Get with the Flow, Man!

A study of the thermodynamics of civil engineering

New Mexico Supercomputing Challenge

Final Report

April 1st, 2009

Team 52

Los Alamos High School

Team Members:

Ben Batha

Daniel Cox

Gannon Nelson

Will Phillips

Jake Poston

Sponsors:

Ms. Diane Medford

Mr. Lee Goodwin

Mentors:

Dr. Steven Batha

Dr. David Poston

Table of Contents

Summary

Any person who has driven on a cold New Mexico night is aware that bridges usually freeze before the rest of the road (some have learned this the hard way). The primary goal of this project is to understand heat conductivity through a bridge, and determine why the roadway on a bridge freezes before the road directly in the ground. The secondary goal is to design a computational model that can determine under which conditions a bridge may or may not freeze.

This simple question, "On a cold night, why does the bridge freeze before the road?" requires sophisticated physics to answer. There are a variety of considerations for such a model. What is the starting air temperature? What materials are used for the bridge? What is the heat capacity of each of these materials? To create an effective model we used a uniform concrete suspension bridge parallel to the ground. We assumed that the starting temperature for the bridge and ground would be the same, and the starting temperature for air would be varied. We additionally, examined the sensitivity of the system to different base conditions.

"Why does *the bridge freeze before the road?"*

Using the principle of conservation of energy, we derived a system of equations to model heat diffusion on both level road and suspended bridge. This model incorporates the scientific principles of conduction and radiation to calculate unit temperatures after successive day-night cycles. The graphical interface allows the user to isolate the temperatures of individual parts of the bridge in order to determine the cause of early freezing.

The model reproduces a real-world situation: material coefficients are based on existing thermal data and the model can be effectively applied to real-world bridges in cold climate areas which are subjected to freezing temperatures during the night. This model has the potential for use by meteorologists, police and transportation safety officials to accurately predict when travel conditions have the potential to become unsafe in certain areas. The diffusion code can be easily adapted to a variety of applications such as cellular osmosis and building efficiency and also the

potential for a multitude of other uses.

Problem Statement

Heat transfer is a broad area of study, thereby requiring the definition of several limitations to narrow the scope of our project. First, heat flow is assumed to be two-dimensional. This assumption means that the ends of the bridge is the same temperature as the center of the bridge. This is the same as assuming that the bridge is infinitely long. Second, the model uses only direct conduction and blackbody radiation, ignoring convection as a variable meaning that cooling by the wind is beyond the scope of our model. Third, the spatial grid has reflective boundary conditions. Meaning that the outermost ground and air in the model have no net heat change.

In this model, heat flow is calculated on a two-dimensional grid experimental and control. The air is represented around the concrete bridge in the experimental model (the bridge) and there is a ground layer below and to the sides of the concrete in the control model (normal road). After this, implementation was simple. There are two primary goals of the diffusion algorithm: (1) calculate how much heat is conducted between cells during each timestep and (2) calculate how much heat escapes due to blackbody radiation at the top layer of the model.

Figure 1: A signpost frequently seen by passing motorists

An explicit step method of adjacent-cell diffusion is used to calculate heat diffusion. The advantages of explicitly calculating the steps is simple math, the code does not require algebra. The disadvantage is that we must use very small time steps to prevent the risk of divergence, i.e. where the temperatures could go out of bounds or exhibit speckling.

Model

1. Spatial Grid

We modeled two cases using the same theoretical stretch of road. The first is a ground road wherein the road is embedded in the ground and air is on top. The second is when the road is completely surrounded by air, as in the example of a suspension bridge.

A. Control Case (Road)

We began with a control case. The road rests on top of a layer of ground representing the typical cross-section of a straight road on even ground. This allowed us to test the heat diffusion code for the convergence and thereby improve the efficiency of our code.

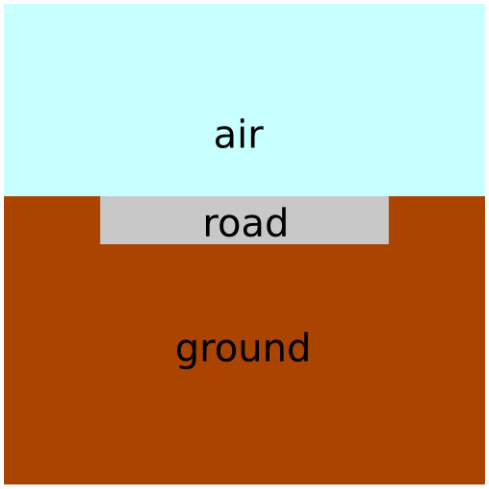

Figure 2: Road on level ground

B. Experimental Case (Bridge)

After the control case was completed, we designed the experimental case. Utilizing the bridge model. Though the ground would have some stabilizing effect, our model assumes that this variable is negligible on a long bridge. The cross-section is taken from the mid-point of the bridge surrounded only by air, with no adjacent ground, thereby accurately modeling a bridge where there is no direct path for heat transfer between the road and the ground.

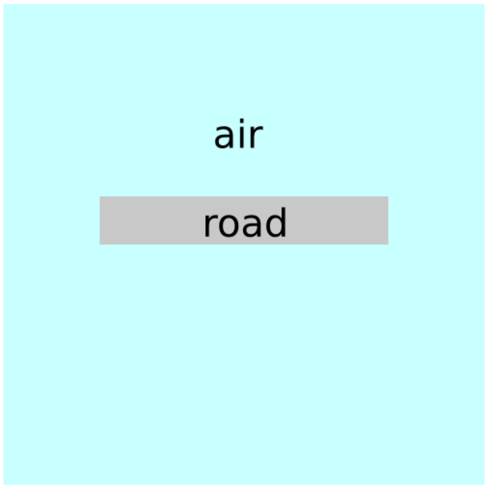

Figure 3: Bridge suspended in air

C. Testing the Experimental Case

The assumption that the bridge is acting as a block of concrete suspended in air is critically important to our experiment. To substantiate this assumption, we tested a preliminary version of the model without blackbody radiation on a different cross-section of the bridge (the bridge cut lengthwise). For a bridge of over fifty meters, the temperature difference due to ground conduction was found to be less than one tenth of a degree. Although the struts of the bridge (Figure 4) cause a change in temperature if they are close to the test cross-section, this change was found to be insignificant when compared to the blackbody radiation from the top of the bridge.

