There are essentially three main parts to the CosmoTransitions package. First, there is code to find the critical bubble associated with a phase transition in both one and multiple field dimensions. This happens in the files `tunneling1D.py` and `pathDeformation.py`. Second, the file `transitionFinder.py` contains code that analyzes phases and phase transitions of arbitrary functions V(phi, T). Determining the location of the minima (using the functions `traceMinimum()` and `traceMultiMin()`) and finding the critical temperature(s) are completely independent of the tunneling problem, whereas finding the amount of supercooling and the precise nucleation temperature are not. Finally, the modules `generic_potential.py` and `finiteT.py` make it easy to define a realistic particle model and then calculate its phase transition.

The following give overviews of each of the modules. For more detail, see comments in the code or type `help(<module.function_name>)`.

`tunneling1D.py`:

This is a self-contained module that will solve for the bounce solution in a one-dimensional potential. It mainly consists of the class "`bubbleProfile`", which wraps the various functions together. As a minimum, the class requires the potential function as an input as well as the locations of the true and metastable vacua.

The main class method that the user would call is the "`findProfile`" method, which finds the profile by the standard overshoot/undershoot method. This has no required inputs, but the initial guess as well as the various tolerances may be set by the user. The method outputs the bubble profile: an array of radial coordinates r, the field values along the radius of the bubble phi(r), and the derivative dphi/dr.

In order to account for both thin- and thick-walled solutions, I use the following parameter to set the initial conditions at the center of the bubble:

$$x = \log\left(\frac{\phi_F - \phi_T}{\phi(r=0) - \phi_T}\right)$$
$$\longrightarrow \phi(r=0) = \phi_T + e^{-x}(\phi_F - \phi_T)$$

where phi_F and phi_T and the false (metastable) and true (stable) vacuum states, respectively, and the boundary conditions are such that phi(r-->infty) = phi_F. When x is very large, the center of the bubble sits very close to the true vacuum and the solution will be very thinly walled. Conversely, if x is small the center will be very close to the false vacuum and be thick-walled. The trick then is to find the value of x such that phi(r-->infty) = phi_F.

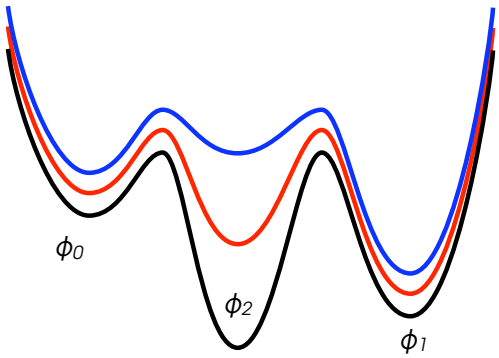
When integrating a thin-walled bubble, we would like to start the integration at the wall boundary instead of near r=0. We can do this by approximating the potential near phi_T as

$$V(\phi) = \tfrac{1}{2}\left.\frac{d^2V}{d\phi^2}\right|_{\phi=\phi_T}(\phi - \phi_T)^2$$

and solving for phi(r) exactly (using Bessel functions). Then we simply find the value of r such that phi(r) starts to deviate significantly from phi_T and use that as the starting point for integration. (Note that in the code I use the same technique for the thick-walled case too, even though there the potential is not well-approximated by the single quadratic term. However, this doesn't really matter since in those cases the integration starts very close to r=0 where phi(r) and dphi/dr are known.)

If there are multiple minima in the potential, then the tunneling behavior can be much more complicated. For instance, consider what happens when we try to tunnel from the minimum at phi_0 to phi_1 with some intermediate minimum phi_2 in the way. If phi_2 > phi_0 > phi_1 (blue line), then the general behavior of the solution is unchanged: the center of a critical bubble will be somewhere near phi_1, and outside of the bubble we be at phi_0. If phi_0 > phi_1 > phi_2, the behavior is still fairly regular. This time the center of the bubble will be near phi_2, and the minimum at phi_1 will be ignored completely since it is not the true vacuum. The overshoot/undershoot method will solve this correctly. The solution can be more complicated if phi_0 > phi_2 > phi_1. It could be that there is no

solution to the equations of motion that tunnels directly from phi_0 to phi_1, and instead the system must first tunnel to phi_2 and then tunnel to phi_1 via a second set of bubbles. Even if direct tunneling is possible, it may be that the tunneling rate is faster if it proceeds in two steps. I do not think that it is possible analytically to see which is the correct procedure for any given potential, although the answer is clear in certain limits. (In the limit phi_0 = phi_2, tunneling must happen all at once since the tunneling rate is zero between degenerate vacua. In the limit phi_2 = phi_1, the tunneling must happen in two steps because any solution that moves from phi_1 to phi_2 will be thin-walled and will invariably over-shoot the minimum at phi_0.) When my code encounters this situation, it may or may not find a solution with the correct boundary conditions, and the center of the bubble may be either at phi_1 or at phi_2.



