Numerical Steppers: Implicit versus Explicit, Single- versus Multi-Step

Joseph Iosue
May 2017

I will treat systems of the form $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ and compare how different vector valued functions $\mathbf{f}$ effect the accuracy and efficiency of different classes of numerical steppers. The `Julia` code with all the implemented methods is included following the bibliography. Small snippets of code will be included in context throughout as needed. All the code was executed at *JuliaBox.com*.

I. INTRODUCTION

It is generally true that implicit methods are harder to implement than explicit methods, and that multi-step methods are slower than single-step methods. However, often the need to decrease the step size in a single-step or explicit method in order to attain good results takes away the original advantage of being faster. Every method performs differently in any given system.

Implicit methods are indeed harder to implement, and multi-step methods are slower for a given step size than their single-step counterparts. However, accurate solutions often require implicit or multi-step methods, or otherwise need impossibly small step sizes to perform well with explicit or single-step methods.

The section below highlights the benefits of these more complicated methods; the particular example shows an case where multi-step methods do far better than their single-step parallel for a given step size, and the implicit methods are much more accurate than their explicit equivalent.

A. Motivating Example

For a system of the form $\dot{y} = f(t, y)$,

$$\Delta y \equiv y(t+h) - y(t) = \int_t^{t+h} f(t, y(t))dt$$

Provided h is sufficiently small, one could make two simple approximations.

1. $\Delta y \approx hf(t, y(t))$
2. $\Delta y \approx hf(t+h, y(t+h))$

These give the two simplest numerical methods for stepping forward one time step:

- $y_{n+1} = y_n + hf(t_n, y_n)$ (Euler)
- $y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$ (Backward Euler)

Notice that Euler's Method is *explicit*; approximating the value of the function at the next time step depends only on values of the function at previous time steps.

In contrast, Backward Euler's Method is *implicit*; approximating the value of the function at the next time step depends on values of the function at previous and later time steps.

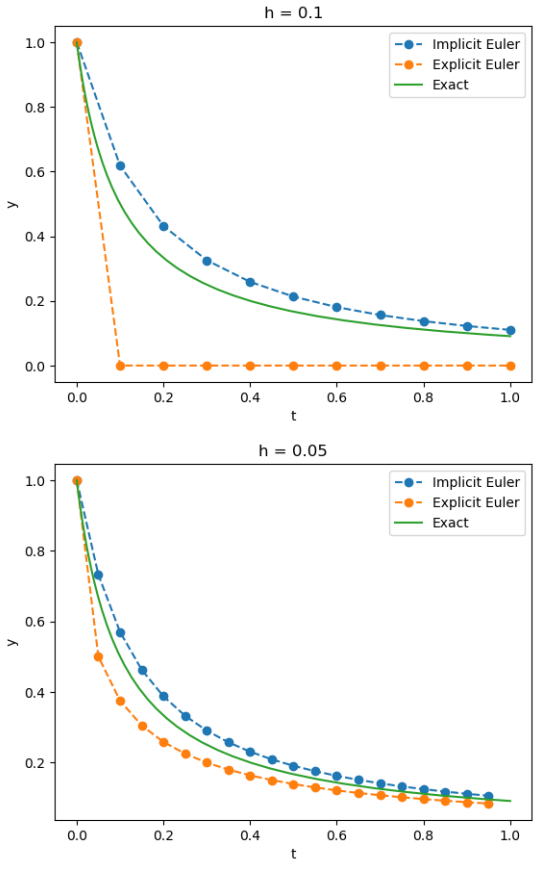
Performing an implicit step seems counter intuitive, since it uses future information to deduce future information. Techniques to perform such a task will be discussed later, but, for now, it is useful to see an example that motivates the need to utilize implicit methods in some situations. Consider the system

$$\dot{y} = -10y^2 \qquad y(0) = 1$$

which is solved by

$$y(t) = \frac{1}{10t + 1}$$

First, we apply explicit and implicit (forward and backward) Euler methods to this example with step size $h = 0.1$ for the range $t \in [0, 1]$. Then we decrease the step size to $h = 0.05$ and apply it again.

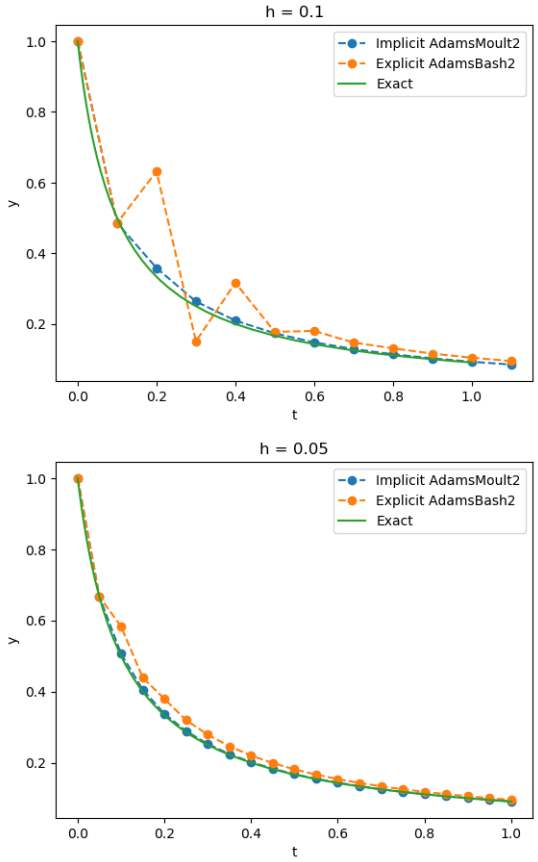


With step size $h = 0.1$, forward Euler is practically useless, whereas backward Euler performs very well.

By decreasing the step size to $h = 0.05$, the forward Euler method has less dramatic error, but the backward Euler method has very similar error.

Perhaps one of the main goals in the field of numerical methods is to have as large a step size as possible while maintaining an error within some tolerance. One could decide that the difference in overall error between the two backward Euler results is not worth the 2x extra work required, whereas that work is required to get any sort of reasonable result with the forward Euler method, seeing as it bears practically no resemblance to the correct solution curve in the first graph.

Now, to motivate the need for multi-step methods, we will solve the same system, but with AdamsBashforth Order 2 explicit method and Adams-Moulton Order 2 implicit method (details of these methods discussed later).



Both these results are vastly superior to those for backward and forward Euler for the same step size. However, they require more function evaluation per step - in this case, since they are order 2, twice as many.

II. HISTORY

The Adams and Bashforth paper on multi-step methods was published in 1883, twelve years before the famous 1895 paper by Runge [7]. The original Adams-Bashforth paper suggested the use of multi-step implicit methods, but they were not studied in detail until Moulton's 1926 paper [8]. The Adams-Moulton methods have the advantage over Adams-Bashforth methods that they do not require as many as many past values to achieve the same order convergence. But, as with any implicit method, the have the disadvantage of being notoriously hard to implement.

Most commonly, Adams-Bashforth and Adams-Moulton methods are used together as "predictor-corrector". The predicted step value is found with the explicit Bashforth variant. Then, the corrected value is computed by using the implicit Moulton variant with the $\mathbf{f}(t, \mathbf{y})$ replaced by the formerly computed predicted value [7].

The order is chosen to be the same for both the predictor and corrector methods. Therefore, if the order of the explicit method is p, then the order of the implicit method, and thus the overall order of the predictor-corrector method, becomes $p + 1$ [7].

I have implemented Adams-Bashforth and Adams-Moulton methods of order two, three, and four. These methods are called with `MultiStepper(f, method, t0, y0, tf, h)`. I did not, however, implement predictor-corrector methods. Instead, each method is just executed as is.

Implicit Runge-Kutta methods were proposed by J. Kuntzmann and by J. Butcher. They are centered around methods based on Gaussian quadrature. The most common example is Gauss-Legendre order two, which apparently was created prior to the general formulation of the Gauss-Legendre methods [1].

The single-step implicit methods that I implemented, called with `Stepper(f, method, t0, y0, tf, h)`, are sometimes classified as "semi-implicit Runge-Kutta methods" [1]. This means that each stage can be evaluated in succession because each implicit stage depends only on one unknown. For example, something of the form $y_{n+1} = y_n + f(t_{n+1}, y_{n+1})$ is semi-implicit because it depends on only one unknown, namely y_{n+1}; but $y_{n+1} = y_n + f(t_{n+1}, y_{n+1}) + f(t_{n+2}, y_{n+2})$ would not be because it depends y_{n+1} and y_{n+1}, both of which are not yet known.

In order to estimate the error of a given step, modern Runge-Kutta methods compute the step with two different methods of different orders and compare. In order to minimize function evaluations, methods are sought that share at least some stages. For example, the Runge-Kutta-Fehlberg step, implemented as `RKFStep`, computes six function evaluations. It uses five of these to compute a fifth order step, and uses four of them to compute a fourth order step. The fifth order step is taken, and the difference between the fifth and fourth order step is used to estimate the error. These methods are called with `AdaptiveStepper(f, method, t0, y0, tf)`.

III. IMPLICIT METHODS

Note: This section will deal only with single-step implicit methods. I will discuss multi-step implicit methods in section IV.

Implicit methods are notoriously difficult to implement because they require function values from the future. Often root finding techniques are used to deduce this future value. All my implicit method implementations use the function `NewtonsMethod` in the following way.

Consider Backward Euler $y_{n+1} = y_n + hf(t+h, y_{n+1})$. Thus, the step itself is $hf(t+h, y_{n+1})$. In order to implement this, we can use a root finding method to solve the equation

$$hf(t+h, y_n + s) - s = 0$$

where s is the Backward Euler step and the unknown. y_n is the previous value of the function, so it is already known.

This technique can be problematic because error now not only occur during the actual step, but also in the root finding algorithm. Often this is not important since Newton's method converges exponentially; nonetheless, implicit methods that use root finding take more time to complete because they must find a root at every step.

In section IIID, I will discuss the Newton's method implementation versus a direct implementation, but for the rest of the paper, I will use Newton's method because it works in generality.

A. Backward Euler

A Backward Euler step is

$$y_n \to y_{n-1} + hf(t+h, y_n)$$

Let $\dot{y}_n \equiv f(t_n, y_n)$. Then, it becomes

$$y_1 \to y_0 + h\dot{y}_1$$

Taylor expanding y_0 gives

$$\begin{aligned} y_1 &\to y_1 - h\dot{y}_1 + h\dot{y}_1 + \mathcal{O}(h^2) \\ &= y_1 + \mathcal{O}(h^2) \end{aligned}$$

Thus, the error per step is $\sim h^2$. Therefore the overall error for the Backward Euler's method is $\sim h$.

B. Gauss-Legendre Two

A Gauss-Legendre Two step is

$$y_n \to y_{n-1} + hf\left(t + \frac{h}{2}, y_{n-1} + \frac{1}{2}hf(t+h, y_n)\right)$$

With the same notation convention as before, it becomes

$$y_1 \to y_0 + hf\left(t_0 + h/2, y_0 + \frac{1}{2}h\dot{y}_1\right)$$

Notice that the step is an approximation of $\dot{y}_{1/2}$ (i.e. the slope at a half step later). So it becomes

$$y_1 \to y_0 + h\dot{y}_{1/2}$$

Now, plug in the Taylor expansion of $\dot{y}_{1/2}$ and y_1.

$$\begin{aligned} y_0 + h\dot{y}_0 + \frac{1}{2}h^2\ddot{y}_0 &\to y_0 + h\left(\dot{y}_0 + \frac{1}{2}h\ddot{y}_0\right) + \mathcal{O}(h^3) \\ 0 &\to \mathcal{O}(h^3) \end{aligned}$$

Thus, the error per step is $\sim h^3$. Therefore, the overall error for the Gauss-Legendre Two method is $\sim h^2$.