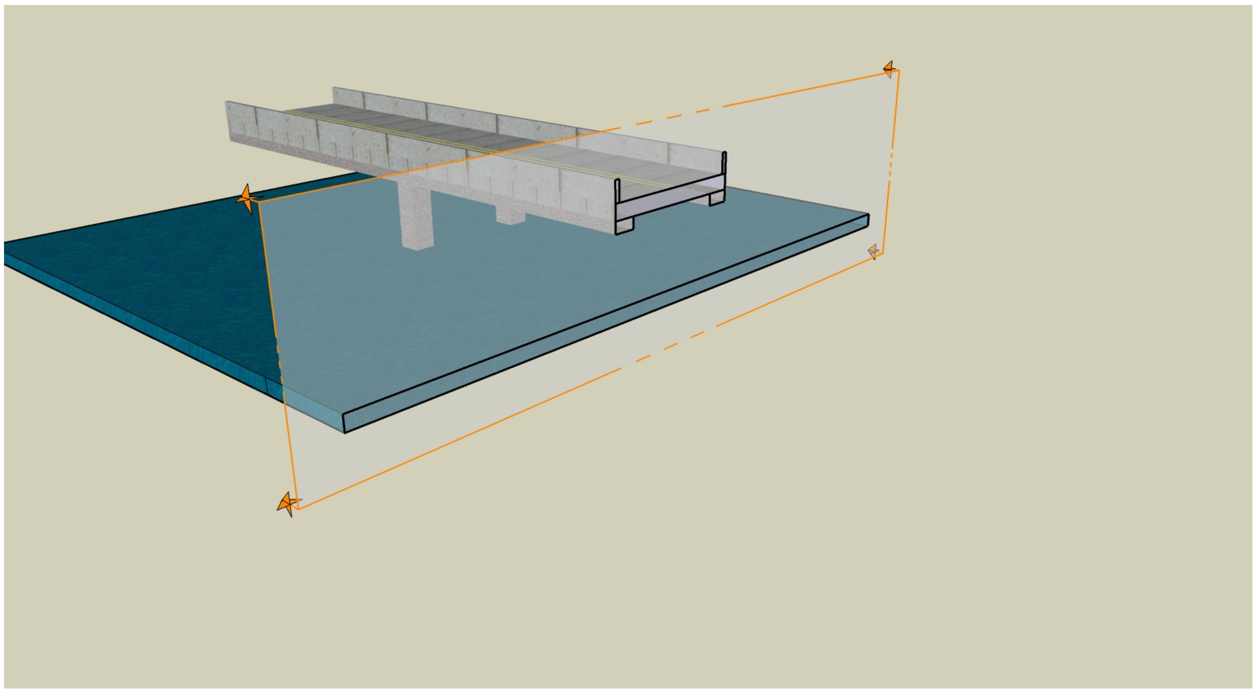

Figure 4: 3D model of a bridge, showing the experimental cross-section (which is perpendicular to the lengthwise cross-section) and the struts that support the bridge

2. Diffusion

The best way to ensure that Newtonian physics is satisfied is to start with the law of conservation of energy. Thus, the net energy of the system is always conserved.

$$\Delta Q = 0$$ *(Equation 1)*

Or, expressed another way:

$$Q_{in} = Q_{out}$$ *(Equation 2)*

Equation 2 will be useful later. Now, we can start with the heat equation:

$$\frac{dU}{dt} = K\nabla^2 U$$ *(Equation 3)*

Let's expand it for three dimensions:

$$\frac{dU}{dt} = K\left(\frac{d^2u}{dx} + \frac{d^2u}{dy} + \frac{d^2u}{dz}\right)$$ *(Equation 4)*

We also know a constant-based value for K, where *k* is thermal conductivity, ρ is density, and C_p is the specific heat capacity.

$$K = \frac{k}{\rho C_p}$$ *(Equation 5)*

Let's look at a segment of our spatial grid

$$\begin{pmatrix} U^t_{n-1,j-1} & U^t_{n-1,j} & U^t_{n-1,j+1} \\ U^t_{n,j-1} & U^t_{n,j} & U^t_{n,j+1} \\ U^t_{n+1,j-1} & U^t_{n+1,j} & U^t_{n+1,j+1} \end{pmatrix}$$ *(Equation 6)*

Note on conventions: The superscript is the time step, and the subscript is the location in the grid.

So the new temperature U at (n,j) will be calculated based on all of the cells adjacent (but not diagonal to it. Let's call each of these individual changes in energy q (which is measured in Watts per second). In order to calculate q, we use our original heat equation and our understanding of the law of conservation of energy.

$$\frac{dQ}{dt} = \sum q \qquad \textit{(Equation 7)}$$

We also have to convert between energy and temperature

$$\frac{dQ}{dt} = \rho V C_p \frac{dU}{dt} \qquad \textit{(Equation 8)}$$

Setting these two equal and expanding, we get:

$$\rho \Delta x \Delta y \Delta z C_p \frac{dU}{dt} = \Delta y \Delta z K \frac{U_{i+1} - U_i}{\Delta x} + \Delta x \Delta z K \frac{U_{j+1} - U_j}{\Delta y} + \Delta x \Delta y K \frac{U_{k+1} - U_k}{\Delta z}$$
$$+ \Delta y \Delta z K \frac{U_i - U_{i+1}}{\Delta x} + \Delta x \Delta z K \frac{U_j - U_{j+1}}{\Delta y} + \Delta x \Delta y K \frac{U_k - U_{k+1}}{\Delta z} \qquad \textit{(Equation 9)}$$

Since we are working in two dimensions, we can assume that that z is dimensionless, which means that

$$\Delta z = \Delta x \Delta y \qquad \textit{(Equation 10)}$$

With this, we get

$$\rho \Delta x \Delta y C_p \frac{dU}{dt} = \Delta y^2 K (U_{i+1} - U_i) + \Delta x^2 K (U_{j+1} - U_j) + \Delta y^2 K (U_i - U_{i+1}) + \Delta x^2 K (U_j - U_{j+1})$$

(Equation 11)

This equation can easily be solved by a computer.