pathDeformation.py

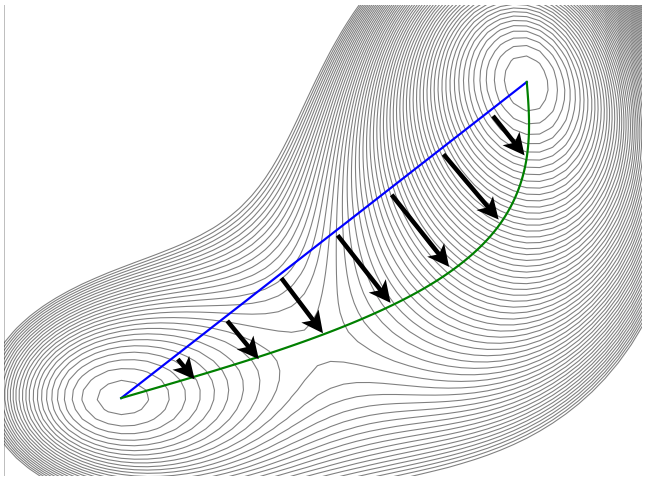
This file started off just containing the code that deforms the one-dimensional potential to fit multiple dimensions, but then I added a couple of classes that find the transition temperature as well.

class Deformation

I solve for the bounce solution in multiple dimensions by solving for the equations of motion along a particular (one-dimensional) path, and then deforming that path so that it matches the equations of motion in all directions. For example, consider the following potential

$$V(\phi_x, \phi_y) = (1.6\phi_x^2 + \phi_y^2)[1.6(\phi_x - 1)^2 + (\phi_y - 1)^2 - .26],$$

which when plotted looks like



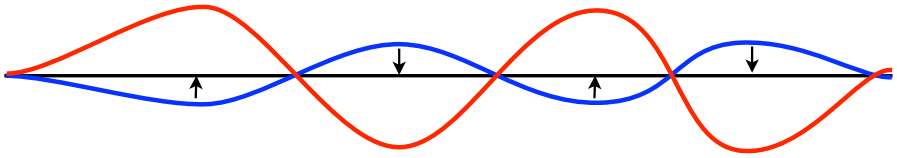
Thinking about the problem as a classical particle moving in a potential (really an inverted potential), the path is the just position (phi) of the particle as a function of time (taking the place of the radial coordinate). At first we confine the particle to move along the linear track (blue line), ignoring any forces perpendicular to this line, and solve for the motion of the particle with the appropriate boundary conditions. Then we move the track step-by-step in the direction of the normal force it

exerts upon the particle, until the procedure converges (green line). This process should generally be repeated a few times (solve for the motion, deform the path, repeat) until both steps converge together.

All path deformation is done using splines. That is, the entire path is broken into some relatively small number of path segments (of order 10), and each segment has its own spline basis function which looks like a bell curve centered on the segment (`pathDeformation.Nbspl` returns these functions). The path is then given by the sum of the basis functions times some vector of coefficients. During each deformation step, the spline coefficients are the important things that change. This avoids the numerical noise that one would see if one tried to do the deformation on the points individually.

To use an instance of Deformation, first initialize it with the one-dimensional solution (phi and dphi/dr — it helps if the points phi are evenly spaced), the multi-dimensional potential, it's gradient function, and the number and degree of spline basis functions. Then, call the `deformPath` method. At each step, this will call the `step` method which will move the spline a small amount in the direction of the normal. The method converges once the magnitude of the forces perpendicular to the path are much smaller than those parallel to it (`fRatio2` is small).

The big trick is to not make the steps too big. Consider the following path in a flat potential:



Since the potential is flat the correct path is just a straight line (black). If we add a small wobble (blue line), then, going back to the classical picture, the normal force that the track exerts on a moving particle will tend to push it back to a straight line. However, if we go too far in that direction we could just make matters worse (red line). The `step` method checks for such behavior, and tries to reverse it if necessary.

`class fullTunneling`:

This class finds both the one-dimensional solution and deforms it. One only needs to initialize it and run the run method. The one-dimensional profile is given by `fullTunneling.lastProfile`, and the points along the path are given by `fullTunneling.phi`. One can find the action *S* with the method `fullTunneling.findAction`. Hopefully the rest is fairly self-explanatory.

`criticalTunneling()` and `class criticalTunneling_class`:

This class finds the critical temperature for thermal tunneling between two different phases. It does this simply by finding the action from a `fullTunneling` instance at different temperatures and seeing when the temperature satisfies the tunneling criterium (generally S/T ~ 140). Note that the minima as a function of temperature should be found with something like `transitionFinder` (see below).