C. Crank-Nicholson

A Crank-Nicholson step is

$$y_n \to y_{n-1} + \frac{h}{2}\left(f(t, y_{n-1}) + f(t+h, y_n)\right)$$

Using the same notation as before, it becomes

$$y_1 \to y_0 + \frac{h}{2}(\dot{y}_0 + \dot{y}_1)$$

It is clear that the step is just averaging the slope at either endpoint of the interval $[t, t+h]$. Taylor expanding y_1 and $\dot{y}_1$ gives

$$\begin{aligned} y_0 + h\dot{y}_0 + \frac{1}{2}h^2\ddot{y}_0 &\to y_0 + \frac{h}{2}\left(2\dot{y}_0 + h\dot{y}_0 + \frac{1}{2}h^2\ddot{y}_0\right) + \mathcal{O}(h^4) \\ 0 &\to \frac{1}{4}h^3\ddot{y}_0 + \mathcal{O}(h^4) \\ 0 &\to \mathcal{O}(h^3) \end{aligned}$$

Thus, the error per step is $\sim h^3$. Therefore, the overall error for the Crank-Nicholson method is $\sim h^2$.

D. Example Implementation

I will use Crank-Nicholson to illustrate a few important facts about the implementation of single step implicit methods. There are two primary ways to implement second order implicit method Crank-Nicholson; (1) attempting to algebraically rearrange into an explicit method (system specific), or (2) use Newton's method (general). In order to illustrate this and compare, I will discuss the problem of a particle subject to perpendicular electric and magnetic fields. Let

$$\vec{E} = \hat{x} \quad \vec{B} = \hat{z} \quad \vec{r}(0) = (0,0,0) \quad \dot{\vec{r}}(0) = (0,0,0)$$

The Lorentz force law gives $\ddot{\vec{r}} = \frac{q}{m}(\vec{E} + \dot{\vec{r}} \times \vec{B})$. For simplicity, normalize the system. $q = m = 1$. Thus, $\ddot{z} = 0$ and

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

First, I will apply Crank-Nicholson to the velocities via method (1); $\vec{r}_{n+1} - \dot{\vec{r}}_n = \frac{h}{2}(f(t, \dot{\vec{r}}_n) + f(t + h, \dot{\vec{r}}_{n+1}))$. Plugging in,

$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}^{\text{new}} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} + \frac{h}{2} \left\{ \begin{pmatrix} 2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \left[\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} + \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}^{\text{new}} \right] \right\}$$

Solving this matrix equation gives

$$\begin{pmatrix} 1 & -h/2 \\ h/2 & 1 \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}^{\text{new}} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} + \frac{h}{2} \left[\begin{pmatrix} 2 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} \right]$$

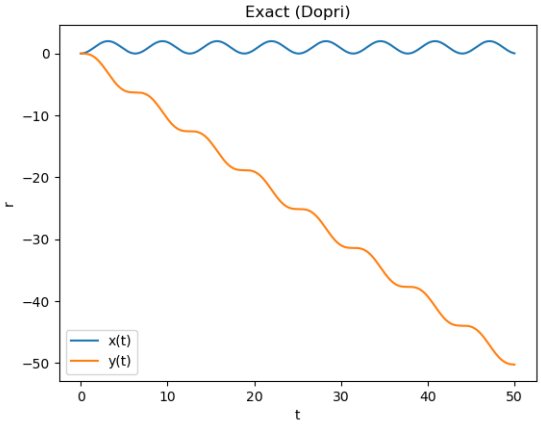
Multiplying by the inverse of the matrix on the left side gives an explicit stepper!

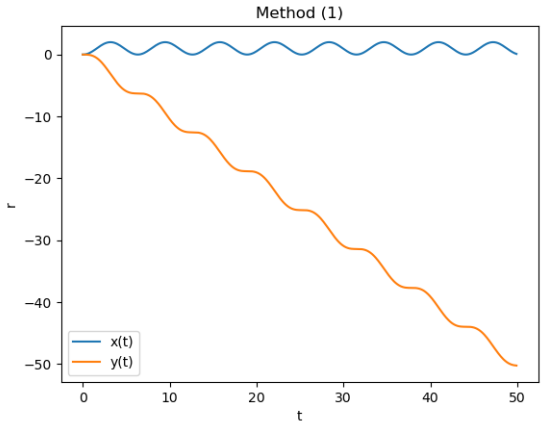
For this method, I will preform the above step and then update the position by a simple Euler step $\vec{r}_{n+1} = \vec{r}_n + h\dot{\vec{r}}$.

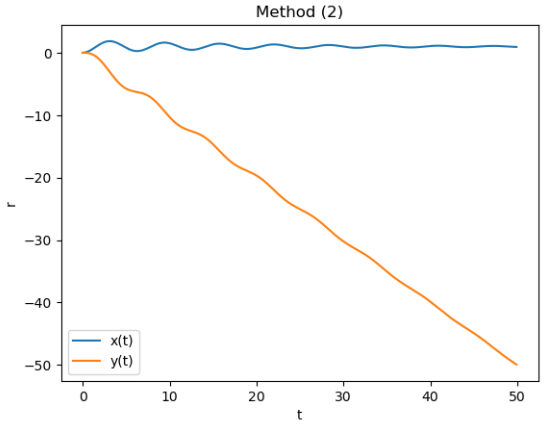
For method (2), I will use the functions `Stepper` and `CrankNicholsonStep` from the code along with

```
function f(t, yvec)
    x, y, vx, vy = yvec
    [vx, vy, 1+vy, -vx]
end
```

For comparison, the Dormand-Prince method (`AdaptiveStepper(DopriStep)`) will be used as the (approximately) exact solution. The results for $h = 0.1$ are





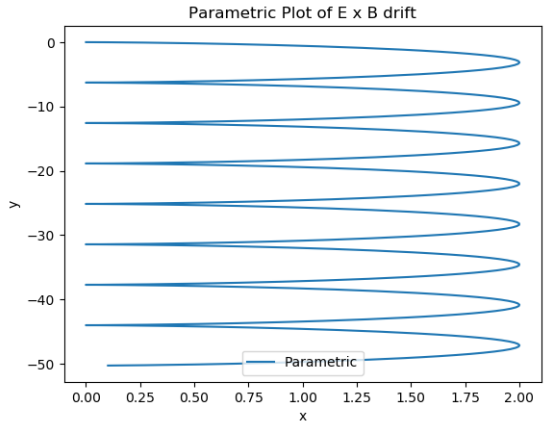


As expected, method (1) preforms far better than method (2), since errors accumulate with (2) not only from the method itself, but also from Newton's method. Unfortunately, though, method (1) cannot be implemented in general for an arbitrary system. We were able to get rid of the implicit part of Crank-Nicholson by plugging in the form of $\mathbf{f}(t, \mathbf{y})$ and rearranging, but this rearranging completely depends on $\mathbf{f}(t, \mathbf{y})$, so it is system specific. On the other hand, method (2) is very easy to implement in general (as can be seen in my code included). But this ease comes with two downfalls: first, it has greater error, and second, it takes more time to run since on every step it must perform Newton's method.

The moral of this result is that implicit methods can be algebraically simplified for certain systems and perform very well in these cases, but this requires more work by the person and must be re-implemented specifically for each system. Using a simple root finding algorithm to perform implicit methods is very easy and general for the person, but requires more work by the computer with still worse results. Nonetheless, from now on I will use Newton's method to perform all implicit methods because this implementation is of particular interest since applies to all systems.

The motion of this particle exhibits "E-cross-B drift"[5]. The electric field is in the x direction and the magnetic field is in the z direction, but the particle drifts in the $-y$ direction, where $-\hat{y} = \hat{E} \times \hat{B}$. It is easy to see

this with a parametric plot of motion (created using the result of method (1)).



One may have guessed that the particle would drift in the x direction since the electric field is directly applying a force. Once again, math trumps intuition.

IV. MULTI-STEP METHODS

To understand the mathematical underpinnings of the methods, I will derive the coefficients of both two-step Adams-Bashforth and Adams-Moulton. The math for higher orders follows the same procedure.

A. Two-Step Adams-Bashforth

Let $f_n \equiv f(t_n, y_n)$. The form of the p^{th} order Adams-Bashforth method is

$$y_p = y_{p-1} + h\sum_{i=0}^{p-1} a_i f_i$$

Intuitively, it makes sense that

$$\sum_i a_i = 1$$

since we are basically averaging the derivative of y in order to estimate the step. For the two-step method,

$$a_0 = a \qquad a_1 = 1 - a$$

Then,

$$y_2 = y_1 + h((1-a)f_1 + af_0)$$

Recall that $\dot{y}_n = f(t_n, y_n)$. Thus,

$$y_2 = y_1 + h((1-a)\dot{y}_1 + a\dot{y}_0)$$

Next, expand $\dot{y}_0$ by Taylor series.

$$\begin{aligned} y_2 &= y_1 + h((1-a)\dot{y}_1 + a(\dot{y}_1 - h\ddot{y}_1 + \mathcal{O}(h^2))) \\ &= y_1 + h\dot{y}_1 - ah\dot{y}_1 + ah\dot{y}_1 - ah^2\ddot{y}_1 + \mathcal{O}(h^3) \\ &= y_1 + h\dot{y}_1 - ah^2\ddot{y}_1 + \mathcal{O}(h^3) \end{aligned} \tag{1}$$

Now, expand y_2 by Taylor series.

$$y_2 = y_1 + h\dot{y}_1 + \frac{1}{2}h^2\ddot{y}_1 + \mathcal{O}(h^3) \tag{2}$$

Finally, subtract equation (1) from equation (2).

$$0 = \frac{1}{2}h^2\ddot{y}_1 + ah^2\ddot{y}_1 + \mathcal{O}(h^3)$$

In order to cancel out the h^2 term, $a = -1/2$. Therefore,

$$y_2 = y_1 + \frac{3}{2}hf_1 - \frac{1}{2}hf_0 + \mathcal{O}(h^3)$$

This gives the Adams-Bashforth two-step method,

$$y_{n+1} \to y_n + \frac{3}{2}hf(t_n, y_n) - \frac{1}{2}hf(t_{n-1}, y_{n-1})$$

Each step has an error $\mathcal{O}(h^3)$. Thus, the overall method has an absolute error $\sim h^2$.