3. Blackbody Radiation

Adding the blackbody radiation term was simple. We added a fixed differential for the blackbody radiation equation at the top-most level of cells in the i+1 direction.

$$q = A\sigma\varepsilon(T^4 - T_0^4)$$ *(Equation 12)*

$\sigma =$ Stephan-Boltzmann constant

$\varepsilon =$ emissivity

We assumed the sink temperature (T_0) was the average temperature of space, which means there are no clouds above our model.

4. Day-Night Cycles

To model a more realistic situation we used a sine wave to model a diurnal (day/night) cycle. The expression:

$$Temperature(time) = Temperature_0 + 10sin(\frac{2\pi * time}{24 * 3600} - \frac{\pi}{3}) \qquad \textit{(Equation 13)}$$

Shows how the temperature, in (K), of the air in the model varies with time (s). The 10 establishes the amplitude of the cycle.

Plus or minus 10 Kelvin was found to be a reasonable change in temperature over the course of a day in Los Alamos, New Mexico.

The next term, inside of the sin function, models where in the day the model currently is. The bottom term is the number of seconds in a day and the numerator is the current time. Pi/3 establishes that the coldest place in the model will be approximately 4:00 AM. The day/night cycles are needed to approximate the blackbody radiation effects on the temperature in the system without actually modeling the blackbody radiation. This makes a more realistic system in the model, modeling only the conservation of energy and the energy lost to space and cannot gain any energy. This allows the model to be far more complex and realistic without the added time of actually modeling the radiation from the sun.

This function is applied to the air in the model so the day/night cycle does not interfere with the diffusion in the other materials. In the real world, the air cools and warms the ground and roads as opposed to the variables. This is highlighted by the fact that the air in the model acts as a temperature sink, so air is the best agent to change temperature of the entire model. The model can only handle blackbody radiation as its mechanism of heat loss. To ensure that other sources of loss and gain were accounted for the day/night cycles, a redundancy check is necessary ensure that the heat lost and gained throughout the day is accurately considered and reflected in a model this complex.

5. Physical Constants Used

Stephan-Boltzmann constant: $5.670 * 10^{-8}$

material	**specific heat ($J * kg^{-1} * K^{-1}$)**	**thermal conductivity ($W * m^{-1} * K^{-1}$)**	**density ($kg * m^{-3}$)**	**emissivities (dimensionless)**
cast concrete, lightweight	1000	0.38	1200	0.92
soil (clay loam)	800	0.36 – 0.69 (median: ~0.50)	1230 – 1590 (median: ~1300)	~0.89
dry air @ 293.15K	1005	0.0257	1.205	~0.30

Sources:

- concrete: “Thermal Properties of Building Materials.” www.bath.ac.uk/~absmaw/BEnv1/properties.pdf
- air: “Air Properties.” *The Engineering Toolbox* http://www.engineeringtoolbox.com/air-properties-d_156.html
- soil: “Soil Thermal Conducitivity.” *Soil Science Society of America Journal*. http://soil.scijournals.org/cgi/content/short/64/4/1285
- emissivities:
  - concrete: http://www.infrared-thermography.com/material-1.htm

Method

1. Design Principles

The reason for stating design principles in a computational science project is the necessity for the code running the model to be elegant as well as powerful. This makes it possible to validate the model, minimize errors and reduce the time spent writing (or rewriting) code. The design principles used are:

•**Simple is better than complex** The model should allow one right way to do things. The algorithm must follow a logical flow and methods must be divided clearly into small tasks.

•**Don't reinvent the wheel** Use existing libraries to create the program. The scope of the coding is the model itself, not the plotting, importing, or array handling.

•**Rapidly prototype** Make the code quickly, minimizing writing time spent while maximizing maintainability. Optimize only when the a module is stable enough to support it.

•**Scale for performance** Design for one processor, and make the program is as efficient as possible. Then scale to hundreds of processors (or GPU cores), ensuring that the problem properly harnesses this power.

2. Language

When considering the language for our model we looked back to our design principles. We chose Python, because it meets goals 1, 2, and 3 “out of the box.” We reached goal 4 using Psyco, a library that enhances computational performance in Python. Pysco enhances the performance of Python by compiling certain things back to essentially what is C code. This allows substantial speed up, many tests by a factor of two. The libraries that are used with Python, such as scipy and numpy, boost speed at scientific tasks and array operations. The combination of Pysco and these libraries allows the team to develop something close in speed to native code, in a about a quarter of the time. These considerations allowed the team to create a model more computationally accurate and more detailed. The model demonstrates extensibility through its Python object structure and efficiency through its use of Psyco.

The computer code is written in the Python computer language and utilizes a Graphical User Interface (GUI) that allows the user to change the variables geometry, boundary conditions, and computational parameters (e.g. time step, node spacing). The code uses the Matplotlib API to produce 2D temperature contour plots which allow the user to see temperature gradients and the coldest place in the model. The Matplotlib API was written to function with the scipy and numpy libraries and allows basic animations “out of the box” plus more complex animation with integration into window management APIs, such as GTK. These careful design considerations will allow the user to experience the maximum amount of benefits without sacrificing traditional speed. The GUI would have been hundreds if not thousands of lines if it was written in C. Python uses less than one hundred lines even with the animation. Without the extensive computational science libraries built on Python these benefits would be unachievable.

