" % self.latexParameterTable()
```

A.2 Matlab Code

```
x = rangerStart:1:rangerEnd;
y = x.^2;

plot(x,y)
title('Plot of y = x^2')
print -dpng test_figure
```

B Complete Example of ABM-U

ABM-U is a more sophisticated example of a model task. An example of this code running is available at `http://www.npsg.uwaterloo.ca/models/ABMU.php`.

```
# Author: Thomas Dimson [tdimson@gmail.com]
# Date:   January 2011
# For distribution details, see LICENSE
import os
import sys
```

```
import csv
import json
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from npsgd.standalone_task import StandaloneTask
from npsgd.model_parameters import *

class ABMU(StandaloneTask):
    short_name = 'abmu_c'
    full_name  = 'ABM-U'
    subtitle='Algorithmic BDF Model Unifacial'
    parameters = [
            IntegerParameter('nSamples', description="Number of
               samples",
                rangeStart=1000, rangeEnd=100000, step=1, default
                   =10000),
            RangeParameter('wavelengths', description="Wavelength
               Range",
                rangeStart=400, rangeEnd=2500, step=5, units="nm",
                   helpText="Modeled spectral curves will be
                   generated in steps of 5nm."),
            FloatParameter('angleOfIncidence', description="Incident
                angle",
                default=8, rangeStart=0, rangeEnd=90, step=0.1,
                   units="degrees"),
            SelectParameter('surfaceOfIncidence', description="
               Surface of incidence",
                options=["Adaxial", "Abaxial"], default="Adaxial",
                helpText="The adaxial surface corresponds to the top
                    epidermal layer of the leaf, while the abaxial
                   surface corresponds to the bottom epidermal layer
                   ."),
            FloatParameter('wholeLeafThickness', description="Leaf
               thickness",
                default=2.04e-4, units="m"),
            FloatParameter('mesophyllPercentage', description="
               Mesophyll percentage",
                default=80, units="%", rangeStart=0, rangeEnd=100,
                   step=0.1,
                helpText="Percentage of the total leaf thickness
                   occupied by the mesophyll tissue."),
            FloatParameter('carotenoidConcentration', description="
               Carotenoid concentration",
                default=0.000658895, units="g/cm^3", rangeStart=0.0)
                   ,
            FloatParameter('chlorophyllAConcentration', description=
               "Chlorophyll A concentration",
```

```
            default=0.002895146, units="g/cm^3", rangeStart=0.0)
                ,
            FloatParameter('chlorophyllBConcentration', description=
               "Chlorophyll B concentration",
                default=0.00079866, units="g/cm^3", rangeStart=0.0),
            FloatParameter('proteinConcentration', description="
               Protein concentration",
                default=0.05308714, units="g/cm^3", rangeStart=0.0),
            FloatParameter('celluloseConcentration', description="
               Cellulose concentration",
                default=0.05318708961, units="g/cm^3", rangeStart
                   =0.0),
            FloatParameter('linginConcentration', description="
               Lingin concentration",
                default=0.006058529380, units="g/cm^3", rangeStart
                   =0.0),
            FloatParameter('cuticleUndulationsAspectRatio',
               description="Cuticle undulations aspect ratio",
                default=10.0, rangeStart=1.0, rangeEnd=50.0, step
                   =0.5,
                helpText="Lower values result in more roughness and
                   a more diffuse behaviour."),
            FloatParameter('epidermisCellCapsAspectRatio',
               description="Epidermis cell caps aspect ratio",
                default=5.0, rangeStart=1.0, rangeEnd=50.0, step
                   =0.5,
                helpText="Lower values correspond to more prolate (
                   or rough) cell caps. This results in more
                   diffusion of the propegated light."),
            FloatParameter('spongyCellCapsAspectRatio', description=
               "Spongy cell caps aspect ratio",
                default=5.0, rangeStart=1.0, rangeEnd=50.0, step
                   =0.5,
                helpText="Lower values correspond to more prolate (
                   or rough) cell caps. This results in more
                   diffusion of the propegated light."),
            BooleanParameter('sieveDetourEffects', description="
               Simulate sieve and detour effects",
                default=True, helpText="To account for the non-
                   homogeneous distribution of pigments (for details
                   , please refer to our related publications).")
]

attachments   = ['spectral_distribution.csv', 'reflectance.png',
    'transmittance.png', 'absorptance.png']