B. Two-Step Adams-Moulton

The form of the p^{th} order Adams-Moulton method is

$$y_p = y_{p-1} + h\sum_{i=0}^{p} a_i f_i$$

Again,

$$\sum_i a_i = 1$$

Therefore, for second order,

$$\begin{aligned} y_2 &= y_1 + ha_2\dot{y}_2 + ha_1\dot{y}_1 + ha_0\dot{y}_0 \\ &= y_1 + ha_2\dot{y}_2 + h(1 - a_2 - a_0)\dot{y}_1 + ha_0\dot{y}_0 \\ &= y_1 + ha_2(\dot{y}_1 + h\ddot{y}_1 + \frac{1}{2}h^2\dddot{y}_1 + \mathcal{O}(h^3)) \\ &\quad + h(1 - a_2 - a_0)\dot{y}_1 + ha_0(\dot{y}_1 - h\ddot{y}_1 + \frac{1}{2}h^2\dddot{y}_1 + \mathcal{O}(h^3)) \\ &= y_1 + h\dot{y}_1 + h^2\ddot{y}_1(a_2 - a_0) + \frac{1}{2}h^3\dddot{y}_1(a_0 + a_2) + \mathcal{O}(h^4) \end{aligned} \tag{3}$$

The Taylor expansion of y_2 is

$$y_2 = y_1 + h\dot{y}_1 + \frac{1}{2}h^2\ddot{y}_1 + \frac{1}{6}h^3\dddot{y}_1 + \mathcal{O}(h^4) \tag{4}$$

Subtracting equation (4) from equation (3) yields

$$0 = h^2\ddot{y}_1\left(a_2 - a_0 - \frac{1}{2}\right) + \frac{1}{2}h^3\dddot{y}_1\left(a_0 + a_2 - \frac{1}{3}\right) + \mathcal{O}(h^4)$$

Therefore, to get rid of all terms of order less than $\mathcal{O}(h^4)$,

$$a_2 - a_0 - 1/2 = 0 \qquad a_0 + a_2 - 1/3 = 0$$

Recall, $a_1 = 1 - a_0 - a_2$. Solving this system yields

$$a_0 = -1/12 \qquad a_1 = 2/3 \qquad a_2 = 5/12$$

Therefore,

$$y_2 = y_1 + \frac{5}{12}hf_2 + \frac{2}{3}hf_1 - \frac{1}{12}hf_0 + \mathcal{O}(h^4)$$

This gives the Adams-Moulton two-step method,

$$\begin{aligned} y_{n+1} \rightarrow & y_n + \frac{5}{12}hf(t_{n+1}, y_{n+1}) \\ & + \frac{2}{3}hf(t_n, y_n) - \frac{1}{12}hf(t_{n-1}, y_{n-1}) \end{aligned}$$

Each step has an error $\mathcal{O}(h^4)$. Thus, the overall method has an absolute error $\sim h^3$.

C. General Adams Method

It is easy to see that for an Adams-Bashforth method of order p, one can choose coefficients such that all error terms have an order $\sim h^{p+1}$ for each step. This is precisely how the coefficients are chosen. Thus, an Adams-Bashforth method of order p has an overall error $\sim h^p$.

Likewise, for an Adams-Moulton method of order p, one can choose coefficients such that all error terms have order $\sim h^{p+2}$ for each step. Thus, an Adams-Moulton of order p has an overall error $\sim h^{p+1}$.

V. ADAPTIVE-STEP METHODS

I have implemented a few adaptive single-step single methods that compare two different routines in order to estimate the error error per step. Using this estimate, the step size can be increased or decreased as necessary. The `AdaptiveStepper` function turns out to have some very useful side effects.

First, if the step size becomes unreasonably small during a procedure, it usually indicates a stiff point or highly varying region. This provides vital information about the system and can predict asymptotes, infinities, and other such objects. In section VIIB, for example, an unreasonable step size around a black hole often indicates an event horizon.

Second, at every time step, the method being used estimates its deviation from the actual solution. Summing the magnitude of all these approximate errors over every time step can give an estimate of an upper bound on the total error. It is an upper bound because, in most cases, at least some error will cancel; perhaps the previous time step was too negative while the next is too positive, making the overall error smaller than the sum of the magnitudes of the errors of each time step. However, the calculated error at each step is an estimate itself, and the real error could indeed be larger at each step. Therefore, the set upper bound is not strict; the total error could indeed be larger than the estimated upper bound. Nonetheless, the approximation is useful and very often holds.

A. Error Estimates

To really test the error estimates, I will use an extreme example. As mentioned before, the error will not usually be as high as the estimated upper bound because errors of opposite sign would cancel. However, consider the system

$$\dot{y} = e^t \qquad y(t_0) = e^{t_0}$$

solved by

$$y(t) = e^t$$

I call this "extreme" because $\mathbf{f}(t, \mathbf{y})$ is always increasing exponentially; presumably, then, the methods will always either underestimate or overestimate meaning that the sign of the error will always be the same. Therefore, error may not cancel, making the upper bound estimate not necessarily accurate.

Results

I will compare the values of `DopriStep`, `CarpStep`, and `RKFStep` at t_f to the exact result $y(t_f) = e^{t_f}$ for various pairs of t_0 and t_f. This is implemented in the function `Test_AdaptiveSteppers`.

- $t_0 = 0$, $t_f = 1$

```
DopriStep
    Estimated upper bound: 5.13e-9
    Actual error: 9.82e-11
CarpStep
    Estimated upper bound: 1.31e-8
    Actual error: 6.03e-10
RKFStep
    Estimated upper bound: 9.32e-9
    Actual error: 1.44e-9
```

  The actual error for each method is indeed less than the estimated upper bound.

- $t_0 = 17$, $t_f = 18$

```
DopriStep
    Estimated upper bound: 2.80e-6
    Actual error: 4.43e-6
CarpStep
    Estimated upper bound: 3.50e-6
    Actual error: 5.29e-6
RKFStep
    Estimated upper bound: 2.48e-6
    Actual error: 5.30e-6
```

For these larger values of t, the slope e^t is becoming very large. Thus, the actual error is catching up to the estimated upper bound and, for `CarpStep` and `RKFStep`, surpassing it.

- $t_0 = 30$, $t_f = 31$

```
DopriStep
    Estimated upper bound: 0.0015
    Actual error: 3.75
CarpStep
    Estimated upper bound: 0.00033
    Actual error: 6.26
RKFStep
    Estimated upper bound: 0.0012
    Actual error: 1.36
```

For these large values of t, the actual error surpasses the upper estimate by an order of magnitude of 10^3.

Despite the error estimates being significantly wrong in the last example, the results still show that the estimates are relatively accurate. After all, it wasn't until the slope $\mathbf{f}(t, \mathbf{y})$ became on the order of $\approx e^{18} \sim 10^7$ that the actual error just reached the upper bound.

Most systems are not as one-sided as this one, but even still the error estimates provided a good, useful upper bound.

VI. CONVERGENCE ANALYSIS

In order to test how quickly the various methods converge with step size, I will test against a few analytically solvable systems and compare the numerical results to the exact solutions.

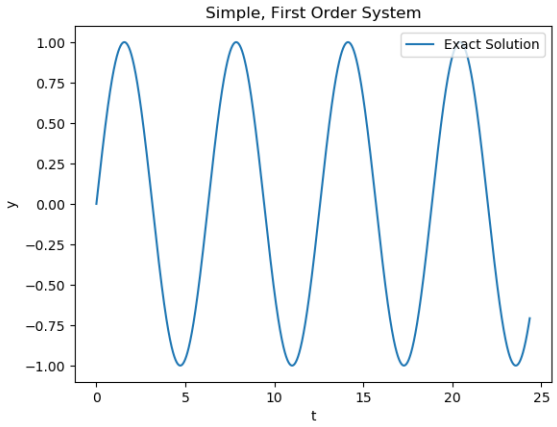
A. Simple, First Order System

Consider the first order differential equation

$$\dot{y} = \cos t \qquad y(0) = 0$$

which is analytically solved by

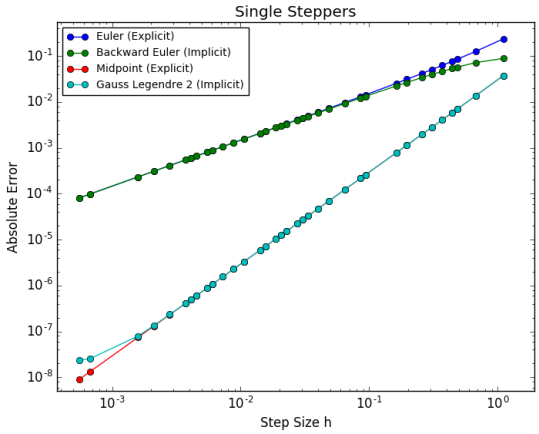
$$y(t) = -\sin t$$



I will use this exact solution to preform convergence analysis on the various methods that I have discussed by comparing values of $y(31\pi/4)$ to the exact result $y(31\pi/4) = -1/\sqrt{2}$. The function `SimpleSystem` implements this system.

1. *Single-Steppers*

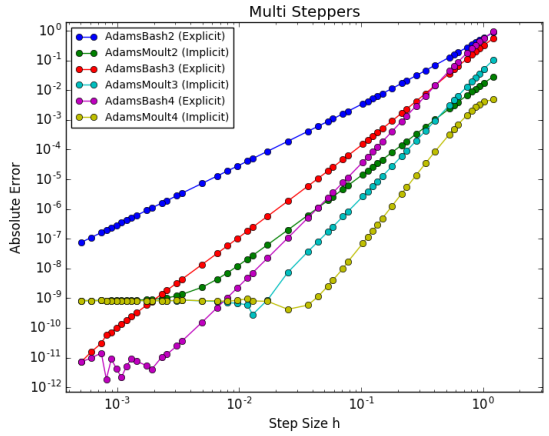
For this analysis, we are particularly interested in comparing sister methods; i.e. Backward Euler to Euler and Gauss Legendre order 2 to Midpoint.



The implicit/explicit pairs perform almost exactly the same. Both Euler and Backward Euler converge with $\sim h$ and both Midpoint and Gauss Legendre 2 converge with $\sim h^2$. But notice how the last two points of GL2 start to curve upward. This is a result of error in Newton's method, which I will discuss more below.

2. *Multi-Steppers*

As with the single steppers, we are interested in comparing the Adams-Bashforth explicit methods to their sister Adams-Moulton implicit methods.



Each explicit method converges at about one order higher than expected, while the implicit methods converge as expected. AdamsBash2 and AdamsMoult2 converge with $\sim h^3$, AdamsBash3 and AdamsMoult3 converge with $\sim h^4$, and AdamsBash4 and AdamsMoult4 converge with $\sim h^5$. In fact, AdamsMoult4 is close to converging with h^6.

However, at an absolute error of about 10^{-9}, all three implicit methods suddenly level off and stop converging. This is a result of error accumulation from Newton's method and the finite difference derivative approximations that it uses. In my implementation of `NewtonsMethod`, the default value for `error_tol` is 10^{-10}. Thus, it is unexpected that any of the implicit methods would ever reach an absolute error below 10^{-10}, as confirmed in the plot.

In section IIID, I highlighted how a system specific implementation of an implicit method performs far better than a general root finding implementation. The result of the convergence of the multi-steppers here adds to this and shows another downfall of the straightforward root finding method.