3. Explicit Method

The temperature change of an object (e.g. a bridge) is determined by the flow of energy to/from the object as compared to the energy storage potential of the object. In solid materials, heat flows via thermal conduction, and energy is stored as a function of the specific heat of the materials. The goal of this project is to write a computer code that solves the time dependent heat conduction equation as a function of material type, geometry, and boundary conditions. The "finite-difference" method is used to simplify the differential heat transfer equation into explicit calculations (which are a function of the node spacing of the solution). Boundary conditions are applied at the edge nodes as needed.

With the explicit method, we always use the previous time step temperatures to calculate the temperatures for the next time step. This is different from other methods, where temperatures are calculated by using an algebraic solver on an equation that contains both steps. The explicit method allowed us to write a simple program that makes the calculations very quickly.

We used first-order differentiation for the finite difference method. This means that all of the calculations are based on an independent value for the change in temperature in between cells. In order to get the heat transfer, we average the thermal properties of adjacent cells to calculate the new (half-step thermal values). This is then used in the original finite difference equation.

Results

At the conclusion of our project we determined that the bridge does in fact freeze before the road. Because the bridge is surrounded by air. Ground has a higher specific heat than air, which means that concrete encased in ground will be less susceptible to significant changes in temperature. Concrete suspended in air however, has much less tolerance for the energy change at night.

In order to display the results in a visual format. We used the matplotlib API and designed an integrated GUI. This allowed us to show multiple graphs and do real-time refreshing. Our first graph accomplished what we required though it still problems redrawing certain elements of the interface.

Our final GUI allowed us to view all of the results on one screen and it allowed live updating. It demonstrates the progress of the bridge plus a line graph to display the progression of each individual material. This made it easy to determine what happened to each of the different materials as the simulation progressed, facilitating our conclusion for why the bridge freezes first.

Conclusion

The computer model has reproduced the expected "real world" observation, i.e. that the bridge freezes before the road. The code can be used to show whether a bridge will freeze on a given night, and how fast it will freeze given specific user input (e.g. dimensions, material constants, and temperatures). The code also provides two-dimensional contour temperature plots that show how the bridge cools down as a function of time.

Through researching, writing and running code we were able to understand the physics involved. Determining why the bridge freezes before the road requires an understanding of heat conduction and storage in materials. The simplest answer to the question "why does the bridge freeze before the road?" is that the bridge has less stored energy than the road (because of the ground under the road). If heat is removed at the same rate from both the road and the bridge (e.g. the simple case of thermal radiation to space), the bridge will cool down faster and freeze before the road.

The most significant original achievement is that we were able to create a heat diffusion model of a bridge. The code achieved this by creating a self updating contour graph using the API Matplotlib. This allowed us to create a powerful framework for typical heat diffusion. The code incorporates important physical phenomena including blackbody radiation and diffusion by conduction.

As a next step, the model could be extended to be three-dimensional or to compensate for convention due to wind. In the future, the code could be modified to solve a wide range of thermal problems, perhaps providing solutions to some of our nation's most urgent needs, such as energy efficiency and electricity production.

References

Evans, Lawrence C. Partial differential equations. Providence, R.I: American Mathematical Society, 1998.

Hetland, Magnus Lie. Beginning Python: From Novice to Professional (Beginning: From Novice to Professional). New York: Apress, 2005.

Holmes, Mark H. Introduction to Numerical Methods in Differential Equations (Texts in Applied Mathematics). New York: Springer, 2006.

McConnell, Steve. Code Complete, Second Edition. New York: Microsoft P, 2004.

McCrory, Prof. RL. “MAS 545 Numerical methods in Hydrodynamics.” University of Rochester. 1985.

Numpy Doucmentation. 9 Feb. 2005. SciPy. 1 Apr, 2009 <http://numpy.scipy.org/#docs>.

Matplotlib v0.98.5.2 documentation. Sourceforge. 1 Apr, 2009 <http://matplotlib.sourceforge.net/contents.html>.

Siegel, Robert and John R. Howell. Thermal radiation heat transfer. Washington, D.C: Hemisphere Pub. Corp., 1992.

Appendix A: Code

License

Because we believe in free software, we have made the code for this project freely available to anyone who is interested. For convenience, we have appended the most recent revision of the code at the time of this writing. You may access the up-to-date version from the repository at http://code.google.com/p/scc08/. This code is distributed under the GNU General Public License v3. You can view this license at http://www.gnu.org/licenses/gpl.html.

Physical Constants (constants.py)

location: trunk/constants.py
size: 637 bytes
last updated: r147 (April, 2009)

```
class Constants:
    stephan_boltzmann_constant = 1.3806503 * 10 ** -23
    SPECIFIC_HEATS = {
    # all specific heats are in J / kg / K
    'concrete': 1000.,
    'soil': 800.,
    'air': 1005.,
    }
    THERMAL_CONDUCTIVITIES = {
    # all thermal conductivities are in W / m / K
    'concrete': .38,
    'soil': .50,
    'air': .0257,
    }
    DENSITIES = {
    # all densities are in kg / m^3
    'concrete': 1200.,
    'soil': 1300.,
    'air': 1.205,
    }
    EMISSIVITIES = {
    # emissivities are scalars
    'concrete': 0.92,    # http://www.infrared-
thermography.com/material-1.htm
    'soil': 0.89,
    'air': 0.30,
    }
```

Single Cell (cell.py)

location: trunk/cell.py
size: 539 bytes
last updated: r88 (March 23, 2009)

```
from constants import Constants as c
class Cell:
    def __init__(self, material, temperature=300.):
        self.material = material        # string to describe material (try ground, concrete, or air)
        self.temperature = temperature
        (self.density, self.thermal_conductivity, self.specific_heat_capacity, self.emissivity) = \
            (c.DENSITIES[material], c.THERMAL_CONDUCTIVITIES[material], c.SPECIFIC_HEATS[material], c.EMISSIVITIES[material])

    def __str__(self):
        return "<" + str(self.temperature) + ">"
```