executable = "/home/tdimson/public_html/npsg/abmb_abmu_cpp/abmu"
```

```
def executableParameters(self):
    if self.surfaceOfIncidence.value == "Abaxial":
        angleIn = 180 - self.angleOfIncidence.value
    else:
        angleIn = self.angleOfIncidence.value

    params = [
        "-d", os.path.join(os.path.dirname(self.executable), "
           data"),
        "-n", str(self.nSamples.value),
        "-p", str(angleIn),
        "-s", str(5), #step
        "-w", str(self.wavelengths.value[0]),
        "-e", str(self.wavelengths.value[1]),
    ]

    if not self.sieveDetourEffects.value:
        params.append("-q")

    params += ["sample.json",
               "spectral_distribution.csv"
    ]

    return params

def readDataTable(self):
    wavelengths, reflectance, transmittance, absorptance = ([],
       [], [], [])
    with open(os.path.join(self.workingDirectory, "
       spectral_distribution.csv"), 'r') as f:
        spectralReader = csv.reader(f)
        headers = [e.strip() for e in spectralReader.next()]
        wIndex = headers.index("wavelength")
        rIndex = headers.index("reflectance")
        tIndex = headers.index("transmittance")
        aIndex = headers.index("absorptance")

        for row in spectralReader:
            wavelengths.append(float(row[wIndex]))
            reflectance.append(float(row[rIndex]))
            transmittance.append(float(row[tIndex]))
            absorptance.append(float(row[aIndex]))

    return wavelengths, reflectance, transmittance, absorptance


def latexDataTable(self):
```

```
        wavelengths, reflectance, transmittance, absorptance = self.
            readDataTable()
        latex = r"""
        \begin{centering}
        \begin{longtable}{l l l l}
        \textbf{Wavelength} & \textbf{Reflectance} & \textbf{
            Transmittance} & \textbf{Absorptance} \\
        \hline
        \endhead
        %s
        \end{longtable}
        \end{centering}
        """ % "\n".join("%snm & %s & %s & %s\\\\" % (w,r,t,a) for (w
            ,r,t,a) in zip(wavelengths,reflectance, transmittance,
            absorptance))

        return latex

    def prepareExecution(self):
        with open(os.path.join(self.workingDirectory, "sample.json")
            , 'w') as f:
            f.write(json.dumps({
                "wholeLeafThickness": self.wholeLeafThickness.value,
                "cuticleUndulationsAspectRatio": self.
                    cuticleUndulationsAspectRatio.value,
                "epidermisCellCapsAspectRatio": self.
                    epidermisCellCapsAspectRatio.value,
                "spongyCellCapsAspectRatio": self.
                    spongyCellCapsAspectRatio.value,
                "palisadeCellCapsAspectRatio": 0.0,
                "linginConcentration": self.linginConcentration.
                    value,
                "proteinConcentration": self.proteinConcentration.
                    value,
                "celluloseConcentration": self.
                    celluloseConcentration.value,
                "chlorophyllAConcentration": self.
                    chlorophyllAConcentration.value,
                "chlorophyllBConcentration": self.
                    chlorophyllBConcentration.value,
                "carotenoidConcentration": self.
                    carotenoidConcentration.value,
                "mesophyllFraction": self.mesophyllPercentage.value
                    / 100
            }))

    def prepareGraphs(self):
```

```
wavelengths, reflectance, transmittance, absorptance = self.
    readDataTable()
axisWavelengthStart = wavelengths[0]
axisWavelengthEnd   = wavelengths[-1]
plotCommand = plt.plot
if len(wavelengths) == 1:
    axisWavelengthStart = wavelengths[0] - 100
    axisWavelengthEnd   = wavelengths[0] + 100
    plotCommand = plt.scatter


plt.clf()
plotCommand(wavelengths, [e*100 for e in reflectance])
plt.xlabel("Wavelength (nm)")
plt.ylabel("Reflectance (%)")
plt.title(self.full_name)
plt.axis([axisWavelengthStart, axisWavelengthEnd, 0, max(
    reflectance) * 100 + 5])
plt.savefig(os.path.join(self.workingDirectory, "reflectance
    .pdf"))
plt.savefig(os.path.join(self.workingDirectory, "reflectance
    .png"))
plt.clf()