B. Rapidly Driven, Damped Harmonic Oscillator

Consider the driven oscillator with $y(t)$ described by

$$\ddot{y} + \frac{1}{4}\dot{y} + y = 100\cos(20t)$$

This is analytically solved by a sum of a homogeneous solution:

$$y_h(t) = e^{-t/8}\left(a\cos\left(\frac{\sqrt{63}}{8}t\right) + b\sin\left(\frac{\sqrt{63}}{8}t\right)\right)$$

and a particular solution:

$$y_p(t) = \frac{50}{79613}\left(5\sin(20t) - 399\cos(20t)\right)$$

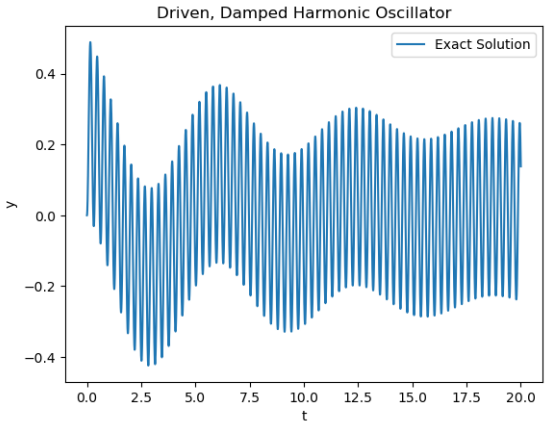
Enforcing the initial conditions

$$y(0) = \dot{y}(0) = 0$$

gives

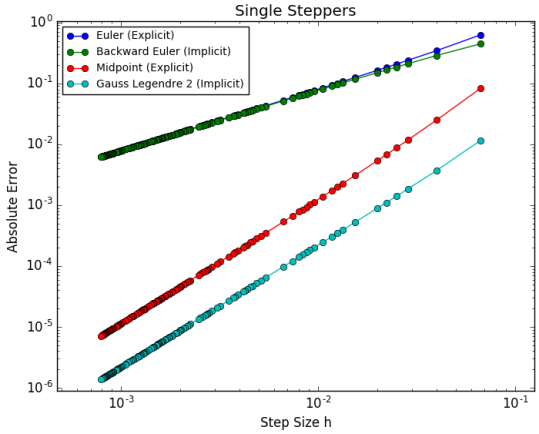
$$a = \frac{19950}{79613} \qquad b = \frac{1}{\sqrt{63}}\left(a - \frac{40000}{79613}\right)$$

This gives the familiar result of a decaying homogeneous oscillation and a steady state oscillation from the drive that eventually dominates.



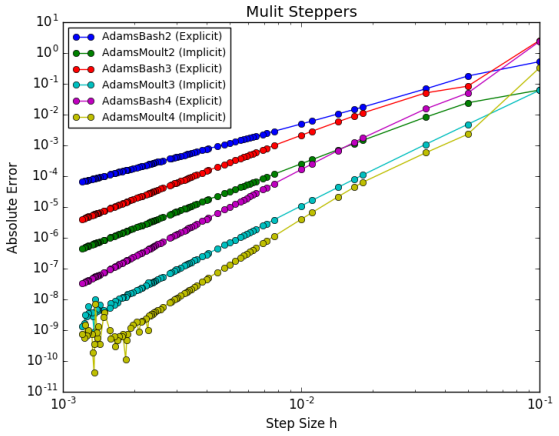
I will use this exact solution to again preform convergence analysis on the methods by comparing values of $y(20)$ to the exact result $y(20) = 0.13800260204215795$. The function `HarmonicOscillator` implements this system.

1. *Single-Steppers*



The absolute error of both explicit and implicit Euler decreases with h. In this example, neither is better than the other. Similarly, the absolute error of both explicit and implicit Midpoint decreases with h^2. But, in this example, the implicit method is overall better for every step size.

2. *Multi-Steppers*



We can see that, as expected, the convergence rate of each explicit method is h^p and each implicit method is $\sim h^{p+1}$ where p is the order of the method. We see again that the implicit methods start to fail around an absolute error of 10^{-9} due to error in Newton's method each step.

C. Fourth Order System

Consider the fourth order differential equation

$$y^{(4)} + 7y^{(3)} + 17\ddot{y} + 17\dot{y} + 6y = e^t \cos(100t)$$

The coefficients were chosen such that the characteristic polynomial is able to be factored. i.e.

$$(x+1)^2(x+2)(x+3) = x^4 + 7x^3 + 17x^2 + 17x + 6$$

With this factorization, finding the homogeneous solution is simple.

$$y_h(t) = (a+bt)e^{-t} + ce^{-2t} + de^{-3t}$$

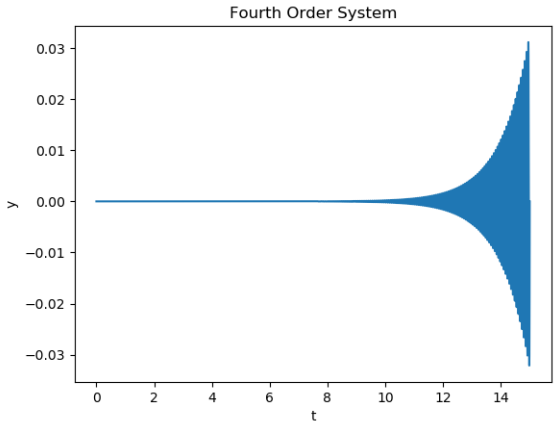
Using

$$e^t \cos(100t) = \frac{1}{2}\left(\exp(t(1+100i)) + \exp(t(1-100i))\right)$$

We realize that the particular solutions will be of the form $y_p = \alpha \exp(t(1 \pm 100i \pm \phi i))$ where ϕ is some phase factor. With this, finding the solution becomes an algebraic exercise. Enforcing the initial values

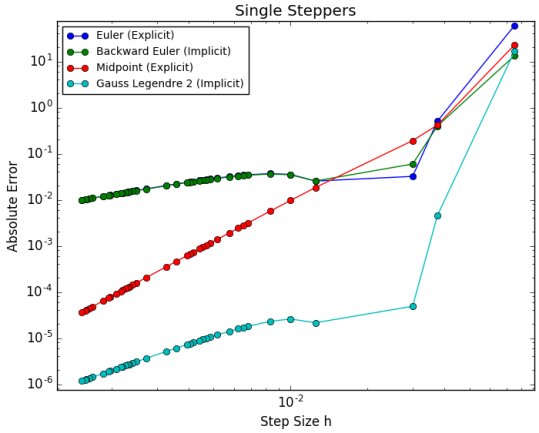
$$y^{(3)}(0) = \ddot{y}(0) = \dot{y}(0) = y(0) = 0$$

gives a closed form solution to the system.



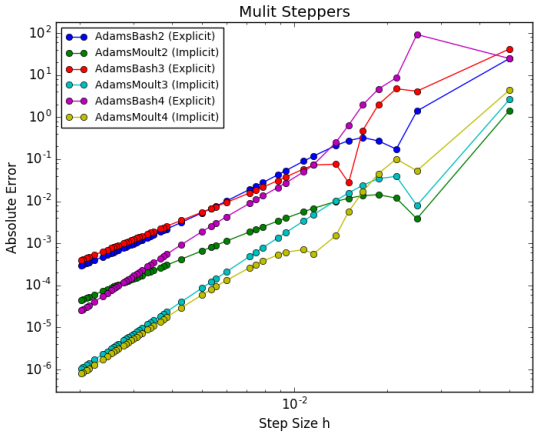
I will use this exact solution to again preform convergence analysis on the methods and see if the higher order of this system greatly effects the error. I will compare the values of $y(15)$ to the exact result $y(15) = -0.000017223530220167676$. The function `FourthOrderSystem` implements this system.

1. *Single-Steppers*



Compared to the harmonic oscillator convergence, the convergence for this fourth order system seems to be overall worse. For relatively large step size, all the single steppers converge more rapidly than for the HO, except for the Midpoint method, which converges about the same. But as the step size decreases, the convergence rates seem to get worse. For $h \in [10^{-3}, 10^{-2}]$, Gauss Legendre order 2 only converges with h, and both Euler and Backward Euler converge with what looks to be $\sqrt{h}$. This may be because the fourth order system is rapidly varying; large step sizes, then, are practically useless, but as the step size increases, the variation on the scale of the step size is still very large, so there is still slow convergence. We see again that Gauss Legendre order 2 performs significantly superior to the others, and Forward and Backward Euler have about the same error.

2. *Multi-Steppers*



Again we see that both Adams Moulton order 3 and 4 have less error than all the explicit methods. But the convergence rates are overall worse than for the HO. Adams Moulton order 3 and 4 and Adams Bashforth 4 converge with $\sim h^3$, and the rest of the methods converge with $\sim h^2$. It seems that, for a fourth order system, there is more room for error to accumulate. After all, we make approximations for the step of the zeroth, first, second, and third derivatives, as opposed to just the zeroth and first in the case of the harmonic oscillator.

VII. APPLYING THE METHODS

I will use various of the new methods discussed to implement three different systems. The main goal of this section is to see a few phenomena that may at first seem counter-intuitive. These visualizations prove the effectiveness of numerical methods and why they are so important in the computer age.

In section VIIA, we will see an interesting effective potential that causes a mass to "defy" gravity. In section VIIB1, we will see that a mass falling into a black hole radiates a significant amount of energy in the form of gravitational waves. In section VIIB2, we will see that rotating spacetime can give a particle angular momentum.

A. Kapitza Pendulum

For the entirety of this section, I will use the implicit `AdamsMoult4` *mulit-step method.*

Consider a pendulum with a massless rod of length l with a mass at the end and a rapidly moving pivot whose coordinates are

$$x_p(t) = a_x \cos(\omega_x t + \phi_x) \quad y_p(t) = a_y \cos(\omega_y t + \phi_y)$$

Let θ be the counterclockwise angle from the vertical where $\theta = 0$ describes the pendulum mass hanging, then the coordinates of the pendulum mass are

$$x(t) = x_p(t) + l\sin\theta(t) \quad y(t) = y_p(t) - l\cos\theta(t)$$

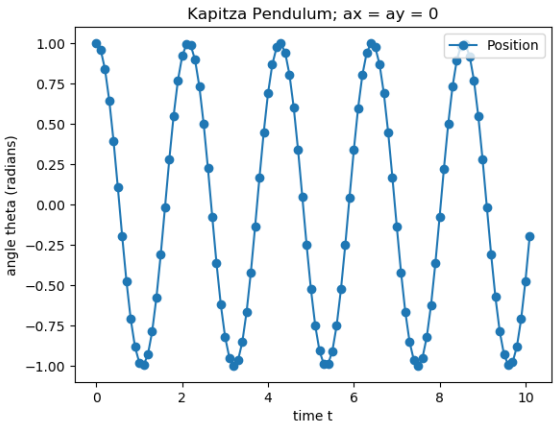
The Lagrangian and consequent equation of motion are

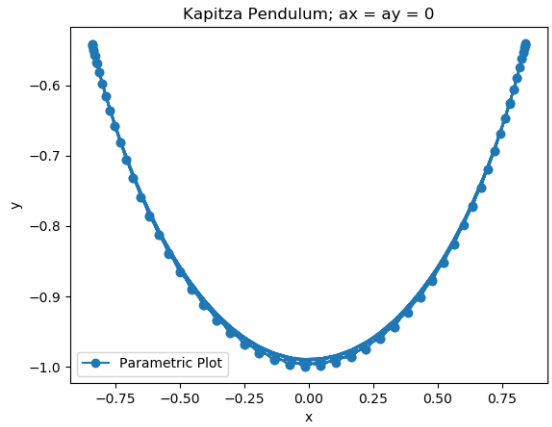
$$\mathcal{L} = \frac{1}{2}(\dot{x}^2 + \dot{y}^2) - gy \qquad \frac{\partial \mathcal{L}}{\partial \theta} = \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{\theta}}$$

The expansion is long enough that it is not included, but can be seen in my code in the function `KapitzaPendulum`.