2D Grid (grid.py)

location: trunk/grid.py
size: 3.6 KB
last updated: r88 (March 23, 2009)

```python
from numpy import shape as npshape
from constants import Constants as c

class Grid:
    def __init__(self, fields, delta_x, delta_y, delta_t, sink_temperature):
        self.fields = fields
        self.shape = npshape(fields)

        self.iteration = 0
        self.sink_temperature = sink_temperature

        self.delta_x = delta_x
        self.delta_y = delta_y

        self.delta_t = delta_t

        print (delta_x, delta_y, delta_t)

        # Your code has to do the work to create a field that works. It must be a 2D array of Cells

    def __str__(self):
        return str(self.fields)

    def downflux(self, i, j):
        if(j == 0):  # 2000 leagues under the sea
            return 0
        dimensions = self.delta_x / self.delta_y
        theta = self.delta_t / (self.fields[i][j].density * self.delta_x * self.delta_y * self.fields[i][j].specific_heat_capacity)
        thermal_conductivity_avg = (self.fields[i][j].thermal_conductivity + self.fields[i][j-1].thermal_conductivity) / 2.0
        delta_temperature = (self.fields[i][j-1].temperature - self.fields[i][j].temperature)

        return dimensions * theta * thermal_conductivity_avg * delta_temperature

    def upflux(self, i, j):
        if(j == self.shape[1] - 1): # watchin' the stars
            return self.blackbodyflux(i, j)
        dimensions = self.delta_x / self.delta_y
        theta = self.delta_t / (self.fields[i][j].density * self.delta_x * self.delta_y * self.fields[i][j].specific_heat_capacity)
        thermal_conductivity_avg = (self.fields[i][j].thermal_conductivity + self.fields[i][j+1].thermal_conductivity) / 2.0
        delta_temperature = (self.fields[i][j+1].temperature - self.fields[i][j].temperature)

        return dimensions * theta * thermal_conductivity_avg * delta_temperature

    def leftflux(self, i, j):
        if(i == 0): # far left... dude
            return 0
        dimensions = self.delta_y / self.delta_x
        theta = self.delta_t / (self.fields[i][j].density * self.delta_x * self.delta_y * self.fields[i][j].specific_heat_capacity)
        thermal_conductivity_avg = (self.fields[i][j].thermal_conductivity + self.fields[i-1][j].thermal_conductivity) / 2.0
        delta_temperature = (self.fields[i-1][j].temperature - self.fields[i][j].temperature)

        return dimensions * theta * thermal_conductivity_avg * delta_temperature

    def rightflux(self, i, j):
        if(i == self.shape[0] - 1): # far right, sir.
            return 0
        dimensions = self.delta_y / self.delta_x
        theta = self.delta_t / (self.fields[i][j].density * self.delta_x * self.delta_y * self.fields[i][j].specific_heat_capacity)
        thermal_conductivity_avg = (self.fields[i][j].thermal_conductivity + self.fields[i+1][j].thermal_conductivity) / 2.0
        delta_temperature = (self.fields[i+1][j].temperature - self.fields[i][j].temperature)

        return dimensions * theta * thermal_conductivity_avg * delta_temperature

    def blackbodyflux(self, i, j):
        theta = self.delta_t / (self.fields[i][j].density * self.delta_x * self.delta_y * self.fields[i][j].specific_heat_capacity)
        temperature_flux = self.fields[i][j].temperature ** 4 - self.sink_temperature ** 4
        return -1.0 * theta * c.stephan_boltzmann_constant *
```

```
self.delta_x * self.fields[i][j].emissivity * temperature_flux

    def calculate(self, i, j):
        return self.fields[i][j].temperature + self.upflux(i,j) +
self.downflux(i,j) + self.leftflux(i,j) + self.rightflux(i,j) \
            + self.blackbodyflux(i,j)

    def step(self):
        for i in range(self.shape[0]):
            for j in range(self.shape[1]):
                self.fields[i][j].temperature = self.calculate(i,j)
```

Cell → XML Converter (CelltoXML.py)

location: trunk/CelltoXML.py
size: 2.8 KB
last updated: r128 (March 25, 2009)

```python
#! /usr/bin/python

import cell as cell
from xml.dom.minidom import Document
from os.path import exists
import psyco

psyco.full()

def toXML(cellArray, filename = ''):
    # Create the minidom document
    doc = Document()

    # Create the <wml> base element
    wml = doc.createElement("Run1")
    doc.appendChild(wml)
    for i in range (0, len(cellArray)):
        for j in range (0, len(cellArray[i])):

            # Create the main <card> element
            material = ""
            temperature = ""
            density = ""
            thermal_conductivity = ""
            specific_heat_capacity = ""
            emissivity = ""
            material += cellArray[i][j].material
            temperature += "%s"% cellArray[i][j].temperature
            thermal_conductivity += "%s"% cellArray[i][j].thermal_conductivity
            specific_heat_capacity += "%s"% cellArray[i][j].specific_heat_capacity
            emissivity  += "%s"% cellArray[i][j].emissivity
            Node = doc.createElement("Cell")# +"%s"% j)
            wml.appendChild(Node)
            Coor = doc.createElement("Material-" + "%s"% i + "-" + "%s"% j)
            Coor.setAttribute("Type ", material)
            Node.appendChild(Coor)

            Temp = doc.createElement("Temperature-" + "%s"% i + "-" +
"%s"% j)
            Temp.setAttribute("u ", temperature)
            Node.appendChild(Temp)

            Specific = doc.createElement("Specific_Heat-" + "%s"% i + "-" + "%s"% j)
            Specific.setAttribute("c", specific_heat_capacity)
            Node.appendChild(Specific)

            Emissivity = doc.createElement("Emissivity-" + "%s"% i + "-" + "%s"% j)
            Emissivity.setAttribute("e", emissivity)
            Node.appendChild(Emissivity)

            tConductivity = doc.createElement("Thermal_Conductivity-" + "%s"% i + "-" + "%s"% j)
            tConductivity.setAttribute("k", thermal_conductivity)
            Node.appendChild(tConductivity)
    kw = False
    i = 0

    while(kw == False):
        if(filename == ""):
            if (exists("test" + "%s"% i + ".xml") == False):
                out_file = open("test" + "%s"% i + ".xml", "w")
                out_file.write(doc.toprettyxml(indent="  "))
                out_file.close()
                kw = True
            else:
                i += 1
        else:
            if (exists(filename + ".xml") == False):
                    out_file = open(filename + ".xml", "w")
                    out_file.write(doc.toprettyxml(indent="  "))
                    out_file.close()
            elif (exists(filename + "%s"% i + ".xml") == False):
                    out_file = open(filename + "%s"% i +  ".xml", "w")
                    out_file.write(doc.toprettyxml(indent="  "))
                    out_file.close()
                    kw = True
            else:
                    i += 1
```