`transitionFinder.py`:

Finding the minima of an effective potential as a function of temperature is a non-trivial task. One could simply run a minimization routine to find all of the different minima at different temperatures (which would be difficult by itself), but then there is no guarantee that the minimum at one temperature corresponds to the same phase as the minimum at another temperature, and it would be very difficult to resolve the points at which phases develop and disappear. Therefore, I wrote this module to trace a phase's vev with respect to temperature and to distinguish between different phases.

`traceMinimum()`:

This is the primary function of the module, although the user probably won't call it directly. It takes as input the function `f()` and its starting vev and temperature, and then traces the vev as a

function of temperature until it either reaches a predefined stopping point or the phase disappears. It returns arrays of vevs and temperatures.

In one dimension, the derivative of the minimum with respect to the parameter t is, analytically,

$$\frac{x_{min}}{dt} = -\left(\frac{\partial^2 f}{\partial x \partial t}\right) / \left(\frac{\partial^2 f}{\partial x^2}\right).$$

In multiple dimensions, it changes to

$$\frac{(x_{min})_i}{dt} = -(M^{-1})_{ij}\left(\frac{\partial^2 f}{\partial x_j \partial t}\right);\ M_{ij} = \left(\frac{\partial^2 f}{\partial x_i \partial x_j}\right).$$

The code essentially takes a series of steps using the above equations while running a minimization routine at each step. When the derivative and the minimization don't agree with each other, then we know that the phase has hit a transition.

`traceMultiMin()`:

This routine essentially does the same thing as `traceMinimum()`, except that it doesn't stop tracing once it hits a phase transition. Instead, it will move on to the next phase and trace that as well. By using this, the one can find both the cold and hot phase (as well as any intermediate phases) starting from just the zero temperature phase. It returns a list of phases, with each object in the list containing an array of minima, of temperatures, and of derivatives, as well as spline coefficients that fit the minima.

`removeRedundantPhases()`:

Often times, especially if there are multiple zero-temperature minima, the `traceMultiMin()` function will result in multiple recordings of the same phase. This removes them.

`findTransitionRegions()`:

With the different phases in hand, one can call `findTransitionRegions()` to determine where the phases actually overlap and in which direction a transition will occur.

`findFullTransitions()`:

Finally, we are ready to analyze the phase transition structure of our potential. From the above functions, we know what the phase structure is and we know where the transitions are possible. This looks at each phase transition region, and sees where the bubble nucleation actually takes place. It outputs a list of transitions, starting with the highest temperature and working down to the low-T phase. Things are somewhat complicated by the possibility of multiple phases lining up next to each other, which could give the program a headache when trying to tunnel around the phase in the middle. To deal with this, the function only tries to tunnel to the nearest phase when there are several in a row (within 'overlap' degrees of each other). Note that if there are multiple low-T minima, there is no guarantee that the final state will be stable rather metastable.

`generic_potential.py`:

In order to make it as easy as possible to implement new models, I've designed an abstract class that contains all of the code that's common to different possible models.

`class generic_potential`:

The first couple of methods are for initialization. `__init__()` is the standard initialization function (called whenever creating an instance via `generic_potential()`. The second `init()` method is called by `__init__()` to implement anything specific to the subclass, and should be overridden.

The next few functions define the potential. `V0()`, `boson_massSq()` and `fermion_massSq()` define the tree-level potential and the particle spectrum to be used in the model. These absolutely must be overridden by subclasses. Other methods calculate the one-loop corrections from the mass spectrum and the derivatives of the potential. These can be left as-is unless you want to do some

non-standard one-loop potential (i.e., using the Hamiltonian formalism or cutoff regularization).

The only minimization function that should need to be overridden is `approxZeroTMin()`, which should return the approximate minima derived from theory. The functions `getPhases()` and `calcTcTrans()` analyze the phase structure of the theory, and find where there are possible transitions between the phases. The function calcFullTrans() finds the actual phase transition temperature(s) by solving the Euclidean equations of motion and finding the bubble nucleation temperature.

The remaining functions in the module are helpful for analyzing many instances `generic_potential` subclasses. `funcOnModels()` allows one to make a multidimensional numpy array out of an array of instances, and `linkTransitions()` matches up the phase transitions of instances that are next to each other in parameter space so that they may be more easily plotted.

`finiteT.py`:

This module contains a variety of functions for calculating the finite temperature contribution to the one-loop effective potential. When importing the module, it will look for the files finiteT_b.dat and finiteT_f.dat to use for making splines. If not found, it will create them by solving the integrals exactly.