1. *Sanity Check*

As a sanity check, let $a_x = a_y = 0$; we should recover a simple pendulum. In the code, calling `KapitzaPendulum()` with default arguments yields the system of a simple pendulum with $l = 1$ (m) and $g = 9.8$ (m/s). With starting position $\theta(0) = 1$ radian and $\dot{\theta}(0) = 0$ rad/sec, the result is as expected:

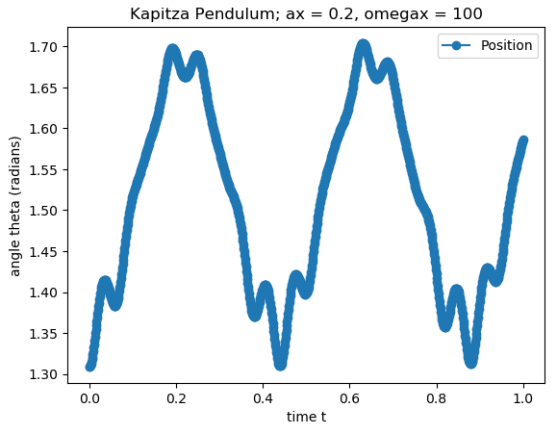


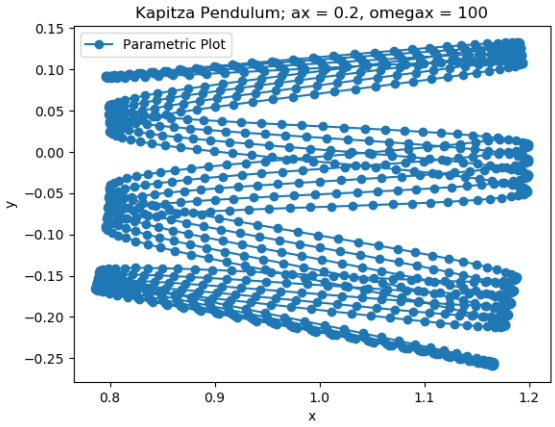


2. *New Equilibrium Loci*

Let $a_x = 0.2$ (m) and $\omega_x = 100$ (rad/s); the system is given by `KapitzaPendulum(ax=0.2, omegax=100)`. This corresponds to the pivot of the pendulum moving

only horizontally. With starting values $\theta(0) = 5\pi/12$ and $\dot{\theta}(0) = 0$, the result is





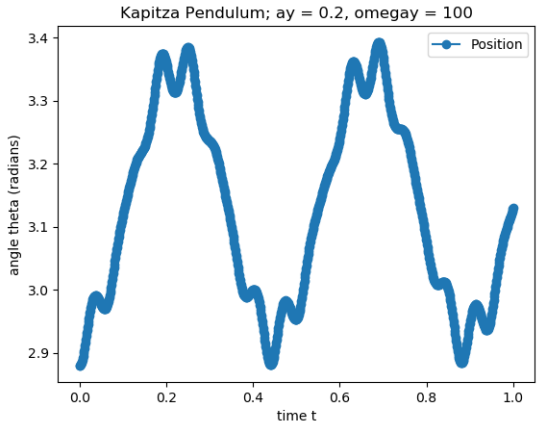
We see that the equilibrium position has changed! Going through the same process as above but with $\theta(0) = -5\pi/12$ yields a symmetric result. Thus, the equilibrium location has shifted from $\theta = 0$ to $\theta \approx \pm\pi/2$ (the loci tend toward $\pm\pi/2$ as $\omega_x \to \infty$). In other words, the pendulum is stable when it is horizontal! The stable points are no longer where the potential is minimized, but instead where the effective potential is minimized, and the effective potential is altered by the rapid motion of the pivot.

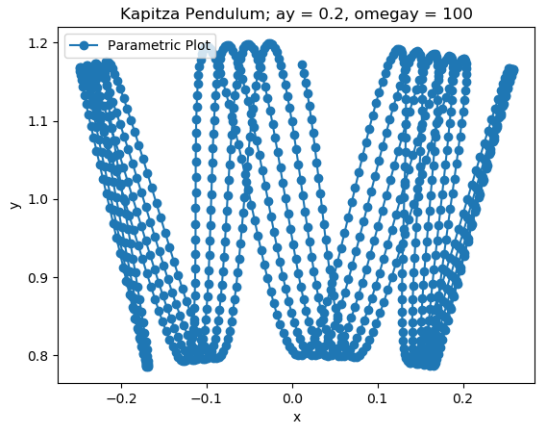
In general, $\theta = \pi$ is an equilibrium point because, for a rigid rod, all forces are balanced;

$$\left.\frac{dV}{d\theta}\right|_{\theta=\pi} = 0$$

But, $d^2V/d\theta^2 < 0$. Thus, any perturbation from the equilibrium point causes to mass on a standard pendulum to fall.

From the above result of the Kapitza Pendulum, though, one can guess that if the pivot move vertically then $\theta = \pi$ will become a stable equilibrium location. The system `KapitzaPendulum(ay=0.2, omegay=100)`, $\theta(0) = 11\pi/12$, and $\dot{\theta}(0) = 0$, results in

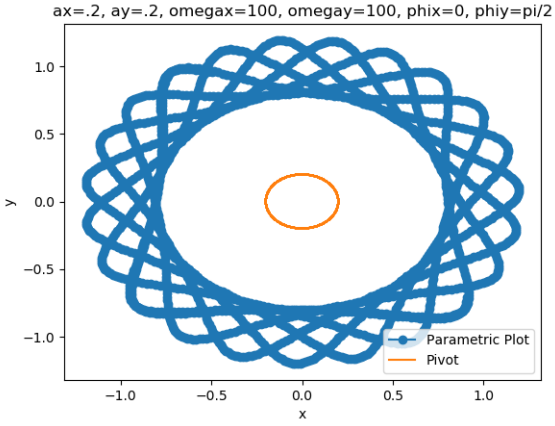




We indeed find that the formerly unstable equilibrium $\theta = \pi$ is now a stable equilibrium point.

3. *Kapitza Doodles*

Playing around with initial conditions and values of the constants can result in beautiful motion. For example, consider the system `KapitzaPendulum(ax=.2, ay=.2, omegax=100, omegay=100, phix=0, phiy=pi/2.0)`. This corresponds to the pivot in circular motion. Starting with $\theta(0) = \pi/4$ and $\dot{\theta}(0) = 0$ and solving for $t \in [0, 2]$ gives



B. Geodesics around Black Holes

For a particle moving with no external forces acting on it, the quantity $\Delta\tau/\Delta t$ will be maximized (i.e. closest to 1, since $\Delta\tau \leq \Delta t$ always) where t is the time coordinate for some observer and τ is the proper time of the particle. Thus, along geodesic paths, $\int d\tau$ is maximized where

$$c^2 d\tau^2 = g_{\mu\nu} dx^\mu dx^\nu$$

This corresponds to maximizing the action of a system, where the action is the functional $\int L dq$. Thus, we get the familiar Euler-Lagrange equations from Lagrangian mechanics. Simplifying gives the equations of motion for a particle on a geodesic in a spacetime given by the metric $g_{\mu\nu}$:

$$\frac{d^2 x^\mu}{d\tau^2} = -\Gamma^\mu_{\alpha\beta} \frac{dx^\alpha}{d\tau} \frac{dx^\beta}{d\tau}$$

$$\Gamma^\mu_{\alpha\beta} = \frac{1}{2} g^{\sigma\mu} \left(\partial_\alpha g_{\sigma\beta} + \partial_\beta g_{\sigma\alpha} - \partial_\sigma g_{\alpha\beta}\right)$$

where Γ are the Christoffel symbols and all contracted indices are summed over. I have implemented a function `geodesic` that takes in a function that outputs the desired Christoffel symbols and returns the function `f` for the system $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$.

Before beginning to numerically solve the geodesics, a condition must be set to describe $\frac{dt}{d\tau}|_{\tau=0}$ in terms of the other initial conditions. To do that, we must make use of the invariant (true in all reference frames all the time) quantity

$$g_{\mu\nu} \frac{dx^\mu}{d\tau} \frac{dx^\nu}{d\tau} = -c^2$$

As for any second order differential equation, all the initial positions and velocities can be set as desired, but the initial $\frac{dt}{d\tau}$ must be set to agree with $u^\mu u_\mu = -c^2$. From here on out, $\dot{k}$ will imply $\frac{dk}{d\tau}$.

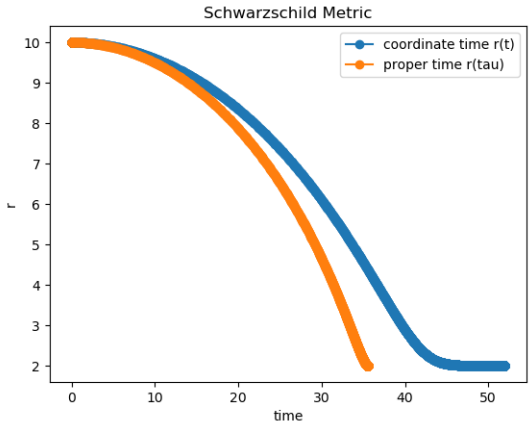
For the following sections, I will use a few different adaptive steppers. One very neat result of these is as the particle gets very close to the event horizon, the step size h (which corresponds to a step of τ) gets smaller and smaller until the `AdaptiveStepper` function says "Step size effectively zero at" some time. The proper time at which the step size becomes incredibly small is the same proper time as the particle crosses the event horizon (the particles proper time is finite upon crossing the event horizon, but the coordinate time from the perspective of an observer at infinity would of course go to infinity before the he/she sees the particle cross the horizon).

1. *Schwarzschild Spacetime*

The Schwarzschild metric describes the spacetime around a non-rotating, uncharged black hole from the perspective of an observer at infinity in a spherical coordinate system with the center of the black hole at the origin. It is a diagonal metric, making the Christoffel symbols easy to calculate by hand. I implemented the system in the function `Schwarzschild`.

Likewise, solving for $\dot{t}(0)$ is straightforward because of the simplicity of the metric. Let $\dot{r}(0) = \dot{\theta}(0) = \dot{\phi}(0) = 0$. Then, $g_{tt}\dot{t}^2 = -c^2$. Note that g_{tt} depends on r; therefore, $\dot{t}(0)$ depends on $r(0)$.

For simplicity, set the gravitational constant, mass of the black hole, and speed of light to one; $G = M = c = 1$. Let $t(\tau = 0) = \phi(\tau = 0) = 0$, $r(\tau = 0) = 10$ (i.e. five Schwarzschild radii), and $\theta(\tau = 0) = \pi/2$. Running `AdaptiveStepper` with `SimpleErrorStep`, which compares an implicit step with an explicit step, results in a tiny step size a little after $\tau = 35.56$; at this proper time, the particle crosses the event horizon. Running the same method a second time, but this time truncating before the critical time gives $\theta(\tau) = \pi/2$, $\phi(\tau) = 0\ \forall\tau$ and



The method yielded an estimated upper bound on total error of $\sim 10^{-4}$. We see that, if Newton's assumption of universal time were to hold true and an outside observer also experienced proper time (i.e. $\dot{t} = 1$), the particle would follow the typical in fall for a $1/r$ potential in both the particle's and the infinite observer's reference frame. However, time dilation does occur, and $\dot{t} \to \infty$ as the particle nears the event horizon at the Schwarzschild radius. Thus, an observer outside of a black hole never sees anything fall in!