XML → Cell Converter (XMLtoCell.py)

location: trunk/XMLtoCell.py
size: 1.5 KB
last updated: r128 (March 25, 2009)

```
#! /usr/bin/python

from xml.dom import minidom
import urllib, sys
import psyco

psyco.full()

class readXML():
    def _init_(self):

    def data(self):
        return self.data
    def read(self):
        Count = 0
        Data = []

        for i in range (0, len(xmldoc.getElementsByTagName("Cell"))):
            for j in range (0, 15):
                data_list = xmldoc.getElementsByTagName("Material-" + "%s"% i + "-" + "%s"% j)#  "[" + "%s"% i + "]" + "[" + "%s"% j + "]")
                for data_element in data_list:
                    type = data_element.getAttribute("Type")

                data_list = xmldoc.getElementsByTagName("Temperature-" + "%s"% i + "-" + "%s"% j)
                for data_element in data_list:
                    t = data_element.getAttribute("u")

                data_list = xmldoc.getElementsByTagName("Specific_Heat-" + "%s"% i + "-" + "%s"% j)
                for data_element in data_list:
                    c = data_element.getAttribute("c")

                data_list = xmldoc.getElementsByTagName("Emissivity-" + "%s"% i + "-" + "%s"% j)
                for data_element in data_list:
                    e = data_element.getAttribute("e")

                data_list = xmldoc.getElementsByTagName("Thermal_Conductivity-" + "%s"% i + "-" +
"%s"% j)
                for data_element in data_list:
                    k = data_element.getAttribute("k")

                Data.append([str(type),float(t),float(c),float(e),float
(k)])
```

Driver (driver.py)

location: trunk/driver.py
size: 4.6 KB
last updated: r129 (March 26, 2009)

```
import psyco
import numpy as np
from cell import Cell
from grid import Grid
import copy as cp
#import matplotlib.pyplot as plt
#from pylab import *
import CelltoXML as xml
import time
import math

psyco.full()

class Driver():

    def __init__(self, deltax, deltay, deltat, case, bridge,
dimensions, soilTemp, airTemp, concreteTemp):
        """
        Creates a grid to run, initializes all of the cells
        with the data, bridge stores (startx, starty, endx, endy)
        dimensions are the (height, and width of the bridge
        case 1: air
        case 2: air and ground
        """

        #initialize class variables
        self.deltax = deltax
        self.deltay = deltay
        self.deltat = deltat
        self.bridge = bridge
        #initialize useful variables
        self.dimensions = dimensions
        self.ground = Cell('soil', soilTemp)
        self.air = Cell('air', airTemp)
        self.concrete = Cell('concrete', concreteTemp)

        self.fields = []
        if case == 1:
            self.fillAir()
        else:
            self.fillGround()

        self.myGrid = Grid(self.fields, self.deltax, self.deltay, self.deltat, 2.725)


    def run(self, iterations):
        """
        Runs the grid for a specified number of iterations
        """
        #print self.myGrid
        #print self.myGrid.fields

        for i in range(iterations):
            self.myGrid.step()


    def fillAir(self):
        """
        Initializes the cells in the grid, with concrete surronded by air
        """

        for i in range(self.dimensions[1]):
            for j in range(self.dimensions[0]):
                if ((i >= self.bridge[0] and i <= self.bridge[2]) \
                and (j >= self.bridge[1] and j <= self.bridge[3])):
                    self.fields.append(cp.deepcopy(self.concrete))
                else:
                    self.fields.append(cp.deepcopy(self.air))

        self.fields = np.reshape(self.fields, (self.dimensions[0],self.dimensions[1]))

    def fillGround(self):
        """
        Initializes the cells in the grid, with concrete surronded by ground
        """

        for i in range(self.dimensions[1]):
            for j in range(self.dimensions[0]):
                if (i >= self.bridge[0] and i <= self.bridge[2]):
                    if(j >= self.bridge[1] and j <= self.bridge[3]):
                        self.fields.append(cp.deepcopy(self.concrete))
                    else:
                        self.fields.append(cp.deepcopy(self.ground))

                elif (i > self.bridge[2]):
                    self.fields.append(cp.deepcopy(self.ground))

                else:
                    self.fields.append(cp.deepcopy(self.air))


        self.fields = np.reshape(self.fields, (self.dimensions[1],self.dimensions[0]))

    def changeAir(self, amount):
        for i in self.fields:
            for j in i:
                if j.material == 'air':
                    j.temperature + amount

    def dayNight(self, time):
        amount = 10 * math.sin(  (2 * math.pi * time) / ( 86400 ) - math.pi / 3 )
        self.changeAir(amount)

    def toXML(self, numIter, name): #, interval, cellArray):
        for i in range(numIter):
            self.myGrid.step()
            self.dayNight(self.deltat * i)
            xml.toXML(self.myGrid.fields, name) #enter name

    def checkCement(self):
        for i in self.fields:
            for j in i:
                if(j.material == 'concrete'):
                    if(j.temperature <= 273.13):
                        return False
                    else:
                        return True

    def tillFrozen(self, increment, name):
        """runs the model until the first cement is frozen"""
        i = 0
        while self.checkCement():
            self.myGrid.step()
            self.dayNight(self.deltat * i)
            i += increment
            xml.toXML(self.myGrid.fields, name) #enter name
        return i

    def rapidChange(self, increment, name, when, amount):
        """runs the model until the first cement is frozen"""
        i = 0
        happened = False
        while self.checkCement():
            self.myGrid.step()
```

```python
            self.dayNight(self.deltat * i)
            if(happened == False and i >= when):
                self.changeAir(amount)
                happened = True
            i += increment
            xml.toXML(self.myGrid.fields, name) #enter name
        return i

if __name__ == "__main__":
    dr = Driver(.5, .5, 0.0004, 2, (7,7,9,9), (16,16), 300,300,300) # don't use time steps over 4 * 10 ** -4

    dr.toXML(2, "filename")
```

