Numerically solve linear differential equations using difEQ

difEQ numerically solves linear differential equations for given initial conditions. That is difEQ solves the following system of equations for the vector $\vec{r} = < x, y, z >$

$$\frac{d^2\vec{r}}{dt^2} + \vec{Q} \cdot \vec{r} + \vec{P} \cdot \vec{r} = \vec{F}$$

$$\overrightarrow{r_0} = < x_0,\ y_0,\ z_0 >$$

$$\frac{d\overrightarrow{r_0}}{dt} = < x'_0,\ y'_0,\ z'_0 >$$

In order to specify a particular initial value problem the user has to provide the three functions *Q, P*, and *F* which specify the equation as well as $\overrightarrow{r_0}$ and $\frac{d\overrightarrow{r_0}}{dt}$ which are the initial conditions.

difEQ numerically approximates the solution to the initial value problem (IVP) by using the provided differential equation (DEQ) and the initial conditions to compute $\frac{d^2\vec{r}}{dt^2}$. Once it has $\frac{d^2\vec{r}}{dt^2}$ it can then Taylor expand $\vec{r}$ about the current time and use this to estimate $\vec{r}$ and $\frac{d\vec{r}}{dt}$ at Δt in the future. Then it recalculates r'' based on the new values of $\vec{r}$ and $\frac{d\vec{r}}{dt}$ and the process over again. This process of estimating $\vec{r}$ and $\frac{d\vec{r}}{dt}$, then using those values to solve for $\frac{d^2\vec{r}}{dt^2}$ is called a frame. If we let $\overrightarrow{r_n}$ be the $\vec{r}$ vector in frame *n*, and Δt the time between frames then we can write down a recursive formula for $\overrightarrow{r_{n+1}}$

$$\overrightarrow{r_{n+1}} = \frac{1}{2}\ \frac{d^2\overrightarrow{r_n}}{dt^2}\Delta t^2 + \frac{d\overrightarrow{r_n}}{dt}\ \Delta t + \overrightarrow{r_n}$$

$$\frac{d\overrightarrow{r_{n+1}}}{dt} = \frac{d^2\overrightarrow{r_n}}{dt^2}\ \Delta t + \frac{d\overrightarrow{r_n}}{dt}$$

It is easy to see that the initial conditions $\overrightarrow{r_0}$ and $\frac{d\overrightarrow{r_0}}{dt}$ are constants and only referenced during the very first frame. In contrast *Q, R, F* are not constants and are called on every frame. In order to be more accurate the number of frames is usually quite large (easily on the order of 10^9) it is imperative that calls to the modifiers *Q, P, F* be quite fast. For this reason in order to change the modifiers you must recompile the entire difEQ.c source code after changing the functions. Making this easier is an ongoing project.

Compiling and running:

difEQ was written using GCC in c99 and Anaconda python on Windows 7, you will need to have these installed in order to run difEQ. If you install Anaconda in a location other than the default *C:\Anaconda* you will need to change the system call at the bottom of difEQ to the correct path. While not tested, difEQ should work on Linux by simply changing all the path arguments to their correct location.

I have included a very primitive version of make by which to compile *difEQ.c*. It will take any number of command line arguments and simply copy those over to its call to gcc. I HIGHLY recommend compiling with the *-O3* or *-Ofast* flags, as these decrease run time by a very significant factor (around 5x in my experience). To compile *make.c* itself you should use the command "*> gcc make.c -o make -std=c99*".

As mentioned above you will have to recompile *difEQ.c* each time you change any of the modifiers, however the initial conditions can be fed in as file using the command "*> difEQ filepath*" where *filepath* is the location of the input file. Input files follow the following format:

t_i, t_f, deltaT, storeFrac
x_0, y_0, z_0
x'_0, y'_0, z'_0

The following characters are ignored and may be added in to the input file without issue: "<>," as well as whitespace and tabs. If there are no arguments to difEQ or there is an error in reading the input file, then the program will revert to preset initial conditions set at compile time. Once running, difEQ will simulate frames and store one in every storeFrac of them in memory. When it's finished computing the stored values are written to a file and then read into the graphUtility.py program.

Interacting with Python and conditions:

Upon startup graphUtility.py loads the data from the file specified by its command line argument into *t_vals[], x_vals[], y_vals[], and z_vals[]*. After this has happened you are free to enter whatever you want in the interactive session. You could load a separate data file using *loadFile()*, you could graph the current data using *showGraph()*, you could find the $\vec{r}$ for a given time using *r(),* or you can find the set of r vectors that satisfy some condition using *print_cnd().*

One of the primary uses for the interactive shell is to define conditions on the fly and find where they are satisfied using *print_cnd().* While the only parameter of a condition is the frame number, this actually gives it access to the position data by simply accessing *x_vals[frameNum]*, so you could for example find all the values of $\vec{r}$ for which *x == 0*, or for which *x' == 0* by numerically computing *x'* from the values near *x_vals[frameNum]*. The other two requirements for a condition is that it should never throw an error, it should simply return "*NaN*" for non valid inputs, and finally and most importantly a condition must return a signed number that changes signs as r passes through the point that satisfies the condition and 0 if the condition is satisfied. For example if you are trying to set a condition to find when *y* =5 you could simply return *y_vals[frameNum] -5*, since this satisfies the requirement that the smaller the number the closer the condition is to being met and in addition the return value changes signs when as *y* passes 5.

The reason the sign of a condition must change is to determine when a condition is met somewhere between two sample points, which happens quite often. In this case the function returns the point that is closest to satisfying the condition and reports that it is an approximate answer*, r()* also does a similar thing if the time requested is in between points.

You can also call *animate()* which will draw the space curve as a function of time. *animate*() can take one optional argument which specifies the number of points to draw each frame. The default number is 50. If the animation call *animate()* again with a larger argument.

Changing the program

Modifying the source code is a key part of using difEQ, luckily this is really easy to do. All of the code that should be modified under normal operations are in *difEQ.h, equations.c, or graphUtility.py*. While this might sound like a lot of different files it's easy to figure out what should go where. If you want to edit the various system variables like *showProgress* go to *difEQ.h*. If you want to add a package to simulate a new equation edit *equations.c*. If wish to add functionality to the python interface edit *graphUtility.py*

To add a package to simulate a new equation simply add a new set of three equations to "equations.c", increment the numEQ variable, add pointers to your equations to the eqns[] variable, and add your equations package name to the eqnNames[] variable. Recompile difEQ using "make -Ofast" and then you can simply call your package from the input file as normal.

- To change the preset initial conditions change the values of *R_0* and *V_0.*
- To change the preset start and end time change the values of *T_INIT* and *T_FIN*
- To change the preset time step and fraction stored change *DELTAT* and *STOREFRAC*
- To change whether the program outputs the time taken for various operations change *useTime*
- To change whether the program shows its progress through various operations change *showProgress*
- To change the file that output is written to change *output*
- Also feel free to add new features to *difEQ.c* or to *graphUtility.py*

Tips

- If the program is taking a long time in the computing phase try to either decrease the number of simulation frames by increasing *deltaT* or decreasing the simulation range, or simplifying the modifying functions.
- If the program is taking a long time in the data writing phase try to decrease the number of points that are written to disk by increasing *deltaT*, decreasing the simulation range, or increasing *storeFrac*.
- To increase the accuracy of individual points decrease *deltaT*.
- To increase the accuracy of python queries and the smoothness of the graph decrease *storeFrac*.
- In my experience a good compromise between speed and accuracy is about 100 points per unit of time.

For more specific info look the source code comments.