Energy Lost to Gravitational Wave Radiation

For this section, I will switch to a non-adaptive step method. This will make approximating derivatives and integrals easier. I will use `AdamsBash4` with `MultiStepper`.

Define a Cartesian coordinate system such that $z = r$ from before (i.e. the particle fall along the z axis) and let the mass of the particle be $m = 1$. Then the second

mass moment tensor (equation 23.34 [3]) becomes

$$I^{ij} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & z^2 \end{pmatrix}$$

making the quadrupole moment tensor (equation 23.50 [3])

$$Q^{ij} = I^{ij} - \frac{1}{3}\delta^{ij} I^k_k = \frac{z^2}{3}\begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

and the resulting power radiated as gravitational waves (equation 23.51 [3])

$$P_{GW} = \frac{1}{5}\langle \dddot{Q}_{ij} \dddot{Q}^{ij} \rangle = \frac{8}{15}(3\dot{z}\ddot{z} + z\dddot{z})$$

To obtain the energy lost, the power lost must be integrated over with the z values from the stepper. The function `fraction_gwloss` takes in z values and a step size and estimates the first, second, and third derivatives by finite difference methods at every time step. Adding the power at each time step times h should give a rough estimate of the total energy lost. Solving for the motion using a constant step $h = 0.01$ with `AdamsBash4` and calling `fraction_gwloss` with the resulting z coordinates results in approximately a 0.96 fractional energy loss of the particle due to gravitational wave emission during its infall.

This result seems outlandish, and perhaps is (there is, of course, always the possibility that my implementation is flawed). But the initial conditions are themselves outlandish. The particle started at just four Schwarzschild radii away from the event horizon with zero velocity. Therefore, it is in an extremely strong gravitational field for a relatively long time since it is just beginning to accelerate. This system is not very physical, since any particle so close to a black hole will almost surely have some velocity. Nonetheless, we get a good idea of how extreme the spacetime around a black hole is with this rough estimate. This is perhaps why black holes are of such interest; they are some of the most extreme objects in the universe, and therefore take theories to their limits.

2. *Kerr Spacetime*

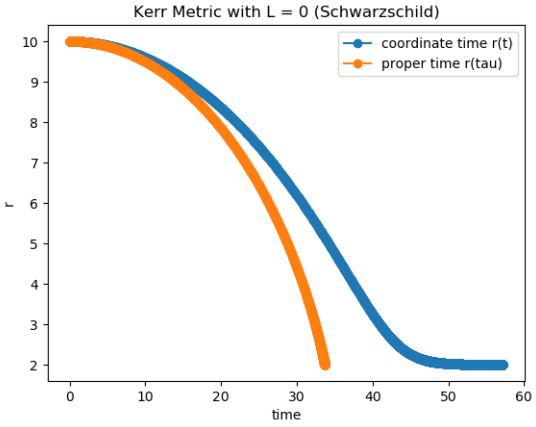
The Kerr metric describes the spacetime around a rotating black hole from the perspective of an observer at infinity in a spherical coordinate system with the center of the black hole at the origin. The metric has two off diagonal terms that make the Christoffel symbols unreasonable to enter directly into the code. Instead, I use the metric and a finite difference method to estimate the partial derivatives of the metric, and then use the definition of the Christoffel symbols in terms of the metric and its derivatives. This is all implemented in the function `Kerr`.

First, to check the system, I will use `Kerr(L=0)` with angular momentum $L = 0$ and $G = M = c = 1$. I will use the same initial conditions as for the Schwarzschild example before; $t(\tau = 0) = \phi(\tau = 0) = 0$, $r(\tau = 0) = 10$, $\theta(\tau = 0) = \pi/2$, and all coordinate velocities equal to zero, except $\dot{t}$ which is the same as before. Solving this system should yield the same as before, since the Kerr metric with $L = 0$ reduces to the Schwarzschild metric.

Running the `AdaptiveStepper` with the implicit method `RKTrapStep` results in a tiny step size a little after $\tau = 33.7$; at this proper time, the particle crosses the event horizon.

We already see there is some error from the method of estimating the metric partial derivatives with finite difference methods; in the Schwarzschild case the particle crossed the event horizon slightly after $\tau = 35.56$.

Running it a second time, but stopping it before it crossed the event horizon gives $\theta = \pi/2$, $\phi = 0$ $\forall\tau$ and

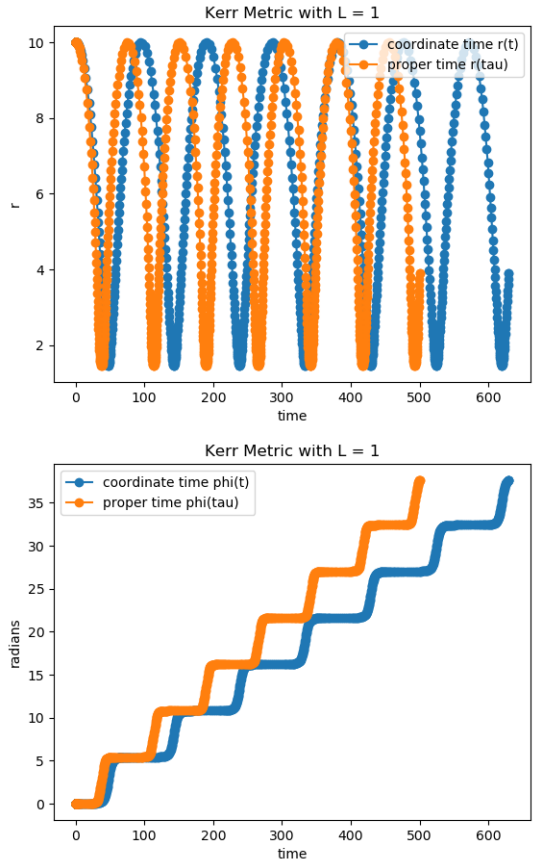


This method estimated an upper bound on the total error ~ 1. The graph is almost the same result as before with just a slight error in the time, as mentioned. This was a good sanity check; the implementation of the Kerr metric does indeed reduce to the Schwarzschild metric when $L = 0$.

However, `RKTrapStep` took a long time to complete and Newton's method took more than 50 iterations for many steps. Because of this, for the remainder of the analysis of the Kerr metric, I will use a different stepper.

Frame Dragging

I will now solve the system `Kerr(L=1)` with all the same initial conditions as before, but allow τ_{max} to be large. Again, $g_{tt}\dot{t}(0)^2 = -c^2$ since all the starting velocities are zero, and it turns out that g_{tt} has the same form as before. Note that L is defined to be along the z axis. Therefore, the black hole is rotating on its azimuthal angle ϕ. Running `CarpStep` with `AdaptiveStepper` yields $\theta = \pi/2$ $\forall\tau$ and



Newtonian mechanics predicts that a rotating mass would not affect masses differently than non-rotating masses. But General Relativity predicts a quite different result. The particle started with no angular velocity, but was given angular momentum by the rotating black hole; more specifically, the black hole causes spacetime itself to rotate, which then causes the particle to rotate. This effect is called "frame dragging" [6].

The particle exhibits multiple hyperbolic orbits. Every time the particle gets close to the black hole, it slingshots around as the azimuthal angle goes in steps of ~ 5.5 radians $\approx 315°$. Also, the particle's r coordinate is less than 2 (the Schwarzschild radius) every orbit! This is not an error; the event horizon of a Kerr black hole is smaller than that of a Schwarzschild black hole. In the Kerr metric, there is a zero in the denominator of g_{rr} when

$$r^2 - r_s r + \frac{L^2}{M^2 c^2} = 0$$

where r_s is the Schwarzschild radius. Thus, if $L = 0$, the event horizon is at $r = r_s$. But, in the current system, by plugging in $L = M = c = 1$ and solving the quadratic equation, we find only an event horizon at $r = 1$ .This intuitively makes sense; if the black hole gives the particle angular momentum, then it should be able to escape even from a smaller distance. The numerical results confirm the decrease in the radius of the event horizon.

VIII. CONCLUSION

Deciding whether to use an implicit or explicit, single- or multi-step method depends on the system at hand. If $\mathbf{f}(t, \mathbf{y})$ is such that an implicit method can be turned into an explicit one (as we saw in section IIID), then often this is the best course of action because one gets all the benefits of an implicit method without any of the drawbacks - except of course having to initially simplify the system to become explicit. Multi-step methods are generally more accurate, but are slower to execute.

As always, there is no simple answer.

[1] J.C. Butcher. "A History of Runge-Kutta Methods". *Applied Numerical Mathematics*, 1996.

[2] Gautschi, Walter. Numerical analysis. Boston: Birkhauser, 2012. Chapters 5, 6.

[3] Hartle, James B. Gravity: an introduction to Einstein's general relativity. Harlow: Pearson, 2014. Chapter 23.

[4] Yano, Masayuki, James Douglass Penn, George Konidaris, and Anthony T. Patera. "Math, Numerics, & Programming (for Mechanical Engineers)." MIT OCW, Sept. 2012. Web.

[5] Blank, H.J. De. "Guiding Center Motion." Fusion Science and Technology (2004). Web.

[6] Ashby, Neil. "General relativity: Frame-dragging confirmed." Nature 431.7011 (2004): 918-19. Web.

[7] Butcher, J.C. "Numerical Methods for Ordinary Differential Equations in the 20th Century." Journal of Computational and Applied Mathematics 125.1-2 (2000): 1-29.