Graphical User Interface (gui.py)

location: trunk/gui.py
size: 3.1 KB
last updated: r148 (April 1, 2009)

```
import psyco
import numpy as np
from cell import Cell
from grid import Grid
import copy as cp
from pylab import *
import time
from driver import Driver

psyco.full()


def updateData(dr):
    data = []
    temp = dr.myGrid.fields
    for i in range(len(temp)):
        data.append([])
        for j in range(len(temp[i])):
            data[i].append(temp[i][j].temperature)
    return data

def getMats(dr):
    mats = []
    temp = dr.myGrid.fields
    for i in range(len(temp)):
        mats.append([])
        for j in range(len(temp[i])):
            mats[i].append(temp[i][j].density)
    return mats

def setLabels(type):
    xlabel("Meters")
    ylabel("Meters")
    if (type == 2):
        title("A road")
    else:
        title("A bridge")

def setUserInput():
    str = raw_input("What do you want: Ground or Air? (The default is
air) \n")
```

```
        str
        if(str == "Ground" or str == "ground"):
            type = 2
        else:
            type = 1
        return type

    def getAirPts(dr):
        temp = dr.myGrid.fields
        for i in range(len(temp)):
            for j in range(len(temp[i])):
                if(temp[i][j].material == 'air'):
                    return temp[i][j].temperature

    def getBridgePts(dr):
        temp = dr.myGrid.fields
        for i in range(len(temp)):
            for j in range(len(temp[i])):
                if(temp[i][j].material == 'cement'):
                    return temp[i][j].temperature

    def getGroundPts(dr, type):
        if (type != 2):
            return None
        else:
            temp = dr.myGrid.fields
            for i in range(len(temp)):
                for j in range(len(temp[i])):
                    if(temp[i][j].material == 'ground'):
                        return temp[i][j].temperature
    def run():
        type = setUserInput()
        dr = Driver(.5, .5, 0.0004, type, (7,7,9,9), (16,16), 300,300,300)
    # don't use time steps over 4 * 10 ** -4
        current = updateData(dr)
        gmats = getMats(dr)
        countsteps = 0
        ion()

        setLabels(type)
        airdata = []
        grounddata = []
        bridgedata = []
        airdata.append(getAirPts(dr))
        grounddata.append(getGroundPts(dr, type))
        bridgedata.append(getBridgePts(dr))

        datapts = plot(airdata, 'g+', grounddata, 'b+')#, bridgedata, 'r*')

        subplot(211)
```

```python
    airdata.append(getAirPts(dr))
    grounddata.append(getGroundPts(dr, type))
    bridgedata.append(getBridgePts(dr))
    CS = contour(gmats, type+1)
    subplot(212)
    grid(True)
    xlabel("The timestep is: " + str(countsteps*.004))
    ylabel("Temperature")
    datapts = plot(airdata, 'gD-', grounddata, 'bs-') #, bridgedata, 'r*')
    for i in range(1,20000):
        dr.run(10)
        current = updateData(dr)
#        clf()
        airdata.append(getAirPts(dr))
        grounddata.append(getGroundPts(dr, type))
        bridgedata.append(getBridgePts(dr))
        subplot(211)
        CSF = contourf(current)
        CD = colorbar(CSF, extend ='both')
        subplot(212)
        datapts  = plot(airdata, 'gD-', grounddata, 'bs-')#, bridgedata, 'r*')
        xlabel("The timestep is: " + str(countsteps*.004))
        countsteps += 10
#        show()
#        draw()


if __name__ == "__main__":
    run()
```

Automatic Runner (jake.py)