plotCommand(wavelengths, [e*100 for e in transmittance])
plt.xlabel("Wavelength (nm)")
plt.ylabel("Transmittance (%)")
plt.title(self.full_name)
plt.axis([axisWavelengthStart, axisWavelengthEnd, 0, max(
    transmittance) * 100 + 5])
plt.savefig(os.path.join(self.workingDirectory, "
    transmittance.pdf"))
plt.savefig(os.path.join(self.workingDirectory, "
    transmittance.png"))
plt.clf()

plotCommand(wavelengths, [e*100 for e in absorptance])
plt.xlabel("Wavelength (nm)")
plt.ylabel("Absorptance (%)")
plt.title(self.full_name)
plt.axis([axisWavelengthStart, axisWavelengthEnd, 0, max(
    absorptance) * 100 + 5])
plt.savefig(os.path.join(self.workingDirectory, "absorptance
    .pdf"))
plt.savefig(os.path.join(self.workingDirectory, "absorptance
    .png"))
plt.clf()
```

```
def latexBody(self):
    return r"""
        These are the results of your run of the \textbf{ABM-U}
           model provided by the
        Natural Phenomenon Simulation Group (NPSG) at the
           University of Waterloo.

        The ABM-U employs an algorithmic Monte Carlo formulation
        to simulate light interactions with unifacial plant
           leaves
        (e.g., corn and sugar cane). More specifically,
           radiation propagation
        is treated as a random walk process whose states
           correspond
        to the main tissue interfaces found in these leaves. For
            more
        details about this model, please refer to our related
           publications~\cite{Ba06,Ba07}.
        Although the ABM-U provides bidirectional readings,
        directional-hemispherical quantities (provided by our
           online system)
        can be obtained by integrating the outgoing light (rays)
            with respect
        to the outgoing (collection)
        hemisphere. Similarly, bihemispherical quantities can be
            calculated
        by integrating the BDF (bidirectional scattering
           distribution function)
        values with respect to incident and collection
           hemispheres.

        The provided spectral curves (directional-hemispherical,
            reflectance,
        transmittance and absorptance) were obtained considering
            an angle of incidence
        measured with respect to the specimen's normal (zenith).
            The curves
        were obtained using a virtual spectrophotometer~\cite{
           Ba01}.
        The researcher interested in BDF
        (bidirectional scattering distribution function)
        plots is referred to a publication describing the
           implementation of virtual
        goniophotometers~\cite{Kr04}. These publications can be
           found at:
        \url{http://www.npsg.uwaterloo.ca/pubs/measurement.php}
```

```
\begin{figure}
\begin{centering}
\includegraphics[width=5in]{reflectance}
\caption{Directional-hemispherical reflectance.}
\end{centering}
\end{figure}

\begin{figure}
\begin{centering}
\includegraphics[width=5in]{transmittance}
\caption{Directional-hemispherical transmittance.}
\end{centering}
\end{figure}

\begin{figure}
\begin{centering}
\includegraphics[width=5in]{absorptance}
\caption{Directional-hemispherical absorptance.}
\end{centering}
\end{figure}

\newpage
\begin{thebibliography}{9}
\bibitem{Ba01}
Baranoski,G.V.G.; Rokne,J.G.; Xu,G.
Virtual spectrophotometric Measurements for biologically
    and physically-based rendering.
\textit{The Visual Computer}, Volume 17, Issue 8, pp.
   506-518, 2001.

\bibitem{Ba06}
Baranoski G.V.G.
Modeling the interaction of infrared radiation (750 to
   2500 nm) with bifacial and unifacial plant leaves.
\textit{Remote Sensing of Environment}, 100(3):335-347,
   2006.

\bibitem{Ba07}
Baranoski G.V.G.; Eng D.
An investigation on sieve and detour effects affecting
   the interaction of collimated and diffuse infrared
   radiation (750 to 2500 nm) with plant leaves.
\textit{IEEE Transactions on Geoscience and Remote
   Sensing}, 45 (8):2593-2599, 2007.

\bibitem{Kr04}
Krishnaswamy,A.; Baranoski,G.V.G.; Rokne,J.G.
```

```
            Improving the reliability/cost ratio of goniophotometric
                comparisons.
            \textit{Journal of Graphics Tools}, Volume 9, Number 3,
               pp. 1-20, 2004.
            \end{thebibliography}

            \newpage
            \appendix
            \section{Parameter List}
            %s
            \newpage
            \section{Data List}
            %s
""" % (self.latexParameterTable(), self.latexDataTable())
```