[8] F.R. Moulton. "New Methods in Exterior Balistics". University of Chicago, Chicago (1926).

```
#### ODE Steppers for y'(t) = f(t, y(t)) #####

##### Secondary Methods ######

function Derivative(x, f, d=1, h=10.0^-8)
    """
    Estimate the derivative of f at x using the center
    finite difference technique with step size h.

    x and f can be vector valued, in which case
    d must be provided. d should be a list of only
    zeros and ones.

    For example, if x and f(x) are vectors of length four
    and we want the derivative with respect to the second
    component in x, then we call Derivative(x, f, [0, 1, 0, 0]).
    """
    (f(x+h*d) - f(x-h*d)) / (2*h)
end

function NewtonsMethod(x, f, error_tol=10.0^-10)
    """
    Estimate the root of f starting from x.

    x and f(x) are vector valued. Newton's method becomes

    \vec{x}_{n+1} = \vec{x}_{n} - inv(Jacobian) * \vec{f(\vec{x}_n)}

    I create the Jacobian by estimating the derivatives
    using a center finite difference method.
    """
    h, n = 10.0^-5, length(x)
    J = zeros(n, n)
    for j=1:n
        d = zeros(n)
        d[j] = 1
        J[:, j] = Derivative(x, f, d)
    end
    count = 0
    while norm(f(x)) > error_tol
        count += 1
        x -= inv(J) * f(x)
        if count > 50
            println("Newton's Method took unusually long so it was terminated prematurely.")
            return x
        end
    end
    x
end

##### End Secondary Methods #####

###### Single Steppers ######

## Step functions that do not include an error estimate ##
# Call these with Stepper(f, method, t0, y0, tf, h)

function EulerStep(f, t, y, h)
    h * f(t, y)
end

function ImprovedEulerStep(f, t, y, h)
    f0 = f(t, y)
    (h / 2.0) * (f0 + f(t + h, y + h * f0))
end
```

```
function BackwardEulerStep(f, t, y, h)
    """ Implicit Backward Euler Step """
    #Use Euler step as starting point for NewtonsMethod.
    NewtonsMethod(f(t, y), f1 -> h*f(t+h, y+f1) - f1)
end

function GL2Step(f, t, y, h)
    """ Implicit Gauss-Legendre Order 2 Method """
    #Use Euler step as starting point for NewtonsMethod.
    NewtonsMethod(f(t, y), f1 -> h*f(t+h/2.0, y+f1/2.0) - f1)
end

function MidpointStep(f, t, y, h)
    h * f(t + h / 2, y + (h / 2) * f(t, y))
end

function CrankNicholsonStep(f, t, y, h)
    """ Second order implicit """
    f1 = f(t, y)
    NewtonsMethod(f1, f2 -> (h/2.0) * (f1 + f(t+h, y+f2)) - f2)
end

function RK4Step(f, t, y, h)
    f1 = f(t, y)
    f2 = f(t + h / 2, y + (h / 2) * f1)
    f3 = f(t + h / 2, y + (h / 2) * f2)
    f4 = f(t + h, y + h * f3)
    (h / 6) * (f1 + f4 + 2 * (f2 + f3))
end

function Stepper(f, method, t0, y0, tf, h)
    """
    The step function `method` returns just the step,
    not an error estimate.
    """
    ts, ys = Float64[], Vector{Float64}[]
    while t0 <= tf
        push!(ts, t0)
        push!(ys, y0)
        y0 += method(f, t0, y0, h)
        t0 += h
    end
    ts, ys
end

## End step functions that do not include an error estimate ##

## Step functions that include an error estimate ##
# Call these with AdaptiveStepper(f, method, t0, y0, tf)

function SimpleErrorStep(f, t, y, h)
    """ Compare implicit Crank Nicholson to Midpoint to get error """
    step = CrankNicholsonStep(f, t, y, h)
    step, norm(step - MidpointStep(f, t, y, h))
end

function RKTrapStep(f, t, y, h)
    """ Implicit Runge-Kutta Trapazoidal Method """
    f1 = h * f(t, y)
    f2 = NewtonsMethod(f1, f2 -> h*f(t+h, y+f1/2.0+f2/2.0) - f2)

    soln = f1/2.0 + f2/2.0
    soln, norm(soln - f1)
end
```

```
function DopriStep(f, t, y, h)
    """ Dormand-Prince Method """
    f1 = h * f(t, y)
    f2 = h * f(t + h/5.0, y + f1/5.0)
    f3 = h * f(t + 3*h/10.0, y + 3*f1/40.0 + 9*f2/40.0)
    f4 = h * f(t + 4*h/5.0, y + 44*f1/45.0 - 56*f2/15.0 + 32*f3/9.0)
    f5 = h * f(t + 8*h/9.0, y + 19372*f1/6561.0 - 25360*f2/2187.0 + 64448*f3/6561.0 - 212*f4/729.0)
    f6 = h * f(t + h, y + 9017*f1/3168.0 - 355*f2/33.0 + 46732*f3/5247.0 + 49*f4/176.0 -
    5103*f5/18656.0)
    soln = 35*f1/384.0 + 500*f3/1113.0 + 125*f4/192.0 - 2187*f5/6784.0 + 11*f6/84.0
    f7 = h * f(t + h, y + soln)
    alt = 5179*f1/57600.0 + 7571*f3/16695.0 + 393*f4/640.0 - 92097*f5/339200.0 + 187*f6/2100.0 +
    f7/40.0

    soln, norm(soln - alt)
end

function CarpStep(f, t, y, h)
    """ Cash-Karp Method """
    f1 = h * f(t, y)
    f2 = h * f(t + h/5.0, y + f1/5.0)
    f3 = h * f(t + 3*h/10, y + 3*f1/40.0 + 9*f2/40.0)
    f4 = h * f(t + 3*h/5.0, y + 3*f1/10.0 - 9*f2/10.0 + 6*f3/5.0)
    f5 = h * f(t + h, y - 11*f1/54.0 + 5*f2/2.0 - 70*f3/27.0 + 35*f4/27.0)
    f6 = h * f(t + 7*h/8.0, y + 1631*f1/55296 + 175*f2/512 + 575*f3/13824 + 44275*f4/110592.0 +
    253*f5/4096.0)
    soln = 37*f1/378.0 + 250*f3/621.0 + 125*f4/594.0 + 512*f6/1771.0
    alt = 2825*f1/27648.0 + 18575*f3/48384.0 + 13525*f4/55296.0 + 277*f5/14336.0 + f6/4.0

    soln, norm(soln - alt)
end

function RKFStep(f, t, y, h)
    """ Runge-Kutta-Fehlberg Method """
    f1 = h * f(t, y)
    f2 = h * f(t + h/4.0, y + f1/4.0)
    f3 = h * f(t + 3*h/8.0, y + 3*f1/32.0 + 9*f2/32.0)
    f4 = h * f(t + 12*h/13.0, y + 1932*f1/2197.0 - 7200*f2/2197.0 + 7296*f3/2197.0)
    f5 = h * f(t + h, y + 439*f1/216.0 - 8*f2 + 3680*f3/513.0 - 845*f4/4104.0)
    f6 = h * f(t + h/2.0, y - 8*f1/27.0 + 2*f2 - 3544*f3/2565.0 + 1859*f4/4104.0 - 11*f5/40.0)
    soln = 16*f1/135.0 + 6656*f3/12825.0 + 28561*f4/56430.0 - 9*f5/50.0 + 2*f6/55.0
    alt = 25*f1/216.0 + 1408*f3/2565.0 + 2197*f4/4104.0 - f5/5.0

    soln, norm(soln - alt)
end

function AdaptiveStepper(f, method, t0, y0, tf, error_tol=10.0^-7)
    """
    The step function `method` must return the tuple,
    (step, error estimate).
    """
    ts, ys = Float64[t0], Vector{Float64}[y0]
    h, totalerror = 0.001, 0
    while t0 < tf
        if h < 10.0^-8 && h < tf - t0
            println("Step size effectively zero at t = ", t0)
            h = 0.001
        else
            if h > tf - t0
                h = tf - t0
            end
            m, error_est = method(f, t0, y0, h)
            if error_est > error_tol
                h *= .75
```

```
            else
                t0 += h
                y0 += m
                push!(ts, t0)
                push!(ys, y0)
                if error_est < error_tol / 10.0
                    h *= 1.2
                end
                totalerror += error_est
            end
        end
    end
    println("Estimated upper bound on total error of ", method, ": ", totalerror)
    ts, ys
end

## End step functions that include an error estimate ##

##### End Single Steppers #####

##### Multi Steppers ######
#Call these with MultiStepper(f, method, t0, y0, tf, h)

function AdamsBash2(f, t, y, h)
    """ Adams-Bashforth Order 2 explicit method """
    y0n = Array[y, y + RK4Step(f, t, y, h)]
    function g(t, yn)
        #yn = Array[y0, y1]
        y0, y1 = yn
        y2 = y1 + h * (1.5 * f(t + h, y1) - 0.5 * f(t, y0))
        Array[y1, y2]
    end
    g, y0n
end

function AdamsBash3(f, t, y, h)
    """ Adams-Bashforth Order 3 explicit method """
    y1 = y + RK4Step(f, t, y, h)
    y2 = y1 + RK4Step(f, t+h, y1, h)
    y0n = Array[y, y1, y2]
    function g(t, yn)
        #yn = Array[y0, y1, y2]
        y0, y1, y2 = yn
        y3 = y2 + h * (23.0 / 12.0 * f(t+2*h, y2) - 4.0 / 3.0 * f(t+h, y1) + 5.0 / 12.0 * f(t, y0))
        Array[y1, y2, y3]
    end
    g, y0n
end

function AdamsBash4(f, t, y, h)
    """ Adams-Bashforth Order 4 explicit method """
    y1 = y + RK4Step(f, t, y, h)
    y2 = y1 + RK4Step(f, t+h, y1, h)
    y3 = y2 + RK4Step(f, t+2*h, y2, h)
    y0n = Array[y, y1, y2, y3]
    function g(t, yn)
        #yn = Array[y0, y1, y2, y3]
        y0, y1, y2, y3 = yn
        y4 = y3+h * (55.0 / 24.0 * f(t+3*h, y3) - 59.0 / 24.0 * f(t+2*h, y2) + 37.0 / 24.0 * f(t+h, y1)
        - 3.0 / 8.0 * f(t, y0))
        Array[y1, y2, y3, y4]
    end
    g, y0n
end
```

```
function AdamsMoult2(f, t, y, h)
    """ Adams-Moulton Order 2 implicit method """
    y1 = y + RK4Step(f, t, y, h)
    y0n = Array[y, y1]
    function g(t, yn)
        y0, y1 = yn
        f0, f1 = f(t, y0), f(t+h, y1)
        y2 = NewtonsMethod(y1, y2 -> y1+h*(5*f(t+2*h, y2)/12.0 + 2*f1/3.0 - f0/12.0) - y2)
        Array[y1, y2]
    end
    g, y0n
end

function AdamsMoult3(f, t, y, h)
    """ Adams-Moulton Order 3 implicit method """
    y1 = y + RK4Step(f, t, y, h)
    y2 = y1 + RK4Step(f, t+h, y1, h)
    y0n = Array[y, y1, y2]
    function g(t, yn)
        y0, y1, y2 = yn
        f0, f1, f2 = f(t, y0), f(t+h, y1), f(t+2*h, y2)
        y3 = NewtonsMethod(y2, y3 -> y2+h*(3*f(t+3*h, y3)/8.0 + 19*f2/24.0 - 5*f1/24.0 + f0/24.0)-y3)
        Array[y1, y2, y3]
    end
    g, y0n
end

function AdamsMoult4(f, t, y, h)
    """ Adams-Moulton Order 4 implicit method """
    y1 = y + RK4Step(f, t, y, h)
    y2 = y1 + RK4Step(f, t+h, y1, h)
    y3 = y2 + RK4Step(f, t+2*h, y2, h)
    y0n = Array[y, y1, y2, y3]
    function g(t, yn)
        y0, y1, y2, y3 = yn
        f0, f1, f2, f3 = f(t, y0), f(t+h, y1), f(t+2*h, y2), f(t+3*h, y3)
        y4 = NewtonsMethod(y3, y4 -> y3+h*(251*f(t+4*h, y4)/720.0+646*f3/720.0 - 264*f2/720.0 +
        106*f1/720.0 - 19*f0/720.0)-y4)
        Array[y1, y2, y3, y4]
    end
    g, y0n
end

function MultiStepper(f, method, t0, y0, tf, h)
    """
    method returns g(t, yn), y0n with y0n is
    an array  of previous values of y.