location: trunk/jake.py
size: 6.6 KB
last updated: r125 (March 25, 2009)

```
import copy as cp
import numpy as np
from driver import Driver

savedir = "output/"

#                         deltax, deltay, deltat, (concrete locs), (dimensions), soil, air, concrete
coldfrontroad = Driver(.5, .5, 0.0004, 2, (56,56,72,72), (128,128), 300, 300, 300)
end = coldfrontroad.rapidChange(10, "coldfrontroad1Temp", 10000, -30)
print "running the first"
f = open(savedir + "coldfrontroad1Temp.data")
f.write(end + "\n" "for regular, temperature 300,300,300")
f.close()
print "done"


coldfrontroad = Driver(.5, .5, 0.0004, 2, (56,56,72,72), (128,128), 330, 340, 320)
end = coldfrontroad.rapidChange(10, "coldfrontroad2Temp", 10000, -30)
print "running the first"
a = open(savedir + "coldfrontroad2Temp.data")
a.write(end + "\n" "for regular, temperature 330,340,320" )
a.close()
print "done"


coldfrontroad = Driver(.5, .5, 0.0004, 2, (56,56,72,72), (128,128), 315, 360, 273)
end = coldfrontroad.rapidChange(10, "coldfrontroad3Temp", 10000, -30)
print "running the first"
b = open(savedir + "coldfrontroad3Temp.data")
b.write(end + "\n" "for regular, temperature 315,360,273")
b.close()
print "done"
##########################################################################################################################

largeboundaryroad = Driver(.5, .5, 0.0004, 2, (56,56,72,72), (128,128), 300, 300, 300)
```

```
end = coldfrontroad.tillFrozen(10, "largeboundaryroad1Temp")
print "running the first"
c = open(savedir + "largeboundaryroad1Tempdata")
c.write(end + "\n" "for (56,56,72,72) - (128,128)  temperature 300,300,300")
c.close()
print "done"



coldfrontroad = Driver(.5, .5, 0.0004, 2, (56,56,72,72), (128,128), 330, 340, 320)
end = coldfrontroad.tillFrozen(10, "largeboundaryroad2Temp")
print "running the first"
d = open(savedir + "largeboundaryroad2Temp.data")
d.write(end + "\n" "for (56,56,72,72) - (128,128)  temperature 330,340,320")
d.close()
print "done"




coldfrontroad = Driver(.5, .5, 0.0004, 2, (56,56,72,72), (128,128), 315, 360, 273)
end = coldfrontroad.tillFrozen(10, "largeboundaryroad3Temp")
print "running the first"
e = open(savedir + "largeboundaryroad3Temp.data")
e.write(end + "\n" + "for (56,56,72,72) - (128,128) temperature 315,360,273")
e.close()
print "done"

#####################################################################################################################################

coldfrontroad = Driver(.5, .5, 0.0004, 2, (20,20,108,108), (128,128), 300, 300, 300)
end = coldfrontroad.tillFrozen(10, "largeroad1Temp")
print "running the first"
g = open(savedir + "largeroad1Temp.data")
g.write(end + "\n" + "for (20,20,108,108) - (128,128) temperature 300,300,300")
g.close()
print "done"


coldfrontroad = Driver(.5, .5, 0.0004, 2, (20,20,108,108), (128,128), 330, 340, 320)
end = coldfrontroad.tillFrozen(10, "largeroad2Temp")
```

```
print "running the first"
h = open(savedir + "largeroad2Temp.data")
h.write(end + "\n" + "for (20,20,108,108) - (128,128) temperature 
330,340,320")
h.close()
print "done"


coldfrontroad = Driver(.5, .5, 0.0004, 2, (20,20,108,108), (128,128), 
315, 360, 273)
end = coldfrontroad.tillFrozen(10, "largeroad1Temp")
print "running the first"
i = open(savedir + "largeroad1Temp.data", )
i.write(end + "\n" + "for (20,20,108,108) - (128,128) temperature 
315,360,273")
i.close()
print "done"

##########################################################################
############################################################

import copy as cp
import numpy as np
from driver import Driver

#                          deltax, deltay, deltat, what, (concrete locs), 
(dimensions), soil, air, concrete
coldfrontbridge = Driver(.5, .5, 0.0004, 1, (56,56,72,72), (128,128), 
300, 300, 300)
end = coldfrontbridge.rapidChange(10, "coldfrontbridge1Temp", 10000, 
-30)
print "running the first"
f = open(savedir + "coldfrontbridge1Temp.data")
f.write(end + "\n" "for regular, temperature 300,300,300")
f.close()
print "done"


coldfrontbridge = Driver(.5, .5, 0.0004, 1, (56,56,72,72), (128,128), 
330, 340, 320)
end = coldfrontbridge.rapidChange(10, "coldfrontbridge2Temp", 10000, 
-30)
print "running the first"
a = open(savedir + "coldfrontbridge2Temp.data")
a.write(end + "\n" "for regular, temperature 330,340,320" )
a.close()
print "done"


coldfrontbridge = Driver(.5, .5, 0.0004, 1, (56,56,72,72), (128,128), 
315, 360, 273)
end = coldfrontbridge.rapidChange(10, "coldfrontbridge3Temp", 10000,
-30)
print "running the first"
b = open(savedir + "coldfrontbridge3Temp.data")
b.write(end + "\n" "for regular, temperature 315,360,273")
b.close()
print "done"
########################################################################
########################################################

largeboundarybridge = Driver(.5, .5, 0.0004, 1, (56,56,72,72),
(128,128), 300, 300, 300)
end = coldfrontbridge.tillFrozen(10, "largeboundarybridge1Temp")
print "running the first"
c = open(savedir + "largeboundarybridge1Tempdata")
c.write(end + "\n" "for (56,56,72,72) - (128,128)  temperature
300,300,300")
c.close()
print "done"



coldfrontbridge = Driver(.5, .5, 0.0004, 1, (56,56,72,72), (128,128),
330, 340, 320)
end = coldfrontbridge.tillFrozen(10, "largeboundarybridge2Temp")
print "running the first"
d = open(savedir + "largeboundarybridge2Temp.data")
d.write(end + "\n" "for (56,56,72,72) - (128,128)  temperature
330,340,320")
d.close()
print "done"




coldfrontbridge = Driver(.5, .5, 0.0004, 1, (56,56,72,72), (128,128),
315, 360, 273)
end = coldfrontbridge.tillFrozen(10, "largeboundarybridge3Temp")
print "running the first"
e = open(savedir + "largeboundarybridge3Temp.data")
e.write(end + "\n" + "for (56,56,72,72) - (128,128) temperature
315,360,273")
e.close()
print "done"

########################################################################
#########################################################

coldfrontbridge = Driver(.5, .5, 0.0004, 1, (20,20,108,108), (128,128),
300, 300, 300)
end = coldfrontbridge.tillFrozen(10, "largebridge1Temp")
print "running the first"
g = open(savedir + "largebridge1Temp.data")
g.write(end + "\n" + "for (20,20,108,108) - (128,128) temperature
300,300,300")
g.close()
print "done"


coldfrontbridge = Driver(.5, .5, 0.0004, 1, (20,20,108,108), (128,128),
330, 340, 320)
end = coldfrontbridge.tillFrozen(10, "largebridge2Temp")
print "running the first"
h = open(savedir + "largebridge2Temp.data")
h.write(end + "\n" + "for (20,20,108,108) - (128,128) temperature
330,340,320")
h.close()
print "done"


coldfrontbridge = Driver(.5, .5, 0.0004, 1, (20,20,108,108), (128,128),
315, 360, 273)
end = coldfrontbridge.tillFrozen(10, "largebridge1Temp")
print "running the first"
i = open(savedir + "largebridge1Temp.data", )
i.write(end + "\n" + "for (20,20,108,108) - (128,128) temperature
315,360,273")
i.close()
print "done"
```