    """
    g, y0n = method(f, t0, y0, h)
    #y0n = Array[y0, y1, ..., yn]
    ts, ys = Float64[], Vector{Float64}[]
    while t0 <= tf + h
        push!(ts, t0)
        push!(ys, y0n[1])
        y0n = g(t0, y0n)
        t0 += h
    end
    ts, ys
end

##### End Multi Steppers #####


#### Test ####
function func(t, y)
```

```
    y1, ydot = y
    [ydot, -t*y1]
end

t, tmax, h = 0.0, 20, 0.01
y0 = [-1.0, 0.0]

#xs, ys = MultiStepper(func, AdamsMoult4, t, copy(y0), tmax, h)
xs, ys = Stepper(func, GL2Step, t, copy(y0), tmax, h)
#xs, ys = AdaptiveStepper(func, RKTrapStep, t, copy(y0), tmax, h)
figure(0)
plot(xs, [ys[i][1] for i=1:length(ys)], "o-", label="Position")
plot(xs, [ys[i][2] for i=1:length(ys)], "x--", label="Velocity")
legend()


xs, ys = AdaptiveStepper(func, RKFStep, t, copy(y0), tmax)
figure(1)
plot(xs, [ys[i][1] for i=1:length(ys)], "o-", label="Position")
plot(xs, [ys[i][2] for i=1:length(ys)], "x--", label="Velocity")
legend()
```

```
function Test_AdaptiveSteppers(t0, tf)
    function func(t, y)
        [exp(t)]
    end

    y0, exact = [exp(t0)], exp(tf)
    methods = DopriStep, CarpStep, RKFStep

    for a=methods
        ts, ys = AdaptiveStepper(func, a, t0, copy(y0), tf)
        println("Actual error of ", a, ": ", abs(exact - ys[end][1]))
    end
end

#Test_AdaptiveSteppers(20, 21)

#### Systems ####

function SimpleSystem(t, y)
    [cos(t)]
end

function HarmonicOscillator(t, y)
    y1, ydot = y
    [ydot, 100*cos(20*t) - ydot/4.0 - y1]
end

function FourthOrderSystem(t, y)
    y1, ydot, yddot, ydddot = y
    [ydot, yddot, ydddot, exp(t)*cos(100*t)-7*ydddot-17*yddot-17*ydot-6*y1]
end


#t0, tf = 0, 15
#y0 = [0, 0, 0, 0]

#ts, ys = Stepper(FourthOrderSystem, BackwardEulerStep, t0, y0, tf, 0.01)
#ts, ys = MultiStepper(FourthOrderSystem, AdamsBash4, t0, y0, tf, 0.001)
#plot(ts, [a[1] for a=ys], "o-")
```

```
function KapitzaPendulum(;l=1, g=9.8, ax=0, ay=0, omegax=0, omegay=0, phix=0, phiy=0)
    """
    A Kapitza Pendulum function defined by the parameters.

    :param l: length.
    :param g: acceleration due to gravity.

    :params ax, omegax, phix: the x coordinate of the pivot xp(t) = ax cos(omegax t + phix).
    :params ay, omegay, phiy: the y coordinate of the pivot yp(t) = ay cos(omegay t + phiy).

    :return: the function f where \vec{theta}'(t) = f(t, \vec{theta(t)}), x, y.
    """
    function f(t, theta)
        """
        For [theta'(t), theta''(t)] = f(t, [theta(t), theta'(t)]).

        :param t: time.
        :param theta: vector [theta(t), theta'(t)].

        :return: \vec{theta'(t)}.
        """
        th, thdot = theta
        [thdot, ax/l*omegax^2*cos(omegax*t+phix)*cos(th)-(g-ay*omegay^2*cos(omegay*t+phiy))*sin(th)/l]
    end
    function x(t, theta)
        """ x(t) = xp(t) + l sin(theta(t)) """
        ax*cos(omegax*t+phix)+l*sin(theta)
    end
    function y(t, theta)
        """ y(t) = yp(t) - y cos(theta(t)) """
        ay*cos(omegay*t+phiy)-l*cos(theta)
    end

    f, x, y
end

#Default arguments represents simple pendulum.
gfunc, x, y = KapitzaPendulum(ax=.2, omegax=100, ay=.2, omegay=100, phiy=pi/2.0)
t, tmax, h = 0.0, 2, 0.0001
y0 = [pi/4.0, 0.0]
ts, theta = MultiStepper(gfunc, AdamsMoult4, t, y0, tmax, h)

using PyPlot

# Theta plot
figure(0)
plot(ts, [theta[i][1] for i=1:length(theta)], "o-", label="Position")
plot(ts, [theta[i][2] for i=1:length(theta)], "x--", label="Velocity")
xlabel("time t")
ylabel("angle theta (radians)")
title("Kapitza Pendulum")
legend()

# Parametric plot
figure(1)
xaxis = [x(ts[i], theta[i][1]) for i=1:length(ts)]
yaxis = [y(ts[i], theta[i][1]) for i=1:length(ts)]
plot(xaxis, yaxis, "o-", label="Parametric Plot")
title("Kapitza Pendulum")
xlabel("x")
ylabel("y")
legend()
```

```
function geodesic(christoffel)
    """
    Returns the function f where \vec{x}'(tau) = f(tau, \vec{x})
    for a geodesic in the spacetime defined by the christoffel symbols.
    """
    function xddot(mu, xdot, affine)
        sum = 0
        for a=1:4
            for b=1:4
                sum -= affine[mu, a, b] * xdot[a] * xdot[b]
            end
        end
        sum
    end
    function func(tau, y)
        """
       tau is proper time
       y is [t, r, theta, phi, tdot, rdot, thetadot, phidot]
       """
        affine = christoffel(y[1:4])
        f = zeros(8)
        f[1], f[2], f[3], f[4] = y[5:8]
        for mu=1:4
            f[mu+4] = xddot(mu, y[5:8], affine)
        end
        f
    end
    func
end


function Schwarzschild(;G=1, M=1, c=1)
    """
    :return: the function f where \vec{x}'(tau) = f(tau, \vec{x(t)}).
    """
    function Christoffel(x)
        """
       Computes all the Christoffel symbols for the Schwartzschild
       metric at t, r, theta, phi.
       x = [t, r, theta, phi].
       """
        t, r, theta, phi = x
        Gamma = zeros(4, 4, 4)
        Gamma[1, 2, 1] = G*M / (r*(c^2*r-2*G*M))
        Gamma[2, 1, 1] = G*M*(1-(2*G*M)/(r*c^2)) / r^2
        Gamma[2, 2, 2] = G*M / (2*G*M*r-c^2*r^2)
        Gamma[2, 3, 3] = 2*G*M/c^2 - r
        Gamma[2, 4, 4] = (2*G*M-r*c^2)*sin(theta)^2 / c^2
        Gamma[3, 3, 2] = 1 / r
        Gamma[3, 4, 4] = -cos(theta)*sin(theta)
        Gamma[4, 4, 2] = 1 / r
        Gamma[4, 4, 3] = cot(theta)
        Gamma
    end
    geodesic(Christoffel)
end

gfunc = Schwarzschild()
tau, taumax = 0, 35.56
t0, r0, theta0, phi0 = 0, 10, pi/2.0, 0
#u^mu u_mu = -c^2
tdot0 = sqrt(1-2/r0)^-1
y0 = [t0, r0, theta0, phi0, tdot0, 0, 0, 0]

""" Used AdamsBash4 when running fraction_gwloss. Otherwise, used SimpleErrorStep. """
```

```
#taus, xs = AdaptiveStepper(gfunc, SimpleErrorStep, tau, y0, taumax)
h = 0.01
taus, xs = MultiStepper(gfunc, AdamsBash4, tau, y0, taumax, h)

ts = [a[1] for a=xs]
rs = [a[2] for a=xs]

using PyPlot
plot(ts, rs, "o-", label="coordinate time r(t)")
plot(taus, rs, "o-", label="proper time r(tau)")
xlabel("time")
ylabel("r")
title("Schwarzschild Metric")
legend()


function fraction_gwloss(rs, h)
    energy_radiated = 0
    for i=3:length(rs)-2
        r = rs[i]
        #estimate first, second, and third derivatives.
        rdot = (rs[i+1] - rs[i-1]) / (2*h)
        rddot = (rs[i+1] + rs[i-1] - 2*r) / (h^2)
        rdddot = (rs[i+2] - rs[i-2] - 2*rs[i+1] + 2*rs[i-1]) / (2*h^3)

        power = (8/15.0) * (3*rdot*rddot + r*rdddot)^2
        energy_radiated += power * h
    end

    #1 is the mass energy in natural units.
    energy_radiated / (1 + energy_radiated)
end

println(fraction_gwloss(rs, h))



function Kerr(;G=1, M=1, c=1, L=0)
    """
    :return: the function f where \vec{x}'(tau) = f(tau, \vec{x(t)}).
    """
    function metric(x)
        """
        The Kerr metric. Returns a 4x4 matrix.
        """
        t, r, theta, phi = x
        a = L / (M * c)
        rho = r^2 + a^2 * cos(theta)^2
        delta = r^2 - 2*G*M*r/c^2 + a^2

        g = zeros(4, 4)
        g[1, 1] = - (delta - a^2 * sin(theta)^2) / rho * c^2                    #dt
        g[2, 2] = rho / delta                                                    #dr
        g[3, 3] = rho                                                            #d(theta)
        g[4, 4] = (a^4+r^4+2*r^2*a^2-a^2*delta*sin(theta)^2)*sin(theta)^2 / rho #d(phi)
        g[1, 4] = (delta - r^2 - a^2) * (2 * a * c * sin(theta)^2) / rho        #dt d(phi)
        g[4, 1] = g[1, 4]                                                        #d(phi) dt

        g
    end

    ### Approximations to partial derivatives of metric ###
    #Map indices to the correct derivative metric. This particular
    #metric does not depend on t or phi so their derivatives are just 0.
```

```
    deriv = [
        x -> zeros(4, 4), x -> Derivative(x, metric, [0, 1, 0, 0]),
        x -> Derivative(x, metric, [0, 0, 1, 0]), x -> zeros(4, 4)
    ]

    ### Christoffel symbols ###
    function Christoffel(x)
        """
       Computes all the Christoffel symbols for the Kerr-Newman
       metric at t, r, theta, phi with approximations.
       x = [t, r, theta, phi].
       """
         #Evaluate partial derivative metric with repsect to each coordinate.
         D = [d(x) for d in deriv]
         invmetric, Gamma = inv(metric(x)), zeros(4, 4, 4)
         for i=1:4
             for j=1:4
                 for k=1:4
                     for a=1:4
                         Gamma[i, j, k] += .5*invmetric[a, i]*(D[j][a, k]+D[k][a, j]-D[a][j, k])
                     end
                 end
             end
         end
         Gamma
    end

    geodesic(Christoffel)
end

#Default arguments approximates the Schwarzschild metric.
gfunc = Kerr(L=0)
tau, taumax = 0, 32.78
t0, r0, theta0, phi0 = 0, 10, pi/2.0, 0
#u^mu u_mu = -c^2
tdot0 = sqrt(1-2/r0)^-1

y0 = [t0, r0, theta0, phi0, tdot0, 0, 0, 0]
taus, xs = AdaptiveStepper(gfunc, RKTrapStep, tau, y0, taumax, 10.0^-4)
#taus, xs = AdaptiveStepper(gfunc, CarpStep, tau, y0, taumax)

ts = [a[1] for a=xs]
rs = [a[2] for a=xs]
#thetas = [a[3] for a=xs]
phis = [a[4] for a=xs]

using PyPlot
figure(0)
plot(ts, rs, "o-", label="coordinate time r(t)")
plot(taus, rs, "o-", label="proper time r(tau)")
xlabel("time")
ylabel("r")
title("Kerr Metric with L = 0")
legend(loc=1)


figure(1)
plot(ts, phis, "o-", label="coordinate time phi(t)")
plot(taus, phis, "o-", label="proper time phi(tau)")
xlabel("time")
ylabel("radians")
title("Kerr Metric with L = 0")
legend()
```